How to think like a computer scientist

Allen B. Downey

Java Version, Third Edition

How to think like a computer scientist
Java Version, Third Edition

Contents

Chapter 1

The way of the program

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

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

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

1.1   What is a programming language?

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

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

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

1

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

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

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

> source
> code
>
> interpreter
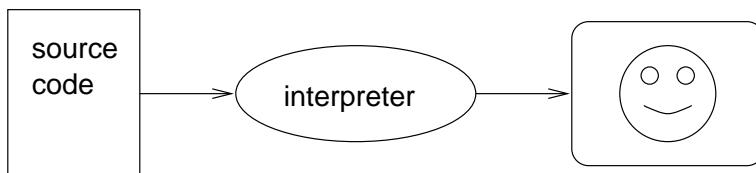
The interpreter   ... and the result
reads the   appears on
source code...   the screen.

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

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

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

> source   object
> code   code
>
> compiler   executor

The compiler   ... and generates   You execute the   ... and the result
reads the   object code.   program (one way   appears on
source code...   or another)...   the screen.

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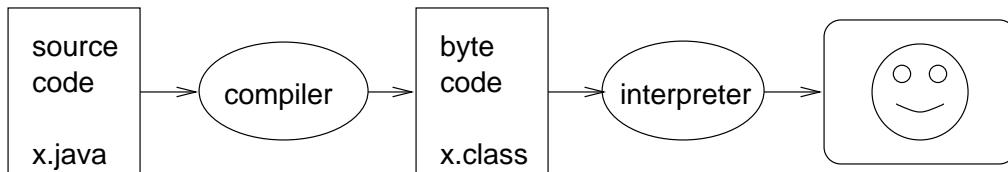
| The compiler reads the source code... | ... and generates Java byte code. | A Java interpreter reads the byte code... | ... and the result appears on the screen. |

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

1.2 What is a program?

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

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

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

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

math: Perform basic mathematical operations like addition and multiplication.

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

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

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

1.3 What is debugging?

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

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

1.3.1 Compile-time errors

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

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

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

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

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

1.3.2 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.

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

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

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

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

1.3.4 Experimental debugging

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

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

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

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

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

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

1.4 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 some order on them); they evolved naturally.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.5 The first program

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

```
class Hello {

  // main: generate some simple output

  public static void main (String[] args) {
    System.out.println ("Hello, world.");
  }
}
```

Some people judge the quality of a programming language by the simplicity of the "Hello, World." program. By this standard, Java does not do very well. Even the simplest program contains a number of features that are hard to explain to beginning programmers. We are going to ignore a lot of them for now, but I will explain a few.

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

```
class CLASSNAME {

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

Here `CLASSNAME` indicates an arbitrary name that you make up. The class name in the example is `Hello`.

In the second line, you should ignore the words `public static void` for now, but notice the word `main`. `main` is a special name that indicates the place in the program where execution begins. When the program runs, it starts by executing the first statement in `main` and it continues, in order, until it gets to the last statement, and then it quits.

There is no limit to the number of statements that can be in `main`, but the example contains only one. It is a **print** statement, meaning that it prints a message on the screen. It is a bit confusing that "print" sometimes means "display something on the screen," and sometimes means "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 command that prints things on the screen is `System.out.println`, and the thing between the parentheses is the thing that will get printed. At the end of the statement there is a semi-colon (`;`), which is required at the end of every statement.

There are a few other things you should notice about the syntax of this program. First, Java uses squiggly-braces (`{` and `}`) to group things together. The outermost squiggly-braces (lines 1 and 8) contain the class definition, and the inner braces contain the definition of `main`.

Also, notice that line 3 begins with `//`. This indicates that this line contains a **comment**, which is a bit of English text that you can put in the middle of a program, usually to explain what the program does. When the compiler sees a `//`, it ignores everything from there until the end of the line.

In the first lab you will compile and run this program, and also modify it in various ways in order to see what the syntax rules are, and to see what error messages the compiler generates when you violate one.

1.6 Glossary

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

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

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

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

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

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

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

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

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

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

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

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

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

bug: An error in a program.

syntax: The structure of a program.

semantics: The meaning of a program.

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

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

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

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

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

Chapter 2

Variables and types

2.1 More printing

As I mentioned in the last chapter, you can put as many statements as you want in `main`. For example, to print more than one line:

```
class Hello {

  // main: generate 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
  }
}
```

Also, as you can see, it is legal to 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 letters. Actually, strings can contain any combination of letters, numbers, punctuation marks, and other special characters.

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

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

```
class Hello {

  // main: generate some simple output
```

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

In this case the output appears on a single line as `Goodbye, cruel world!`.
Notice that there is a space between the word "Goodbye" and the second quo-
tation 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:

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

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:

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

That would work, too, although you have probably noticed that 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 syntax
errors.

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 that stores a **value**.
Values are things that can be printed and stored and (as we'll see later) operated
on. The strings we have been printing (`"Hello, World."`, `"Goodbye, "`, etc.)
are values.

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

```
  String fred;
```

This statement is a **declaration**, because it declares that the variable named `fred` 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 it will probably come as no surprise that the `String` type can store strings.

You will notice that 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:

```
String firstName;
String lastName;
int hour, minute;
```

you could probably make a good guess at 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).

2.3 Assignment

Now that we have created some variables, we would like to store values in them. We do that with an **assignment statement**.

```
fred = "Hello.";      // give fred the value "Hello."
hour = 11;            // assign the value 11 to hour
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:
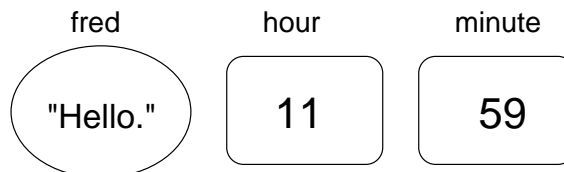
For these diagrams I will use different shapes to indicate different variable types. These shapes should help remind you that one of the rules in Java is that a variable has to have the same type as the value you assign it. You cannot store a `String` in `minute` or an integer in `fred`. It would be like putting a square peg in a round hole.

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. But for now you should remember that as a general rule variables and values have the same type, and we'll talk about special cases later.

Another source of confusion is that some strings *look* like integers, but they are not. For example, `fred` 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.

```
fred = "123";     // this is legal
fred = 123;       // this is not legal
```

2.4   Printing variables

You can print the value of a variable using the same commands we used to print `String`s.

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

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");`

If you want to get a little tricky, you could write

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

The output of this program is

```
The value of firstLine is Hello, again!
```

I am pleased to report that the syntax for printing a variable is the same regardless of the variable's type.

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

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

2.5   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://java.sun.com/docs/books/tutorial/java/nutsandbolts/keywords.html`

This WWW site, provided by Sun, includes Java documentation I will be referring to throughout this book.

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

2.6   Operators

Operators are special symbols that are used to represent simple computations like addition and multiplication. Most of the operators in Java do exactly what you would expect them to do, because they are common mathematical symbols. For example, the operator for adding two integers is `+`.

The following are all legal Java expressions whose meaning is more or less obvious:

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

Expressions can contain both variable names and numbers. In each case the name of the variable is replaced with its value before the computation is performed.

Addition, subtraction and multiplication all do what you expect, but you might be surprised by division. For example, the following program:

```
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);
```

would generate the following output:

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

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

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

A possible alternative in this case is to calculate a percentage rather than a fraction:

```
System.out.print ("Percentage of the hour that has passed: ");
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. In order to get an even more accurate answer, we could use a different type of variable, called floating-point, that is capable of storing fractional values. We'll get to that in the next chapter.

2.7   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 just to get you started:

- Multiplication and division take precedence (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.

2.8 Operators for Strings

In general you cannot perform mathematical operations on Strings, even if the strings look like numbers. The following are illegal (if we know that fred has type String)

```
fred - 1          "Hello"/123      fred * "Hello"
```

By the way, can you tell by looking at those expressions whether fred 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 Strings, although it does not do exactly what you might expect. For Strings, the + operator represents **concatenation**, which means joining up the two operands by linking them end-to-end. So "Hello, " + "world." yields the string "Hello, world." and fred + "ism" adds the suffix *ism* to the end of whatever fred is, which is often handy for naming new forms of bigotry.

2.9 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** them. For example, we know how to multiply numbers and we know how to print; it turns out we can do both at the same time:

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

Actually, I shouldn't say "at the same time," since in reality the multiplication
has to happen before the printing, but the point is that any expression, involving
numbers, strings, and variables, can be used inside a print statement. We've
already seen one example:

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

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

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

This ability may not seem so impressive now, but we will see other examples
where composition makes it possible to express complex computations neatly
and concisely.

WARNING: There are limits on where you can use certain expressions; most
notably, the left-hand side of an assignment statement 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;`.

2.10   Glossary

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 one type.

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

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

statement: A line of code that represents a command or action. So far, the
statements we have seen are declarations, assignments, and print state-
ments.

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 result value. Expressions also have types, as determined by their
operators and operands.

operator: A special symbol that represents a simple 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 in order to represent complex computations concisely.

Chapter 3

Methods

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`.

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;
```

In fact, this syntax is quite common. A combined declaration and assignment is sometimes called an **initialization**.

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

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

```
int x = 1.1;
```

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

```
double y = 1;
```

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

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

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

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

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

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

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

3.2   Converting from `double` to `int`

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

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

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

```
int x = (int) Math.PI;
```

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

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

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

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

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

3.3 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. For example, $\pi/2$ is approximately 1.571, and $1/x$ is 0.1 (assuming that x is 10).

Then you can evaluate the function itself, either by looking it up in a table or by performing various computations. The sin of 1.571 is 1, and the log of 0.1 is -1 (assuming that log indicates the logarithm base 10).

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 a set of built-in functions that includes most of the mathematical operations you can think of. These functions are called **methods**. Most math methods operate on `doubles`.

The math methods are invoked using a syntax that is similar to the `print` commands we have already seen:

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

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

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

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

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

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

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

3.4 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 the `cos` method. (Notice that `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` function 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.

3.5 Adding new methods

So far we have only been using the methods that are built into Java, but it is also possible to add new methods. Actually, we have already seen one method definition: `main`. The method named `main` is special in that it indicates where the execution of the program begins, but the syntax for `main` is the same as for any other method definition:

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

You can make up any name you want for your method, except that you can't call it `main` or any other Java keyword. The list of parameters specifies what information, if any, you have to provide in order to use (or **invoke**) the new function.

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

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

This method is named `newLine`, and the empty parentheses indicate that it takes no parameters. It contains only a single statement, which prints an empty `String`, indicated by `""`. Printing a `String` with no letters in it may not seem all that useful, except remember that `println` skips to the next line after it prints, so this statement has the effect of skipping to the next line.

In `main` we can invoke this new method using syntax that is similar to the way we invoke the built-in Java commands:

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

The output of this program is

```
First line.
```

```
Second line.
```

Notice the extra space between the two 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 procedure repeatedly. In fact, it is quite common and useful to do so.

- You can have one method invoke another method. In this case, `main` invokes `threeLine` and `threeLine` invokes `newLine`. Again, this is common and useful.

- 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). On the other hand, it is usually a better idea to put each statement on a line by itself, to make your program easy to read. I sometimes break that rule in this book to save space.

So far, it may not be clear why it is worth the trouble to create all these new methods. Actually, there are a lot of reasons, but this example only 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 command, 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, how would you print nine consecutive new lines? You could just invoke `threeLine` three times.

3.6   Classes and methods

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

```
class NewLine {

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

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

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

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. It contains methods named `sqrt`, `sin`, and many others. When we invoke a mathematical function, we have to specify the name of the class (`Math`) and the name of the function. That's why the syntax is slightly different for built-in methods and the methods that we write:

```
Math.pow (2.0, 10.0);
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 (correctly) is in the `NewLine` class, which is what we are writing.

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

```
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.

3.7 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 case 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 the built-in method `println`, which causes yet another detour. Fortunately, Java is quite 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.

Actually, 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 Java 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.

3.8   Parameters and arguments

Some of the built-in methods we have used have **parameters**, which are values that you provide to let the method do its job. For example, if you want to find the sine of a number, you have to indicate what the number is. Thus, `sin` takes a `double` value as a parameter. To print a string, you have to provide the string, which is why `println` takes a `String` as an parameter.

Some methods take more than one parameter, like `pow`, which takes two `doubles`, the base and the exponent.

Notice that in each of these cases we have to specify not only how many parameters there are, but also what type they are. So it shouldn't surprise you that when you write a class definition, the parameter list indicates the type of each parameter. For example:

```
public static void printTwice (String phil) {
  System.out.println (phil);
  System.out.println (phil);
}
```

This method takes a single parameter, named `phil`, that has type `String`. Whatever that parameter is (and at this point we have no idea what it is), it gets printed twice. I chose the name `phil` to suggest that the name you give a parameter is up to you, but in general you want to choose something more illustrative than `phil`.

In order to invoke this method, we have to provide a `String`. For example, we might have a `main` method like this:

```
public static void main (String[] args) {
  printTwice ("Don't make me say this twice!");
}
```

The string you provide is called an **argument**, and we say that the argument is **passed** to the method. In this case we are creating a string value that contains the text "Don't make me say this twice!" and passing that string as an argument to `printTwice` where, contrary to its wishes, it will get printed twice.

Alternatively, if we had a `String` variable, we could use it as an argument instead:

```
public static void main (String[] args) {
  String argument = "Never say never.");
  printTwice (argument);
}
```

Notice something very important here: the name of the variable we pass as an argument (`argument`) has nothing to do with the name of the parameter (`phil`). Let me say that again:

> **The name of the variable we pass as an argument has nothing to do with the name of the parameter.**

They can be the same or they can be different, but it is important to realize that they are not the same thing, except that they happen to have the same value (in this case the string `"Never say never."`).

The value you provide as an argument must have the same type as the parameter of the method you invoke. This rule is very important, but it often gets complicated in Java for two reasons:

- There are some methods that can accept arguments with many different types. For example, you can send *any* type to `print` and `println`, and it will do the right thing no matter what. This sort of thing is an exception, though.

- If you violate this rule, the compiler often generates a confusing error message. Instead of saying something like, "You are passing the wrong kind of argument to this method," it will probably say something to the effect that it could not find a method with that name that would accept an argument with that type. Once you have seen this error message a few times, though, you will figure out how to interpret it.

One last thing you should realize is that parameters and other variables only exist inside their own methods. Within the confines of `main`, there is no such thing as `phil`. If you try to use it, the compiler will complain. Similarly, inside `printTwice` there is no such thing as `argument`.

3.9   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 `int hour, minute`, but that format is only legal for variable declarations, not for parameters.

Another common source of confusion is that you do not have to declare the types of arguments. 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 unnecessary and illegal to include the type when you pass them as arguments. The correct syntax is `printTime (hour, minute)`.

3.10   Methods with results

You might have noticed by now that some of the methods we are using, like the `Math` methods, yield results. Other methods, like `println` and `newLine`, perform some 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 yield results, or are we stuck with things like `newLine` and `printTwice`?

The answer to the third question is "yes, you can write methods that return values," and we'll do it in a couple of chapters. I will 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.

3.11   Glossary

floating-point: A type of variable (or value) that can contain fractions as well as integers. In Java this type is called `double`.

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 some useful function. Methods may or may not take parameters, and may or may not produce a result.

parameter: A piece of information you provide in order to invoke a method. Parameters are like variables in the sense that they contain values and have types.

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

invoke: Cause a method to be executed.

Chapter 4

Conditionals, graphics and recursion

4.1 The modulus operator

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

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

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

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

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

4.2 Conditional execution

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

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

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

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

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

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

The two sides of a condition operator have to be the same type. You can only compare `ints` to `ints` and `doubles` to `doubles`. Unfortunately, at this point you can't compare `Strings` at all! There is a way to compare `Strings`, but we won't get to it for a couple of chapters.

4.3   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:

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

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 set of statements is executed. Since the condition must be true or false, exactly one of the alternatives will be executed.

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:

```
public static void printParity (int x) {
  if (x%2 == 0) {
    System.out.println ("x is even");
```

```
  } else {
    System.out.println ("x is odd");
  }
}
```

Now you have a method named `printParity` that will print an appropriate message for any integer you care to provide. In `main` you would invoke this method as follows:

```
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:

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

4.4   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:

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

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

4.5   Nested conditionals

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

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

```
      System.out.println ("x is positive");
    } else {
      System.out.println ("x is negative");
    }
  }
```

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

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

On the other hand, this kind of **nested structure** is common, and we will see it again, so you better get used to it.

4.6   The return statement

The `return` statement allows you to terminate the execution of a method before you reach the end. One reason to use it is if you detect an error condition:

```
public static void printLogarithm (double x) {
  if (x <= 0.0) {
    System.out.println ("Positive numbers only, please.");
    return;
  }

  double result = Math.log (x);
  System.out.println ("The log of x is " + result);
}
```

This defines a method named `printLogarithm` that takes a `double` named x as a parameter. The first thing it does is check 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 caller and the remaining lines of the method are not executed.

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

4.7   Type conversion

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

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

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

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

4.8 Slates and Graphics objects

In order to draw things on the screen, you need two objects, a `Slate` and a `Graphics` object.

Slate: a `Slate` is a window that contains a blank rectangle you can draw on. The `Slate` class is not part of the standard Java library; it is something I wrote for this course.

Graphics: the `Graphics` object is the object we will use to draw lines, circles, etc. It is part of the Java library, so the documentation for it is on the Sun web site.

The methods that pertain to `Graphics` objects are defined in the built-in `Graphics` class. The methods that pertain to `Slates` are defined in the `Slate` class, which is shown in Appendix **??**.

The primary method in the `Slate` class is `makeSlate`, which does pretty much what you would expect. It creates a new window and returns a `Slate` object you can use to refer to the window later in the program. You can create more than one `Slate` in a single program.

```
Slate slate = Slate.makeSlate (500, 500);
```

`makeSlate` takes two arguments, the width and height of the screen. Because it belongs to a different class, we have to specify the name of the class using "dot notation."

The return value gets assigned to a variable named `slate`. There is no conflict between the name of the class (with an upper-case "S") and the name of the variable (with a lower-case "s").

The next method we need is `getGraphics`, which takes a `Slate` object and creates a `Graphics` object that can draw on it. You can think of a `Graphics` object as a piece of chalk.

```
Graphics g = Slate.getGraphics (slate);
```

Using the name **g** is conventional, but we could have called it anything.

4.9   Invoking methods on a `Graphics` object

In order to draw things on the screen, you invoke methods on the graphics object. We have invoked lots of methods already, but this is the first time we have "invoked a method on an object." The syntax is similar to invoking a method from another class:

```
g.setColor (Color.black);
g.drawOval (x, y, width, height);
```

The name of the object comes before the dot; the name of the method comes after, followed by the arguments for that method. In this case, the method takes a single argument, which is a color.

 `setColor` changes the current color, in this case to black. Everything that gets drawn will be black, until we use `setColor` again.

 `Color.black` is a special value provided by the `Color` class, just as `Math.PI` is a special value provided by the `Math` class. `Color`, you will be happy to hear, provides a palette of other colors, including:

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

To draw on the `Slate`, we can invoke `draw` methods on the `Graphics` object. For example:

```
g.drawOval (x, y, width, height);
```

`drawOval` takes four integers as arguments. These arguments specify a **bounding box**, which is the rectangle in which the oval will be drawn (as shown in the figure). The bounding box itself is not drawn; only the oval is. The bounding box is like a guideline. Bounding boxes are always oriented horizontally or vertically; they are never at a funny angle.
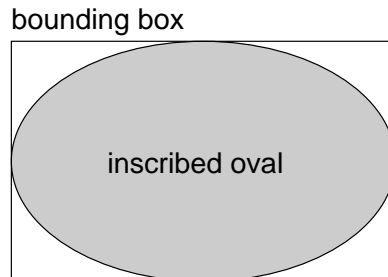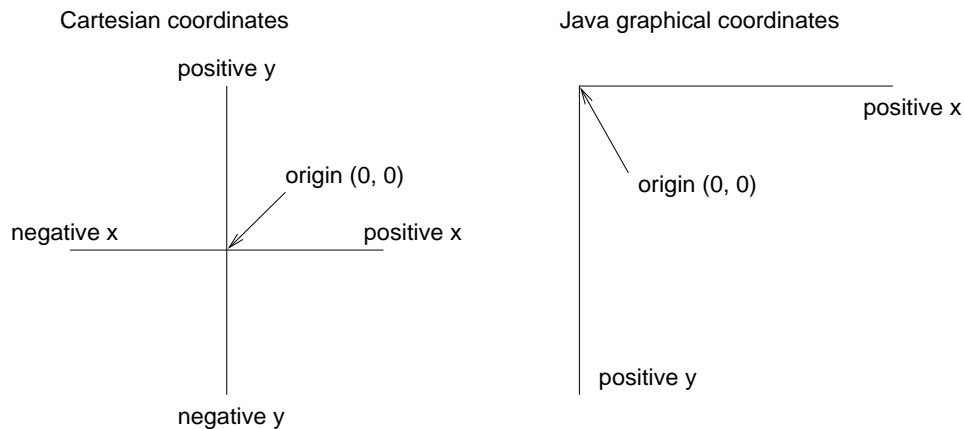
If you think about it, there are lots of ways to specify the location and size of a rectangle. You could give the location of the center or any of the corners, along with the height and width. Or, you could give the location of opposing corners. The choice is arbitrary, but in any case it will require the same number of parameters: four.

By convention, the usual way to specify a bounding box is to give the location of the *upper-left* corner and the width and height. The usual way to specify a location is to use a **coordinate system**.

4.10 Coordinates

You are probably familiar with Cartesian coordinates in two dimensions, in which each location is identified by an x-coordinate (distance along the x-axis) and a y-coordinate. By convention, Cartesian coordinates increase to the right and up, as shown in the figure.

Cartesian coordinates   Java graphical coordinates

positive y   positive x

origin (0, 0)   origin (0, 0)

negative x   positive x

negative y   positive y

Annoyingly, it is conventional for computer graphics systems to use a variation on Cartesian coordinates in which the origin is in the upper-left corner of the screen or window, and the direction of the positive y-axis is *down*. Java follows this convention.

The unit of measure is called a **pixel**; a typical screen is about 1000 pixels wide. Coordinates are always integers. If you want to use a floating-point value as a coordinate, you have to round it off to an integer (See Section **??**).

4.11 A lame Mickey Mouse

Let's say we want to draw a picture of Mickey Mouse. We can use the oval we just drew as the face, and then add ears. Before we do that it is a good idea to break the program up into two methods. `main` will create the `Slate` and `Graphics` objects and then invoke `draw`, which does the actual drawing.

```
public static void main (String[] args) {
  int width = 500;
  int height = 500;

  Slate slate = Slate.makeSlate (width, height);
  Graphics g = Slate.getGraphics (slate);

  g.setColor (Color.black);
  draw (g, 0, 0, width, height);
}

public static void draw
            (Graphics g, int x, int y, int width, int height) {
  g.drawOval (x, y, width, height);
  g.drawOval (x, y, width/2, height/2);
  g.drawOval (x+width/2, y, width/2, height/2);
}
```

The parameters for draw are the Graphics object and a bounding box. draw
invokes drawOval three times, to draw Mickey's face and two ears. The following
figure shows the bounding boxes for the ears.

bounding box of left ear   bounding box of right ear   result

As shown in the figure, the coordinates of the upper-left corner of the bound-
ing box for the left ear are (x, y). The coordinates for the right ear are
(x+width/2, y). In both cases, the width and height of the ears are half the
width and height of the original bounding box.

Notice that the coordinates of the ear boxes are all relative to the location (x
and y) and size (width and height) of the original bounding box. As a result,
we can use draw to draw a Mickey Mouse (albeit a lame one) anywhere on the
screen in any size. As an exercise, modify the arguments passed to draw so that
Mickey is one half the height and width of the screen, and centered.

4.12 Other drawing commands

Another drawing command with the same parameters as `drawOval` is

```
drawRect (int x, int y, int width, int height)
```

Here I am using a standard format for documenting the name and parameters of methods. This information is sometimes called the method's **interface** or **prototype**. Looking at this prototype, you can tell what types the parameters are and (based on their names) infer what they do. Here's another example:

```
drawLine (int x1, int y1, int x2, int y2)
```

The use of parameter names x1, x2, y1 and y2 suggests that `drawLine` draws a line from the point (x1, y1) to the point (x2, y2).

One other command you might want to try is

```
drawRoundRect (int x, int y, int width, int height,
        int arcWidth, int arcHeight)
```

The first four parameters specify the bounding box of the rectangle; the remaining two parameters indicate how rounded the corners should be, specifying the horizontal and vertical diameter of the arcs at the corners.

There are also "fill" versions of these commands, that not only draw the outline of a shape, but also fill it in. The interfaces are identical; only the names have been changed:

```
fillOval (int x, int y, int width, int height)
fillRect (int x, int y, int width, int height)
fillRoundRect (int x, int y, int width, int height,
        int arcWidth, int arcHeight)
```

There is no such thing as `fillLine`—it just doesn't make sense.

4.13 Recursion

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

For example, look at the following method:

```
public 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` (what a trip). So the total output looks like:

```
3
2
1
Blastoff!
```

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

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

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

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

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

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

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

4.14 A fractal Mickey Mouse

If you write recursive graphical methods, you often get interesting shapes called **fractals**. For example, to draw a fractal Mickey Mouse, we can modify draw so that it calls itself recursively:

```
public static void draw
              (Graphics g, int x, int y, int width, int height) {
  if (height == 0) return;

  g.drawOval (x, y, width, height);
  draw (g, x, y, width/2, height/2);
  draw (g, x+width/2, y, width/2, height/2);
}
```

Only a couple of changes are required. First, instead of invoking drawOval to draw the ears, we invoke draw to draw recursive ears; that is, ears that have ears that have ears of their own, and so on. Also, instead of invoking drawOval on the Graphics object g, we just invoke draw and pass g as an argument.

Finally, and most importantly, notice that I added a line at the beginning of the method that checks whether the height is equal to zero, and if it is, it returns immediately without drawing any circles, *and without making any recursive calls*. This is necessary because otherwise we would just draw smaller and smaller circles and the program would never terminate. This is known as infinite recursion, and it is generally not considered a good idea.

Actually, in Java an infinite recursion will eventually cause a StackOverflowException. If you ever get this error, you will know why.

The output of this program looks like this:
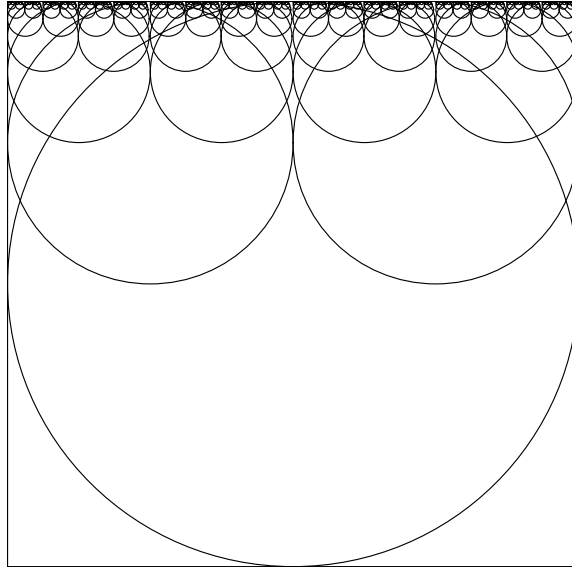
That might not seem too exciting yet, but with only a few modifications, you can get this:
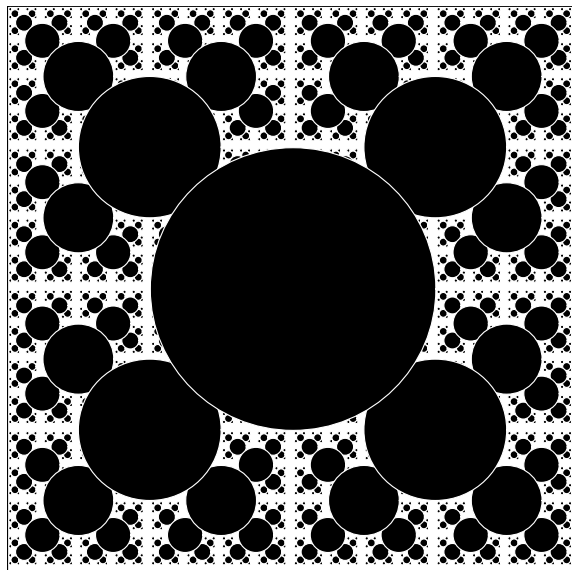
Can you figure out how?

4.15   Stack diagrams for recursive methods

In the previous chapter we used a stack diagram to represent the state of a program during a method call. 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 instance of the method that contains a new version of the method's local variables and parameters.

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

		main
n:	3	countdown
n:	2	countdown
n:	1	countdown
n:	0	countdown
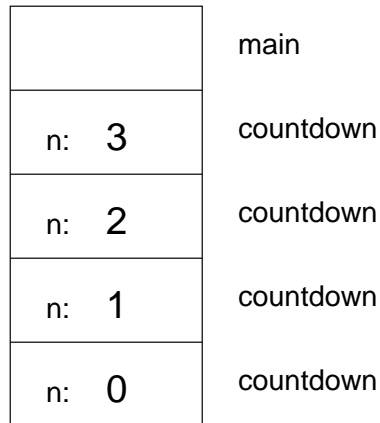
There is one instance of `main` and four instances of `countdown`, each with a different value for the parameter `n`. The bottom of the stack, `countdown` with n=0 is the base case. It does not make a recursive call, so there are no more instances of `countdown`.

The instance of `main` is empty because `main` does not have any parameters or local variables. As an exercise, draw a stack diagram for `nLines`, invoked with the parameter n=4.

4.16   Convention and divine law

In the last few sections, I used the phrase "by convention" several times to indicate design decisions that are arbitrary in the sense that there are no significant reasons to do things one way or another, but dictated by convention.

In these cases, it is to your advantage to be familiar with convention and use it, since it will make your programs easier for others to understand. At the same time, it is important to distinguish between (at least) three kinds of rules:

Divine law: This is my phrase to indicate a rule that is true because of some underlying principle of logic or mathematics, and that is true in any programming language (or other formal system). For example, there is no way to specify the location and size of a bounding box using fewer than four pieces of information. Another example is that adding integers is

commutative. That's part of the definition of addition and has nothing to do with Java.

Rules of Java: These are the syntactic and semantic rules of Java that you cannot violate, because the resulting program will not compile or run. Some are arbitrary; for example, the fact that the + symbol represents addition *and* string concatenation. Others reflect underlying limitations of the compilation or execution process. For example, you have to specify the types of parameters, but not arguments.

Style and convention: There are a lot of rules that are not enforced by the compiler, but that are essential for writing programs that are correct, that you can debug and modify, and that others can read. Examples include indentation and the placement of squiggly braces, as well as conventions for naming variables, methods and classes.

As we go along, I will try to indicate which category various things fall into, but you might want to give it some thought from time to time.

While I am on the topic, you have probably figured out by now that the names of classes always begin with a capital letter, but variables and methods begin with lower case. If a name includes more than one word, you usually capitalize the first letter of each word, as in `newLine` and `printParity`. Which category are these rules in?

4.17   Glossary

modulus: 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.

coordinate: A variable or value that specifies a location in a two-dimensional graphical window.

pixel: The unit in which coordinates are measured.

bounding box: A common way to specify the coordinates of a rectangular area.

typecast: An operator that converts from one type to another. In Java it appears as a type name in parentheses, like (`int`).

interface: A description of the parameters required by a method and their types.

prototype: A way of describing the interface to a method using Java-like syntax.

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

infinite recursion: A method that invokes itself recursively without every reaching the base case. The usual result is a StackOverflowException.

fractal: A kind of image that is defined recursively, so that each part of the image is a smaller version of the whole.

Chapter 5

Fruitful methods

5.1   Return values

Some of the built-in methods we have used, like the Math functions, have produced results. That is, 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 the methods we have written have been **void** methods; that is, methods that return no value. When you invoke a void method, it is typically on a line by itself, with no assignment:

```
nLines (3);
g.drawOval (0, 0, width, height);
```

In this chapter, we are going to write methods that return things, which I will refer to as **fruitful** methods, for want of a better name. 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 indicates that the return value from this method will have type `double`. I still haven't explained what `public static` means, but be patient.

49

Also, notice that the last line is an alternate form of the `return` statement
that includes a return value. This statement means, "return immediately from
this method and use the following expression as a 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.

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 returns statements are in an alternative conditional, only one will
be executed. Although 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!!
}
```

This program is not legal because if x happens to be 0, then neither condition will
be true and the method will end without hitting a return statement. A typical

compiler message would be "return statement required in absoluteValue," which is a confusing message considering that there are already two of them.

5.2 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 one technique that I call **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,

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{5.1}$$

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-keeper that is necessary in order 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 make it more complicated.

In order 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. That way, at any point we know exactly where the error must be—in the last line we added.

The next step in the computation is to find the differences $x_2 - x_1$ and $y_2 - y_1$. I will 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 that will let me check the intermediate values before proceeding. As I mentioned, I already know that they should be 3.0 and 4.0.

When the method is finished I will 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. Sometimes it is a good idea to keep the scaffolding around, but comment it out, just in case you need it later.

The next step in the development is to square dx and dy. We could use the Math.pow method, but it is simpler and faster to just 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 and return the result.

```
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;
}
```

Then in main, we should print and check the value of the result.

As you gain more experience programming, you might find yourself writing and debugging more than one line at a time. Nevertheless, this incremental development process can save you a lot of debugging 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 might want to remove some of the scaffolding or consolidate multiple statements into compound expressions, but only if it does not make the program difficult to read.

5.3 Composition

As you should expect by now, 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 fred
              (double xc, double yc, double xp, double yp) {
  double radius = distance (xc, yc, xp, yp);
  double area = area (radius);
  return area;
}
```

The name of this method is `fred`, which may seem odd. I will explain why in the next section.

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 fred
              (double xc, double yc, double xp, double yp) {
  return area (distance (xc, yc, xp, yp));
}
```

5.4   Overloading

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

If two methods do the same thing, it is natural to give them the same name. In other words, it would make more sense if `fred` were called `area`.

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 can go ahead and rename `fred`:

```
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 a single `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`. More amazing still, the second version of `area` actually invokes the first.

Many of the built-in Java commands 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 `doubles`, and there is also a version for `ints`.

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.

Actually, 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 time 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 for one reason or another you are not running the version of the program you think you are. To check, stick in a `print` statement (it doesn't matter what you print) and make sure the behavior of the program changes accordingly.

5.5 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 `ints` and produces an `int`, or two `doubles` and produces a `double`, etc.

The exceptions we have seen are the **relational operators**, which compare `ints` and `floats` 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 `ints`, `doubles` and `Strings`, those sets are pretty big. For `booleans`, not so big.

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

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

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

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

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

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

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

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

5.6 Logical operators

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

`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 has the effect of negating or inverting a boolean expression, so `!evenFlag` is true if `evenFlag` is false—if the number is odd.

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

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

5.7   Boolean methods

Methods can return boolean values just like any other type, which is often convenient for hiding complicated 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 a bit 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 assigns the value `true` to `bigFlag` only if 17 is *not* a single-digit number. The second line prints `true` because 2 is a single-digit number. Yes, `println` is overloaded to handle booleans, too.

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");
}
```

5.8 More recursion

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

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

To give you an idea of what you can do with the tools we have learned so far, 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 typically not very useful:

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

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

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

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

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

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

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

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

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.

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

If we look at the flow of execution for this program, it is similar to `nLines` from the previous chapter. 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 calls.
> >
> > 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 function calls:
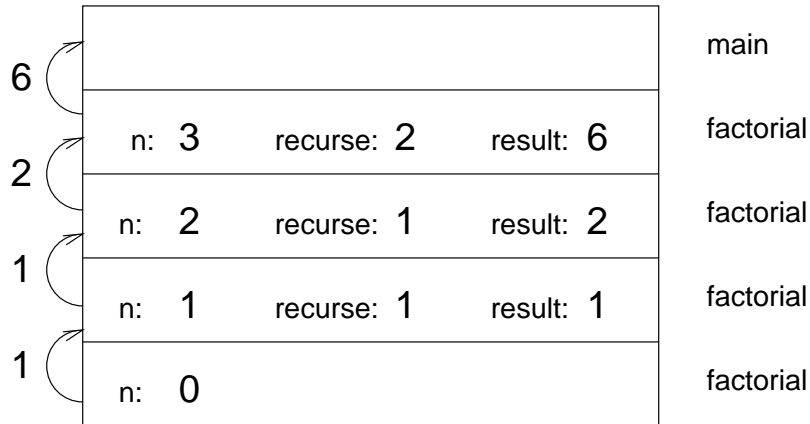
The return values are shown being passed back up the stack.

Notice that in the last instance of `factorial`, the local variables `recurse` and `result` do not exist because when `n=0` the branch that creates them does not execute.

5.9 Leap of faith

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

In fact, you are already practicing this leap of faith when you use built-in methods. When you invoke `Math.cos` or `drawOval`, you don't examine the implementations of those methods. You just assume that they work, because the people who wrote the built-in classes were good programmers.

Well, the same is true when you invoke one of your own methods. For example, in Section **??** we wrote a method called `isSingleDigit` that determines whether a number is between 0 and 9. Once we have convinced 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 (yields the correct result), and then ask yourself, "Assuming that I can find the factorial of $n-1$, can I compute the factorial of n?" In this case, it is clear that you can, by multiplying by n.

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

5.10   One more example

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

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

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

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

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

   Translated into Java, this is

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

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

5.11   Glossary

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.

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.

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

Chapter 6

Iteration

6.1 Multiple assignment

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

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

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

This kind of **multiple assignment** is the reason I described variables as a *container* for values. When you assign a value to a variable, you change the contents of the container, as shown in the figure:
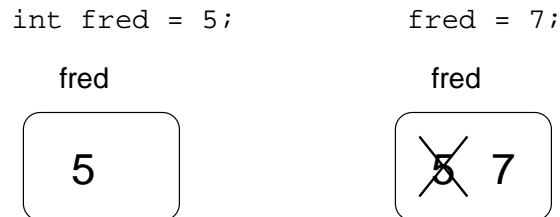
When there are multiple assignments to a variable, it is especially important to distinguish between an assignment statement and a statement of equality. Because Java uses the = symbol for assignment, it is tempting to interpret a statement like `a = b` as a statement of equality. It is not!

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, 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!

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

The third line changes the value of `a` but it does not change the value of `b`, and so they are no longer equal. In many programming languages an alternate symbol is used for assignment, such as `<-` or `:=`, in order to avoid this confusion.

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

6.2   Iteration

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

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

The two features we are going to look at are the `while` statement and the `for` statement.

6.3   The `while` statement

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.4   Tables

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

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

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

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

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

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

The output of this program is

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

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

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

$$\log_2 x = \frac{log_e x}{log_e 2} \tag{6.1}$$

Changing the `print` statement to

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

yields

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

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

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

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

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

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

6.5 Two-dimensional tables

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

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

```
int i = 1;
while (i <= 6) {
  System.out.print (2*i + "   ");
  i = i + 1;
}
System.out.println ("");
```

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

As I mentioned in Section **??**, 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**.

6.6   Encapsulation and generalization

Encapsulation usually means taking a piece of code and wrapping it up in a method, allowing you to take advantage of all the things methods are good for. We have seen two examples of encapsulation, when we wrote printParity in Section **??** and isSingleDigit in Section **??**.

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

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

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

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

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

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

and with argument 4, the output is

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

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

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

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

The output of this program is

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

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

6.7   Methods

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

- By giving a name to a sequence of statements, you make your program easier to read and debug.

- Dividing a long program into methods allows you to separate parts of the program, debug them in isolation, and then compose them into a whole.

- Methods facilitate both recursion and iteration.

- Well-designed methods are often useful for many programs. Once you write and debug one, you can reuse it.

6.8   More encapsulation

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 process I am demonstrating is a common development plan. You develop code gradually by adding lines to main or someplace else, and then when you get it working, you extract it and wrap it up in a method.

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

6.9   Local variables

About this time, you might be wondering 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, and changing the value of one of them has no effect on the other.

Variables that are declared inside a method definition are called **local variables** because they are local to their own methods. 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.

It is often a good idea to use different variable names in different methods, to avoid confusion, but 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 will probably make the program harder to read.

6.10   More generalization

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

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

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

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

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

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

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

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

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

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

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

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

to

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

and you get

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

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

6.11   Glossary

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

infinite loop: A loop whose condition is always true.

body: The statements inside the loop.

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

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

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

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

development plan: A process for developing a program. In this chapter, I demonstrated a style of development based on developing code to do simple, specific things, and then encapsulating and generalizing. In Section **??** I demonstrated a technique I called incremental development. In later chapters I will suggest other styles of development.

Chapter 7

Strings and things

7.1 Invoking methods on objects

In Section **??** we used a Graphics object to draw circles in a window, and I used the phrase "invoke a method on an object," to refer to the statements like

```
g.drawOval (0, 0, width, height);
```

In this case `drawOval` is the method being invoked on the object named `g`. At the time I didn't provide a definition of **object**, and I still can't provide a complete definition, but it is time to try.

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.

So far we have only seen one object, `g`, so this definition might not mean much yet. Another example is `String`s. `String`s are objects (and `int`s and `double`s are not). Based on the definition of object, you might ask "What is the data contained in a `String` object?" and "What are the methods we can invoke on `String` objects?"

The data contained in a `String` object are the letters of the string. There are quite a few methods the operate on `String`s, but I will only use a few in this book. The rest are documented at

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

The first method we will look at is `charAt`, which allows you to extract letters from a `String`. In order to store the result, we need a variable type that can store individual letters (as opposed to strings). Individual letters are called characters, and the variable type that stores them is called `char`.

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

```
char fred = 'c';
if (fred == 'c') {
```

```
    System.out.println (fred);
}
```

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

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

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

The syntax `fruit.charAt` indicates that I am invoking the `charAt` method on the object named `fruit`. I am passing the argument 1 to this method, which indicates that I would like to know the first letter of the string. The result is a character, which is stored in a `char` named `letter`. When I print the value of `letter`, I get a surprise:

```
a
```

`a` is not the first letter of `"banana"`. Unless you are a computer scientist. For perverse reasons, computer scientists always start counting from zero. The 0th letter ("zeroeth") of `"banana"` is `b`. The 1th letter ("oneth") is `a` and the 2th ("twoeth") letter is `n`.

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

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

7.2   Length

The second `String` method we'll look at is `length`, which returns the number of characters in the string. For example:

```
int length = fruit.length();
```

`length` takes no arguments,as indicated by (), and returns an integer, in this case 6. Notice that it is legal to have a variable with the same name as a method (although it can be confusing for human readers).

To find the last letter of a string, you might be tempted to try something like

```
int length = fruit.length();
char last = fruit.charAt (length);        // WRONG!!
```

That won't work. The reason is that there is no 6th letter in `"banana"`. Since we started counting at 0, the 6 letters are numbered from 0 to 5. To get the last character, you have to subtract 1 from `length`.

```
int length = fruit.length();
char last = fruit.charAt (length-1);
```

7.3 Traversal

A common thing to do with a string is start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**. A natural way to encode a traversal is with a `while` statement:

```
int index = 0;
while (index < fruit.length()) {
  char letter = fruit.charAt (index);
  System.out.println (letter);
  index = index + 1;
}
```

This loop traverses the string and prints each letter on a line by itself. Notice that the condition is `index < fruit.length()`, which means that when `index` is equal to the length of the string, the condition is false and the body of the loop is not executed. The last character we access is the one with the index `fruit.length()-1`.

The name of the loop variable is `index`. An **index** is a variable or value used to specify one member of an ordered set (in this case the set of characters in the string). The index indicates (hence the name) which one you want. The set has to be ordered so that each letter has an index and each index refers to a single character.

As an exercise, write a method that takes a `String` as an argument and that prints the letters backwards, all on one line.

7.4 Run-time errors

Way back in Section **??** 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**.

So far, 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` command and you provide an index that is negative or greater than `length-1`, you will get an exception: specifically, a `StringIndexOutOfBoundsException`. Try it and see how it looks.

If your program causes an exception, it prints an error message indicating the type of exception and where in the program it occurred. Then the program terminates.

7.5 Reading documentation

If you go to

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

and click on `charAt`, you will get the following documentation (or something like it):

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

```
Returns the character at the specified index.
An index ranges from 0 to length() - 1.
```

```
Parameters: index - the index of the character.
```

```
Returns: the character at the specified index of this string.
        The first character is at index 0.
```

```
Throws: StringIndexOutOfBoundsException if the index is out of range.
```

The first line is the method's `prototype` (see Section **??**), which indicates the name of the method, the type of the parameters, and the return type.

The next line describes what the method does. The next two lines explain the parameters and return values. In this case the explanations are a bit redundant, but the documentation is supposed to fit a standard format. The last line explains what exceptions, if any, can be caused by this method.

7.6 The `indexOf` method

In some ways, `indexOf` is the opposite of `charAt`. `charAt` takes an index and returns the character at that index. `indexOf` takes a character and finds the index where that character appears.

`charAt` fails if the index is out of range, and causes an exception. `indexOf` fails if the character does not appear in the string, and returns the value `-1`.

```
    String fruit = "banana";
    int index = fruit.indexOf('a');
```

This finds the index of the letter 'a' in the string. In this case, the letter appears three times, so it is not obvious what `indexOf` should do. According to the documentation, it returns the index of the *first* appearance.

In order to find subsequent appearances, there is an alternate version of `indexOf` (for an explanation of this kind of overloading, see Section **??**). It takes a second argument that indicates where in the string to start looking. If we invoke

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

it will start at the twoeth letter (the first n) and find the second a, which is at index 3. If the letter happens to appear at the starting index, the starting index is the answer. Thus,

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

returns 5. Based on the documentation, it is a little tricky to figure out what happens if the starting index is out of range:

> `indexOf` returns the index of the first occurrence of the character in the character sequence represented by this object that is greater than or equal to fromIndex, or -1 if the character does not occur.

One way to figure out what this means is to try out a couple of cases. Here are the results of my experiments:

- If the starting index is greater than or equal to `length()`, the result is -1, indicating that the letter does not appear at any index greater than the starting index.

- If the starting index is negative, the result is 1, indicating the first appearance of the letter at an index greater than the starting index.

If you go back and look at the documentation, you'll see that this behavior is consistent with the definition, even if it was not immediately obvious. Now that we have a better idea how `indexOf` works, we can use it as part of a program.

7.7 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**, and unrelated to **excrement**, which is a noun). When we exit the loop, `count` contains the result: the total number of a's.

As an exercise, encapsulate this code in a method named `countLetters`, and generalize it so that it accepts the string and the letter as arguments.

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

7.8   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`.

Technically, it is legal to increment a variable and use it in an expression at the same time. For example, you might see something like:

```
System.out.println (i++);
```

Looking at this, it is not clear whether the increment will take effect before or after the value is printed. Because expressions like this tend to be confusing, I would discourage you from using them. In fact, to discourage you even more, I'm not going to tell you what the result is. If you really want to know, you can try it.

Using the increment operators, we can rewrite the letter-counter:

```
int index = 0;
while (index < length) {
  if (fruit.charAt(index) == 'a') {
    count++;
  }
  index++;
}
```

It is a common error to write something like

```
index = index++;                // WRONG!!
```

Unfortunately, this is syntactically legal, so the compiler will not warn you. The effect of this statement is to leave the value of `index` unchanged. This is often a difficult bug to track down.

Remember, you can write `index = index +1;`, or you can write `index++;`, but you shouldn't mix them.

7.9   Character arithmetic

It may seem odd, but you can do arithmetic with characters! The expression `'a' + 1` yields the value `'b'`. Similarly, if you have a variable named `letter` that contains a character, then `letter - 'a'` will tell you where in the alphabet it appears (keeping in mind that 'a' is the zeroeth letter of the alphabet and 'z' is the 25th).

This sort of thing is useful for converting between the characters that contain numbers, like '0', '1' and '2', and the corresponding integers. They are not the same thing. For example, if you try this

```
char letter = '3';
int x = (int) letter;
System.out.println (x);
```

you might expect the value 3, but depending on your environment, you might get 51, which is the ASCII code that is used to represent the character '3', or you might get something else altogether. To convert '3' to the corresponding integer value you can subtract '0':

```
int x = (int)(letter - '0');
```

Technically, in both of these examples the typecast ((int)) is unnecessary, since Java will convert type char to type int automatically. I included the typecasts to emphasize the difference between the types, and because I'm a stickler about that sort of thing.

Since this conversion can be a little ugly, it is preferable to use the digit method in the Character class. For example:

```
int x = Character.digit (letter, 10);
```

converts letter to the corresponding digit, interpreting it as a base 10 number.

Another use for character arithmetic is to loop through the letters of the alphabet in order. For example, in Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings form an abecedarian series: Jack, Kack, Lack, Mack, Nack, Ouack, Pack and Quack. Here is a loop that prints these names in order:

```
char letter = 'J';
while (letter <= 'Q') {
  System.out.println (letter + "ack");
  letter++;
}
```

Notice that in addition to the arithmetic operators, we can also use the conditional operators on characters. The output of this program is:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Of course, that's not quite right because I've misspelled "Ouack" and "Quack." As an exercise, modify the program to correct this error.

7.9.1   Typecasting for experts

Here's a puzzler: normally, the statement `x++` is exactly equivalent to `x = x + 1`. Unless `x` is a `char`! In that case, `x++` is legal, but `x = x + 1` causes an error.

Try it out and see what the error message is, then see if you can figure out what is going on.

7.10   Strings are immutable

As you look over the documentation of the `String` methods, you might notice `toUpperCase` and `toLowerCase`. These methods are often a source of confusion, because it sounds like they have the effect of changing (or mutating) an existing string. Actually, neither these methods nor any others can change a string, because strings are **immutable**.

When you invoke `toUpperCase` on a `String`, you get a *new* `String` as a return value. For example:

```
String name = "Alan Turing";
String upperName = name.toUpperCase ();
```

After the second line is executed, `upperName` contains the value `"ALAN TURING"`, but `name` still contains `"Alan Turing"`.

7.11   Strings are incomparable

It is often necessary to compare strings to see if they are the same, or to see which comes first in alphabetical order. It would be nice if we could use the comparison operators, like `==` and `>`, but we can't.

In order to compare `Strings`, we have to use the `equals` and `compareTo` methods. For example:

```
String name1 = "Alan Turing";
String name2 = "Ada Lovelace";

if (name1.equals (name2)) {
  System.out.println ("The names are the same.");
}

int flag = name1.compareTo (name2);
if (flag == 0) {
  System.out.println ("The names are the same.");
} else if (flag < 0) {
  System.out.println ("name1 comes before name2.");
} else if (flag > 0) {
  System.out.println ("name2 comes before name1.");
}
```

The syntax here is a little weird. To compare two things, you have to invoke a method on one of them and pass the other as an argument.

The return value from `equals` is straightforward enough; `true` if the strings contain the same characters, and `false` otherwise.

The return value from `compareTo` is a little odd. It is the difference between the first characters in the strings that differ. If the strings are equal, it is 0. If the first string (the one on which the method is invoked) comes first in the alphabet, the difference is negative. Otherwise, the difference is positive. In this case the return value is positive 8, because the second letter of "Ada" comes before the second letter of "Alan" by 8 letters.

Using `compareTo` is often tricky, and I never remember which way is which without looking it up, but the good news is that the interface is pretty standard for comparing many types of objects, so once you get it you are all set.

Just for completeness, I should admit that it is *legal*, but very seldom *correct*, to use the `==` operator with `Strings`. But what that means will not make sense until later, so for now, don't do it.

7.12 Glossary

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 the `Graphics` object provided by the system, and `Strings`.

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

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

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

increment: Increase the value of a variable by one. The increment operator in Java is `++`.

decrement: Decrease the value of a variable by one. The decrement operator in Java is `--`.

exception: A run time error. Exceptions cause the execution of a program to terminate.

Chapter 8

Interesting objects

8.1 What's interesting?

Although `Strings` are objects, they are not very interesting objects, because

- They are immutable.

- They have no instance variables.

- You don't have to use the `new` command to create one.

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

The definitions of the `Point` and `Rectangle` classes are in the `java.awt` package, so we have to import them.

8.2 Packages

The built-in Java classes are divided into a number of **packages**, including `java.lang`, which contains almost all of the classes we have seen so far, and `java.awt`, which contains classes that pertain to the Java **Abstract Window Toolkit** (AWT), which contains classes for windows, buttons, graphics, etc.

In order to use a package, you have to **import** it, which is why the program in Section **??** starts with `import java.awt.*`. The `*` indicates that we want to import all the classes in the AWT package. If you want, you can name the classes you want to import explicitly, but there is no great advantage. The classes in `java.lang` are imported automatically, which is why most of our programs haven't required an `import` statement.

All `import` statements appear at the beginning of the program, outside the class definition.

8.3   `Point objects`

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

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

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

The first line is a conventional variable declaration: `blank` has type `Point`. The second line is kind of funny-looking; it invokes the `new` command, specifies the type of the new object, and provides arguments. It will probably not surprise you that the arguments are the coordinates of the new point, $(3, 4)$.

The result of the `new` command is a **reference** to the new point. I'll explain references more later; for now the important thing is that the variable `blank` contains a reference to the newly-created object. There is a standard way to diagram this assignment, shown in the figure.
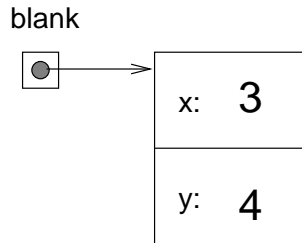
As usual, the name of the variable `blank` appears outside the box and its value appears inside the box. In this case, that value is a reference, which is shown graphically with a dot and an arrow. The arrow points to the object we're referring to.

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.

8.4   Instance variables

The pieces of data that make up an object are sometimes called components, records, or fields. In Java they 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 asked me to get something from the glove compartment of your car, you would 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."

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

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

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

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

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

8.5 Objects as parameters

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

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

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

```
java.awt.Point[x=3,y=5]
```

This is a standard format Java uses for printing objects. It prints the name of the type, followed by the contents of the object, including the names and values of the instance variables.

As a second example, we can rewrite the `distance` method from Section **??** 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; I just added them as a reminder that the instance variables in a `Point` are integers.

8.6   Rectangles

`Rectangles` are similar to points, except that they have four instance variables, named `x`, `y`, `width` and `height`. Those names should sound familiar; they are the names of the variables we used to specify bounding boxes for the Mickey Mouse fractal in Section **??**.

Other than that, everything is pretty much the same.

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

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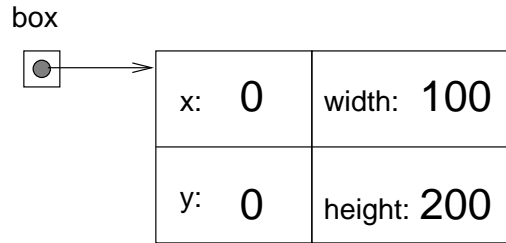
If you print `box`, you get

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

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

8.7   Objects as return types

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

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

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

8.8 Objects are mutable

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

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

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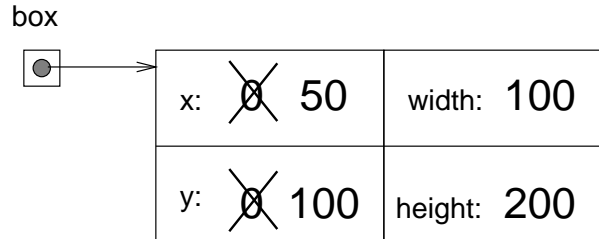
We could take this code and encapsulate it in a method, and generalize it
to move the rectangle by any amount:

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

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

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

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

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

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

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

The effect is exactly the same.

8.9   Aliasing

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

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

generates a state diagram that looks like this:
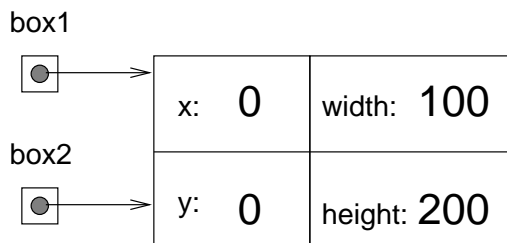
Both `box1` and `box2` refer or "point" to the same object. In other words, this object has two names, `box1` and `box2`. When a person uses two names, it's called **aliasing**. Same thing with objects.

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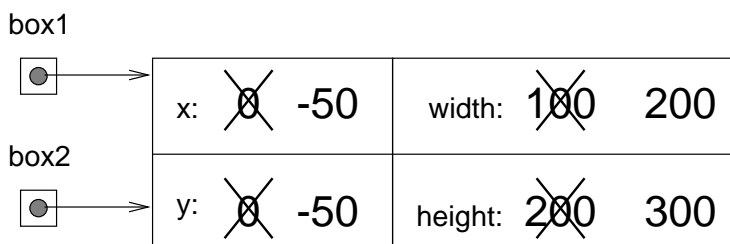
As should be clear from this figure, whatever changes are made to `box1` also apply to `box2`. Thus, the value printed by the third line is 200, the width of the expanded rectangle. (As an aside, it is perfectly legal for the coordinates of a `Rectangle` to be negative.)

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

8.10 null

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

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

```
Point blank = null;
```

and is shown in the following state diagram:

blank

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

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

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

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

8.11 Garbage collection

In Section **??** 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).
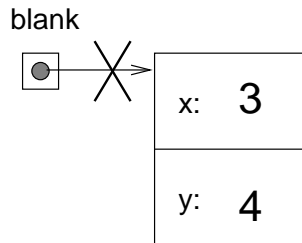
If no one refers to an object, then no one can read or write any of its values, or invoke a method on it. In effect, it ceases to exist. We could keep the object in memory, but it would only waste space, so periodically as your program runs, the Java system looks for stranded objects and reclaims them, in a process called **garbage collection**. Later, the memory space occupied by the object will be available to be used as part of a new object.

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

8.12   Objects and primitives

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

- When you declare a primitive variable, you get storage space for a primitive value. When you declare an object variable, you get a space for a reference to an object. In order to get space for the object itself, you have to use the `new` command.

- If you don't initialize a primitive type, it is given a default value that depends on the type. For example, `0` for `int`s and `true` for `boolean`s. The default value for object types is `null`, which indicates no object.

- Primitive variables are well isolated in the sense that there is nothing you can do in one method that will affect a variable in another method. Object variables can be tricky to work with because they are not as well isolated. If you pass a reference to an object as an argument, the method you invoke might modify the object, in which case you will see the effect. The same is true when you invoke a method on an object. Of course, that can be a good thing, but you have to be aware of it.

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

8.13 Glossary

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

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

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

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

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

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

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

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.

Chapter 9

Create your own objects

9.1 Class definitions and object types

Every time you write a class definition, you create a new Object type, with the same name as the class. Way back in Section **??**, when we defined the class named `Hello`, we also created an object type named `Hello`. We didn't create any variables with type `Hello`, and we didn't use the `new` command to create any `Hello` objects, but we could have!

That example may not make any sense, since there is no reason to create a `Hello` object, and it is not clear what it would be good for if we did. In this chapter, we will look at some examples of class definitions that create *useful* new 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; hence, it is an instance of some class.

- When you invoke the `new` command to create an object, Java invokes a special method called a **constructor** to initialize the instance variables. You provide one or more constructors as part of the class definition.

- Typically all the methods that operate on a type go in the class definition for that type.

Here are some syntax issues about class definitions:

- Class names (and hence object types) always begin with a capital letter, which helps distinguish them from primitive types and variable names.

- You usually put one class definition in each file, and the name of the file must be the same as the name of the class, with the suffix `.java`. For example, the `Time` class is defined in the file named `Time.java`.

- In any program, one class is designated as the **startup class**. The startup class must contain a method named `main`, which is where the execution of the program begins. Other classes *may* have a method named `main`, but they will not be executed.

With those issues out of the way, let's look at an example of a user-defined type, `Time`.

9.2   Time

A common motivation for creating a new Object type is to take several related pieces of data and encapsulate them into an object that can be manipulated (passed as an argument, operated on) as a single unit. We have already seen two built-in types like this, `Point` and `Rectangle`.

Another example, which we will implement ourselves, is `Time`, which is used to record the time of day. The various pieces of information that form a time are the hour, minute and second. Because every `Time` object will contain these data, we need to create instance variables to hold them.

The first step is to decide what type each variable should be. It seems clear that `hour` and `minute` should be integers. Just to keep things interesting, let's make `second` a `double`, so we can record fractions of a second.

Instance variables are declared at the beginning of the class definition, outside of any method definition, like this:

```
class Time {
  int hour, minute;
  double second;
}
```

All by itself, this code fragment is a legal class definition. The state diagram for a `Time` object would look like this:
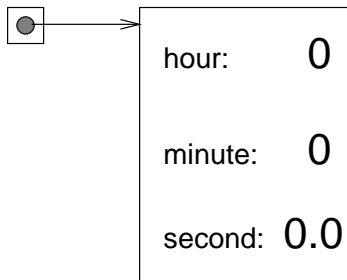
After declaring the instance variables, the next step is usually to define a constructor for the new class.

9.3 Constructors

The usual role of a constructor is to initialize the instance variables. The syntax for constructors is similar to that of other methods, with three exceptions:

- The name of the constructor is the same as the name of the class.

- Constructors have no return type and no return value.

- The keyword `static` is omitted.

Here is an example for the `Time` class:

```
public Time () {
  this.hour = 0;
  this.minute = 0;
  this.second = 0.0;
}
```

Notice that where you would expect to see a return type, between `public` and `Time`, there is nothing. That's how we (and the compiler) can tell that this is a constructor.

This constructor does not take any arguments, as indicated by the empty parentheses `()`. Each line of the constructor initializes an instance variable to an arbitrary default value (in this case, midnight). The name `this` is a special keyword that is the name of the object we are creating. You can use `this` the same way you use the name of any other object. For example, you can read and write the instance variables of `this`, and you can pass `this` as an argument to other methods.

But you do not declare `this` and you do not use `new` to create it. In fact, you are not even allowed to make an assignment to it! `this` is created by the system; all you have to do is store values in its instance variables.

A common error when writing constructors is to put a `return` statement at the end. Resist the temptation.

9.4 More constructors

Constructors can be overloaded, just like other methods, which means that you can provide multiple constructors with different parameters. Java knows which constructor to invoke by matching the arguments of the `new` command with the parameters of the constructors.

It is very common to have one constructor that takes no arguments (shown above), and one constructor that takes a parameter list that is identical to the list of instance variables. For example:

```
public Time (int hour, int minute, double second) {
  this.hour = hour;
```

```
    this.minute = minute;
    this.second = second;
  }
```

The names and types of the parameters are exactly the same as the names and types of the instance variables. All the constructor does is copy the information from the parameters to the instance variables.

If you go back and look at the documentation for `Points` and `Rectangles`, you will see that both classes provide constructors like this. Overloading constructors provides the flexibility to create an object first and then fill in the blanks, or to collect all the information before creating the object.

So far this might not seem very interesting, and in fact it is not. Writing constructors is a boring, mechanical process. Once you have written two, you will find that you can churn them out in your sleep, just by looking at the list of instance variables.

9.5   Creating a new object

Although constructors look like methods, you never invoke them directly. Instead, when you use the `new` command, the system allocates space for the new object and then invokes your constructor to initialize the instance variables.

The following program demonstrates two ways to create and initialize `Time` objects:

```
class Time {
  int hour, minute;
  double second;

  public Time () {
    this.hour = 0;
    this.minute = 0;
    this.second = 0.0;
  }

  public Time (int hour, int minute, double second) {
    this.hour = hour;
    this.minute = minute;
    this.second = second;
  }

  public static void main (String[] args) {

    // one way to create and initialize a Time object
    Time t1 = new Time ();
    t1.hour = 11;
    t1.minute = 8;
```

```
    t1.second = 3.14159;
    System.out.println (t1);

    // another way to do the same thing
    Time t2 = new Time (11, 8, 3.14159);
    System.out.println (t2);
  }
}
```

As an exercise, figure out the flow of execution through this program.

In `main`, the first time we invoke the `new` command, we provide no arguments, so Java invokes the first constructor. The next few lines assign values to each of the instance variables.

The second time we invoke the `new` command, we provide arguments that match the parameters of the second constructor. This way of initializing the instance variables is more concise (and slightly more efficient), but it can be harder to read, since it is not as clear which values are assigned to which instance variables.

9.6 Printing an object

The output of this program is:

```
Time@80cc7c0
Time@80cc807
```

When Java prints the value of a user-defined object type, it prints the name of the type and a special hexadecimal (base 16) code that is unique for each object. This code is not meaningful in itself; in fact, it can vary from machine to machine and even from run to run. But it can be useful for debugging, in case you want to keep track of individual objects.

In order to print objects in a way that is more meaningful to users (as opposed to programmers), you usually want to write a method called something like `printTime`:

```
  public static void printTime (Time t) {
    System.out.println (t.hour + ":" + t.minute + ":" + t.second);
  }
```

Compare this method to the version of `printTime` in Section **??**.

The output of this method, if we pass either `t1` or `t2` as an argument, is `11:8:3.14159`. Although this is recognizable as a time, it is not quite in the standard format. For example, if the number of minutes or seconds is less than 10, we expect a leading `0` as a place-keeper. Also, we might want to drop the decimal part of the seconds. In other words, we want something like `11:08:03`.

In most languages, there are simple ways to control the output format for numbers. In Java there are no simple ways.

Java provides very powerful tools for printing formatted things like times and dates, and also for interpreting formatted input. Unfortunately, these tools are not very easy to use, so I am going to leave them out of this book. If you want, though, you can take a look at the documentation for the `Date` class in the `java.util` package.

9.7   Operations on objects

Even though we can't print times in an optimal format, we can still write methods that manipulate `Time` objects. In the next few sections, I will demonstrate several of the possible interfaces for methods that operate on objects. For some operations, you will have a choice of several possible interfaces, so you should consider the pros and cons of each of these:

pure function: Takes objects and/or primitives as arguments but does not modify the objects. The return value is either a primitive or a new object created inside the method.

modifier: Takes objects as arguments and modifies some or all of them. Often returns void.

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.

9.8   Pure functions

A method is considered a pure function if the result depends only on the arguments, and it has no side effects like modifying an argument or printing something. The only result of invoking a pure function is the return value.

One example is `after`, which compares two `Time`s and returns a `boolean` that indicates whether the first operand comes after the second:

```
public static boolean after (Time time1, Time time2) {
  if (time1.hour > time2.hour) return true;
  if (time1.hour < time2.hour) return false;

  if (time1.minute > time2.minute) return true;
  if (time1.minute < time2.minute) return false;

  if (time1.second > time2.second) return true;
  return false;
}
```

What is the result of this method if the two times are equal? Does that seem like the appropriate result for this method? If you were writing the documentation for this method, would you mention that case specifically?

A second example is `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:

```
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;
  return sum;
}
```

Although this method returns a `Time` object, it is not a constructor. You should go back and compare the syntax of a method like this with the syntax of a constructor, because it is easy to get confused.

Here is an example of how to use this method. If `currentTime` contains the current time and `breadTime` contains the amount of time it takes for your breadmaker to make bread, then you could use `addTime` to figure out when the bread will be done.

```
  Time currentTime = new Time (9, 14, 30.0);
  Time breadTime = new Time (3, 35, 0.0);
  Time doneTime = addTime (currentTime, breadTime);
  printTime (doneTime);
```

The output of this program is 12:49:30.0, which is correct. On the other hand, there are cases where the result is not correct. Can you think of one?

The problem is that this method does not deal with cases where the number of seconds or minutes adds up to more than 60. In that case, we have to "carry" the extra seconds into the minutes column, or extra minutes into the hours column.

Here's a second, corrected version of this method.

```
public 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 will suggest an alternate approach to this problem that will be much shorter.

This code demonstrates two operators we have not seen before, `+=` and `-=`. These operators provide a concise way to increment and decrement variables. They are similar to `++` and `--`, except (1) they work on `doubles` as well as `ints`, and (2) the amount of the increment does not have to be 1. The statement `sum.second -= 60.0;` is equivalent to `sum.second = sum.second - 60;`

9.9   Modifiers

As an example of a modifier, consider `increment`, which adds a given number of seconds to a `Time` object. Again, a rough draft of this method looks like:

```
  public 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 simply 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?

9.10 Fill-in methods

Occasionally you will see methods like `addTime` written with a different interface (different arguments and return values). Instead of creating a new object every time `addTime` is invoked, we could require the caller to provide an "empty" object where `addTime` should store the result. Compare the following with 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;
    }
  }
```

One advantage of this approach is that the caller has the option of reusing the same object repeatedly to perform a series of additions. This can be slightly more efficient, although it can be confusing enough to cause subtle errors. For the vast majority of programming, it is worth spending a little run time to avoid a lot of debugging time.

9.11 Which is best?

Anything that can be done with modifiers and fill-in methods can also be done with pure functions. In fact, there are programming languages, called **functional** programming languages, that only allow pure functions. Some programmers believe that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers. Nevertheless, there are times when modifiers are convenient, and some cases where functional programs are less efficient.

In general, I recommend that you write pure functions whenever it is reasonable to do so, and resort to modifiers only if there is a compelling advantage. This approach might be called a functional programming style.

9.12   Incremental development vs. planning

In this chapter I have demonstrated an approach to program development I refer to as **rapid prototyping with iterative improvement**. In each case, I wrote a rough draft (or prototype) that performed the basic calculation, and then tested it on a few cases, correcting flaws as I found them.

Although this approach can be effective, it can lead to code that is unnecessarily complicated—since it deals with many special cases—and unreliable—since it is hard to convince yourself that you have found *all* the errors.

An alternative is high-level planning, in which a little insight into the problem can make the programming much easier. In this case the insight is that a Time is really a three-digit number in base 60! The second is the "ones column," the minute is the "60's column", and the hour is the "3600's column."

When we wrote addTime and increment, we were effectively doing addition in base 60, which is why we had to "carry" from one column to the next.

Thus an alternate approach to the whole problem is to convert Times into doubles and take advantage of the fact that the computer already knows how to do arithmetic with doubles. 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 third constructor:

```
public Time (double secs) {
   this.hour = (int) (secs / 3600.0);
   secs -= this.hour * 3600.0;
   this.minute = (int) (secs / 60.0);
   secs -= this.minute * 60;
   this.second = secs;
}
```

This constructor is a little different from the others, since it involves some calculation along with assignments to the instance variables.

You might have to think a bit to convince yourself that the technique I am using to convert from one base to another is correct. Assuming you are convinced, we can use these methods to rewrite addTime:

```
public static Time addTime (Time t1, Time t2) {
  double seconds = convertToSeconds (t1) + convertToSeconds (t2);
  return new Time (seconds);
}
```

This is much shorter than the original version, and it is much easier to demonstrate that it is correct (assuming, as usual, that the methods it invokes are correct). As an exercise, rewrite `increment` the same way.

9.13 Generalization

In some ways converting from base 60 to base 10 and back is harder than just dealing with times. Base conversion is more abstract; our intuition for dealing with times is better.

But if we have the insight to treat times as base 60 numbers, and make the investment of writing the conversion methods (`convertToSeconds` and the third constructor), we get a program that is shorter, easier to read and debug, and more reliable.

It is also easier to add more features later. For example, imagine subtracting two `Times` to find the duration between them. The naive approach would be to implement subtraction complete with "borrowing." Using the conversion methods would be much easier.

Ironically, sometimes making a problem harder (more general) makes is easier (fewer special cases, fewer opportunities for error).

9.14 Algorithms

When you write a general solution for a class of problems, as opposed to a specific solution to a single problem, you have written an **algorithm**. I mentioned this word in Chapter 1, but did not define it carefully. It is not easy to define, so I will try a couple of approaches.

First, consider some things that are not algorithms. For example, when you learned to multiply single-digit numbers, you probably memorized the multiplication table. In effect, you memorized 100 specific solutions, so that knowledge is not really algorithmic.

But if you were "lazy," you probably cheated by learning a few tricks. For example, to find the product of n and 9, you can write $n - 1$ as the first digit and $10 - n$ as the second digit. This trick a general solution for multiplying any single-digit number by 9. That's an algorithm!

Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms. One of the characteristics of algorithms is that they do not require any intelligence to carry out. They are mechanical processes in which each step follows from the last according to a simple set of rules.

In my opinion, it is embarrassing that humans spend so much time in school learning to execute algorithms that, quite literally, require no intelligence.

On the other hand, the process of designing algorithms is interesting, intellectually challenging, and a central part of what we call programming.

Some of the things that people do naturally, without difficulty or conscious thought, are the most difficult to express algorithmically. Understanding natural language is a good example. We all do it, but so far no one has been able to explain *how* we do it, at least not in the form of an algorithm.

Later you will have the opportunity to design simple algorithms for a variety of problems.

9.15   Glossary

class: Previously, I defined a class as a collection of related methods. In this chapter we learned that a class definition is also a template for a new type of object.

instance: A member of a class. Every object is an instance of some class.

constructor: A special method that initializes the instance variables of a newly-constructed object.

project: A collection of one or more class definitions (one per file) that make up a program.

startup class: The class that contains the `main` method where execution of the program begins.

function: A method whose result depends only on its parameters, and that has no side-effects other than returning a value.

functional programming style: A style of program design in which the majority of methods are functions.

modifier: A method that changes one or more of the objects it receives as parameters, and usually returns `void`.

fill-in method: A type of method that takes an "empty" object as a parameter and fills it its instance variables instead of generating a return value. This type of method is usually not the best choice.

algorithm: A set of instructions for solving a class of problems by a mechanical, unintelligent process.

Chapter 10

Arrays

An **array** is a set of values where each value is identified by an index. You can make an array of `ints`, `doubles`, or any other type, but all the values in an array have to have the same type.

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

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

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

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

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

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

The following figure shows how arrays are represented in state diagrams:
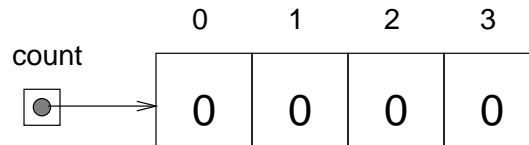
The large numbers inside the boxes are the **elements** of the array. The small numbers outside the boxes are the indices used to identify each box. When you allocate a new array, the elements are initialized to zero.

10.1   Accessing elements

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

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

All of these are legal assignment statements. Here is the effect of this code fragment:
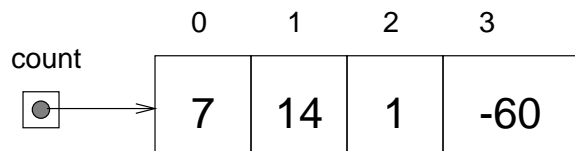
By now you should have noticed that the four elements of this array are numbered from 0 to 3, which means that there is no element with the index 4. This should sound familiar, since we saw the same thing with `String` indices. Nevertheless, it is a common error to go beyond the bounds of an array, which will cause an `ArrayOutOfBoundsException`. As with all exceptions, you get an error message and the program quits.

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

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

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

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

10.2   Copying arrays

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

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

This code creates one array of three `doubles`, and sets two different variables to refer to it. This situation is a form of aliasing.
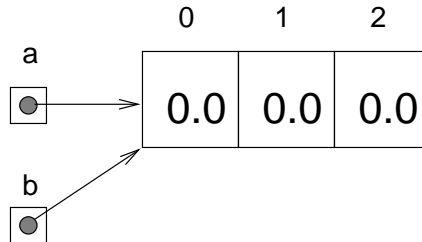
Any changes in either array will be reflected in the other. This is not usually the behavior you want; instead, you should make a copy of the array, by allocating a new array and copying each element from one to the other.

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

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

10.3 `for` loops

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

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

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

This statement is exactly equivalent to

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

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

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

is equivalent to

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

As an exercise, write a `for` loop to copy the elements of an array.

10.4   Arrays and objects

In many ways, arrays behave like objects:

- When you declare an array variable, you get a reference to an array.

- You have to use the `new` command to create the array itself.

- When you pass an array as an argument, you pass a reference, which
  means that the invoked method can change the contents of the array.

Some of the objects we have looked at, like `Rectangles`, are similar to arrays,
in the sense that they are named collection of values. This raises the question,
"How is an array of 4 integers different from a Rectangle object?"

If you go back to the definition of "array" at the beginning of the chapter,
you will see one difference, which is that the elements of an array are identified
by indices, whereas the elements (instance variables) of an object have names
(like `x`, `width`, etc.).

Another difference between arrays and objects is that all the elements of an
array have to be the same type. Although that is also true of `Rectangles`, we
have seen other objects that have instance variables with different types (like
`Time`).

10.5   Array length

Actually, arrays do have one named instance variable: `length`. Not surprisingly,
it contains the length of the array (number of elements). It is a good idea to
use this value as the upper bound of a loop, rather than a constant value. That
way, if the size of the array changes, you won't have to go through the program
changing all the loops; they will work correctly for any size array.

```
for (int i = 0; i < a.length; i++) {
  b[i] = a[i];
}
```

The last time the body of the loop gets executed, `i` is `a.length - 1`, which is the index of the last element. When `i` is equal to `a.length`, the condition fails and the body is not executed, which is a good thing, since it would cause an exception. This code assumes that the array `b` contains at least as many elements as `a`.

As an exercise, write a method called `cloneArray` that takes an array of integers as a parameter, creates a new array that is the same size, copies the elements from the first array into the new one, and then returns a reference to the new array.

10.6 Random numbers

Most computer programs do the same thing every time they are executed, so they are said to be **deterministic**. Usually, determinism is a good thing, since we expect the same calculation to yield the same result. For some applications, though, we would like the computer to be unpredictable. Games are an obvious example, but there are many more.

Making a program truly **nondeterministic** turns out to be not so easy, but there are ways to make it at least seem nondeterministic. One of them is to generate random numbers and use them to determine the outcome of the program. Java provides a built-in method that generates **pseudorandom** numbers, which are not truly random in the mathematical sense, but for our purposes, they will do.

Check out the documentation of the `random` method in the `Math` class. The return value is a `double` between 0.0 and 1.0. Each time you invoke `random` you get a different randomly-generated number. To see a sample, run this loop:

```
for (int i = 0; i < 10; i++) {
  double x = Math.random ();
  System.out.println (x);
}
```

To generate a random `double` between 0.0 and an upper bound like `high`, you can multiply `x` by `high`. How would you generate a random number between `low` and `high`? How would you generate a random integer?

10.7 Statistics

The numbers generated by `random` are supposed to be distributed uniformly. If you have taken statistics, you know what that means. Among other things, it means that if we divide the range of possible values into equal sized "buckets,"

and count the number of times a random value falls in each bucket, each bucket should get the same number of hits (eventually).

In the next few sections, we will write programs that generate a sequence of random numbers and check whether this property holds true.

10.8   Array of random numbers

The first step is to generate a large number of random values and store them in an array. By "large number," of course, I mean 8. It's always a good idea to start with a manageable number, to help with debugging, and then increase it later.

The following method takes a single argument, the size of the array. It allocates a new array of `doubles`, fills it with random values, and returns a reference to the new array.

```
public static double[] randomArray (int n) {
  double[] a = new double[n];
  for (int i = 0; i<a.length; i++) {
    a[i] = Math.random ();
  }
  return a;
}
```

The return type is `double[]`, which means that this method returns an array of `doubles`. To test this method, it is convenient to have a method that prints the contents of an array.

```
public static void printArray (double[] 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;
double[] array = randomArray (numValues);
printArray (array);
```

On my machine the output is

```
0.7344558779885422
0.6224282219647016
0.09591424515329172
0.2992298398883563
0.7736458103088713
0.7069110192991597
```

```
0.7042440765950522
0.977839532249852
```

which is pretty random-looking. Your results may differ.

If these numbers are really random, we expect half of them to be greater than 0.5 and half to be less. In fact, six are greater than 0.5, so that's a little high.

If we divide the range into four buckets—from 0.0 to 0.25, 0.25 to 0.5, 0.5 to 0.75, and 0.75 to 1.0—we expect 2 values to fall in each bucket. In fact, we get 1, 1, 4, 2. Again, not exactly what we expected.

Do these results mean the values are not really random? It's hard to tell. With so few values, the chances are slim that we would get exactly what we expect. But as the number of values increases, the outcome should be more predictable.

To test this theory, we'll write some programs that divide the range into buckets and count the number of values in each.

10.9 Counting

A good approach to problems like this is to think of simple methods that are easy to write, and that might turn out to be useful. Then you can combine them into a solution. Of course, it is not easy to know ahead of time which methods are likely to be useful, but as you gain experience you will have a better idea.

Also, it is not always obvious what sort of things are easy to write, but a good approach is to look for subproblems that fit a pattern you have seen before.

Back in Section **??** we looked at 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, I have a method in mind called `inBucket` that counts the number of elements in an array that fall in a given bucket. The parameters are the array and two `doubles` that specify the lower and upper bounds of the bucket.

```java
public static int inBucket (double[] a, double low, double high) {
  int count = 0;
  for (int i=0; i<a.length; i++) {
    if (a[i] >= low && a[i] < high) count++;
  }
  return count;
}
```

I haven't been very careful about whether something equal to `low` or `high` falls in the bucket, but you can see from the code that `low` is in and `high` is out. That should prevent me from counting any elements twice.

Now, to divide the range into two pieces, we could write

```
int low = inBucket (a, 0.0, 0.5);
int high = inBucket (a, 0.5, 1.0);
```

To divide it into four pieces:

```
int bucket1 = inBucket (a, 0.0, 0.25);
int bucket2 = inBucket (a, 0.25, 0.5);
int bucket3 = inBucket (a, 0.5, 0.75);
int bucket4 = inBucket (a, 0.75, 1.0);
```

You might want to try out this program using a larger `numValues`. As `numValues` increases, are the numbers in each bucket levelling off?

10.10   Many buckets

Of course, as the number of buckets increases, we don't want to have to rewrite the program, especially since the code is getting big and repetitive. Any time you find yourself doing something more than a few times, you should be looking for a way to automate it.

Let's say that we wanted 8 buckets. The width of each bucket would be one eighth of the range, which is 0.125. To count the number of values in each bucket, we need to be able to generate the bounds of each bucket automatically, and we need to have some place to store the 8 counts.

We can solve the first problem with a loop:

```
int numBuckets = 8;
double bucketWidth = 1.0 / numBuckets;

for (int i = 0; i < numBuckets; i++) {
  double low = i * bucketWidth;
  double high = low + bucketWidth;
  System.out.println (low + " to " + high);
}
```

This code uses the loop variable `i` to multiply by the bucket width, in order to find the low end of each bucket. The output of this loop is:

```
0.0 to 0.125
0.125 to 0.25
0.25 to 0.375
0.375 to 0.5
0.5 to 0.625
```

```
0.625 to 0.75
0.75 to 0.875
0.875 to 1.0
```

You can confirm that each bucket is the same width, that they don't overlap, and that they cover the whole range from 0.0 to 1.0.

Now we just need a way to store 8 integers, preferably so we can use an index to access each one. Immediately, you should be thinking "array!"

What we want is an array of 8 integers, which we can allocate outside the loop; then, inside the loop, we'll invoke `inBucket` and store the result:

```
int numBuckets = 8;
int[] buckets = new int [8];
double bucketWidth = 1.0 / numBuckets;

for (int i = 0; i<numBuckets; i++) {
  double low = i * bucketWidth;
  double high = low + bucketWidth;
  //System.out.println (low + " to " + high);

  buckets[i] = inBucket (a, low, high);
}
```

The tricky thing here is that I am using the loop variable as an index into the `buckets` array, in addition to using it to compute the range of each bucket.

This code works. I cranked the number of values up to 1000 and divided the range into 8 buckets. The output is:

```
129
109
142
118
131
124
121
126
```

which is pretty close to 125 in each bucket. At least, it's close enough that I can believe the random number generator is working.

10.11 A single-pass solution

Although this code works, it is not as efficient as it could be. Every time it invokes `inBucket`, it traverses the entire array. As the number of buckets 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 bucket it falls in. Then we could increment the appropriate counter.

In the previous section, we took an index, `i`, and multiplied it by the `bucketWidth` in order to find the lower bound of a given bucket. Now we want to take a value in the range 0.0 to 1.0, and find the index of the bucket where it falls.

Since this problem is the inverse of the previous problem we might guess that we should *divide* by the `bucketwidth` instead of multiplying. That guess is correct.

Remember that since `bucketWidth = 1.0 / numBuckets`, dividing by `bucketWidth` is the same as multiplying by `numBuckets`. If we take a number in the range 0.0 to 1.0 and multiply by `numBuckets`, we get a number in the range from 0.0 to `numBuckets`. If we round that number to the next lower integer, we get exactly what we are looking for—the index of the appropriate bucket.

```
int numBuckets = 8;
int[] buckets = new int [8];

for (int i = 0; i < numValues; i++) {
  int index = (int) (a[i] * numBuckets);
  buckets[index]++;
}
```

Here I am using a typecast to round the value down to the next integer and convert it to type `int` at the same time.

Is it possible for this calculation to produce an index that is out of range (either negative or greater than `a.length-1`)? If so, how would you fix it?

An array like `buckets`, that contains counts of the number of values in each range, is called a **histogram**. As an exercise, write a method called `histogram` that takes an array and a number of buckets as parameters, and that returns a histogram with the given number of buckets.

10.12   Glossary

array: A named collection of values, where all the values have the same type, and each value is identified by an index.

collection: Any data structure that contains a set of items or elements.

element: One of the values in an array. The [] operator selects elements of an array.

index: An integer variable or value used to indicate an element of an array.

deterministic: A program that does the same thing every time it is invoked.

pseudorandom: A sequence of numbers that appear to be random, but which are actually the product of a deterministic computation.

histogram: An array of integers where each integer counts the number of values that fall into a certain range.

Chapter 11

Arrays of Objects

11.1 Composition

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 have arrays of objects. In fact, you can also have objects that contain arrays (as instance variables); you can have arrays that contain arrays; you can have objects that contain objects, and so on.

In the next two chapters we will look at some examples of these combinations, using `Card` objects as an example.

11.2 `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. There are 52 cards in a deck, each of which 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 rank of the Ace may be 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 `Strings`, 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 symbol $\mapsto$ is mathematical notation for "maps to." 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 Java 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 am providing two constructors, one of which takes a parameter for each instance variable and the other of which takes no parameters.

To create an object that represents the 3 of Clubs, we would use the `new` command:

```
    Card threeOfClubs = new Card (0, 3);
```

The first argument, 0 represents the suit Clubs.

11.3 The `printCard` method

When you create a new class, the first step is usually to declare the instance variables and write constructors. The second step is often to write the standard methods that every object should have, including one that prints the object, and one or two that compare objects. I will start with `printCard`.

In order 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 `Strings`. You can create an array of `Strings` 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" };
```

The effect of this statement is identical to that of the separate declaration, allocation, and assignment. A state diagram of this array might look like:
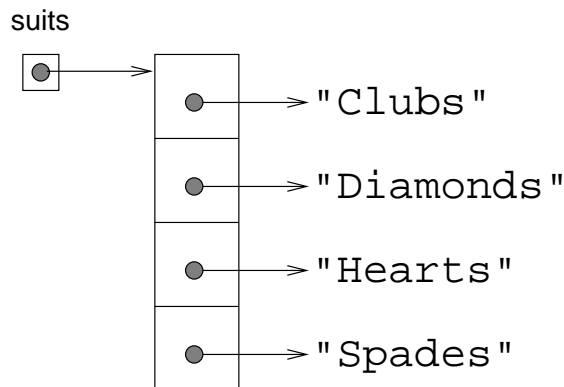
The elements of the array are *references* to the `Strings`, rather than `Strings` themselves. This is true of all arrays of objects, as I will discuss in more detail later. For now, all we need is another array of `Strings` to decode the ranks:

```
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 will never be used. The only valid ranks are 1–13. This wasted entry is not necessary, of course. We could have started at 0, as usual, but it is best to encode 2 as 2, and 3 as 3, etc.

Using these arrays, we can select the appropriate `Strings` by using the `suit` and `rank` as indices. In the method `printCard`,

```
public static void printCard (Card c) {
   String[] suits = { "Clubs", "Diamonds", "Hearts", "Spades" };
   String[] ranks = { "narf", "Ace", "2", "3", "4", "5", "6",
"7", "8", "9", "10", "Jack", "Queen", "King" };

   System.out.println (ranks[c.rank] + " of " + suits[c.suit]);
}
```

the expression `suits[c.suit]` means "use the instance variable `suit` from the object `c` as an index into the array named `suits`, and select the appropriate string." The output of this code

```
   Card card = new Card (1, 11);
   printCard (card);
```

is `Jack of Diamonds`.

11.4   The `sameCard` method

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 and the same. 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 can use the `==` operator. For example:

```
   Card card1 = new Card (1, 11);
   Card card2 = card1;

   if (card1 == card2) {
     System.out.println ("card1 and card2 are the same object.");
   }
```

This type of equality is called **shallow equality** because it only compares the references, not the contents of the objects.

To compare the contents of the objects—**deep equality**—it is common to write a method with a name like `sameCard`.

```
public static boolean sameCard (Card c1, Card c2) {
   return (c1.suit == c2.suit && c1.rank == c2.rank);
}
```

Now if we create two different objects that contain the same data, we can use `sameCard` to see if they represent the same card:

```
Card card1 = new Card (1, 11);
Card card2 = new Card (1, 11);

if (sameCard (card1, card2)) {
   System.out.println ("card1 and card2 are the same card.");
}
```

In this case, `card1` and `card2` are two different objects that contain the same data
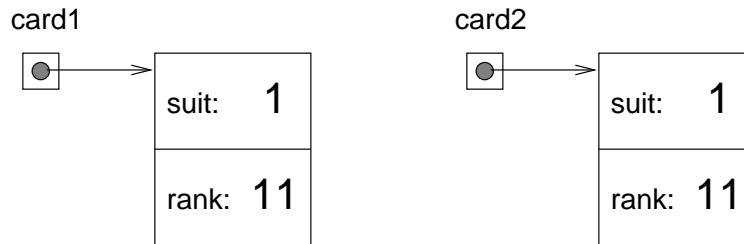
so the condition is true. What does the state diagram look like when `card1 == card2` is true?

In Section **??** I said that you should never use the `==` operator on `Strings` because it does not do what you expect. Instead of comparing the contents of the `String` (deep equality), it checks whether the two `Strings` are the same object (shallow equality).

11.5   The `compareCard` method

For primitive types, there are conditional operators that 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 `Strings` there is a built-in `compare` method. For `Cards` we have to write our own, which we will call `compareCard`. Later, we will use this method to sort a deck of cards.

Some sets are completely ordered, which means that you can compare any two elements and tell which is bigger. For example, the integers and the 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. For example, the 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.

In order to make cards comparable, we have to decide which is more important, rank or suit. To be honest, the choice is completely arbitrary. For the sake of choosing, I will say that suit is more important, because when you buy a new deck of cards, it comes sorted with all the Clubs together, followed by all the Diamonds, and so on.

With that decided, we can write `compareCard`. It will take two `Cards` as parameters and return 1 if the first card wins, -1 if the second card wins, and 0 if they tie (indicating deep equality). It is sometimes confusing to keep those return values straight, but they are pretty standard for comparison methods.

First we compare the suits:

```
if (c1.suit > c2.suit) return 1;
if (c1.suit < c2.suit) return -1;
```

If neither statement is true, then the suits must be equal, and we have to compare ranks:

```
if (c1.rank > c2.rank) return 1;
if (c1.rank < c2.rank) return -1;
```

If neither of these is true, the ranks must be equal, so we return 0. In this ordering, aces will appear lower than deuces (2s).

As an exercise, fix it so that aces are ranked higher than Kings, and encapsulate this code in a method.

11.6   Arrays of cards

The reason I chose `Cards` as the objects for this chapter is that there is an obvious use for an array of cards—a deck. Here is some code that creates a new deck of 52 cards:

```
Card[] deck = new Card [52];
```

Here is the state diagram for this object:
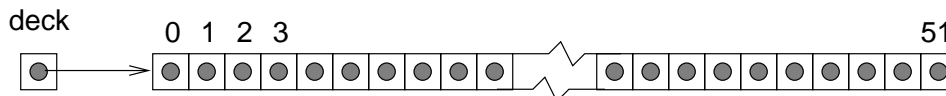
The important thing to see here is that the array contains only *references* to objects; it does not contain any `Card` objects. The values of the array elements are initialized to `null`. You can access the elements of the array in the usual way:

```
if (deck[3] == null) {
    System.out.println ("No cards yet!");
}
```

But if you try to access the instance variables of the non-existent `Cards`, you will get a `NullPointerException`.

```
deck[2].rank;                // NullPointerException
```

Nevertheless, that is the correct syntax for accessing the `rank` of the "twoeth" card in the deck (really the third—we started at zero, remember?). This is another example of composition, the combination of 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 a nested loop:

```
int index = 0;
for (int suit = 0; suit <= 3; suit++) {
    for (int rank = 1; rank <= 13; rank++) {
        deck[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 iterates 4 times, and the inner loop iterates 13 times, the total number of times the body is executed is 52 (13 times 4).

I used the variable `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:
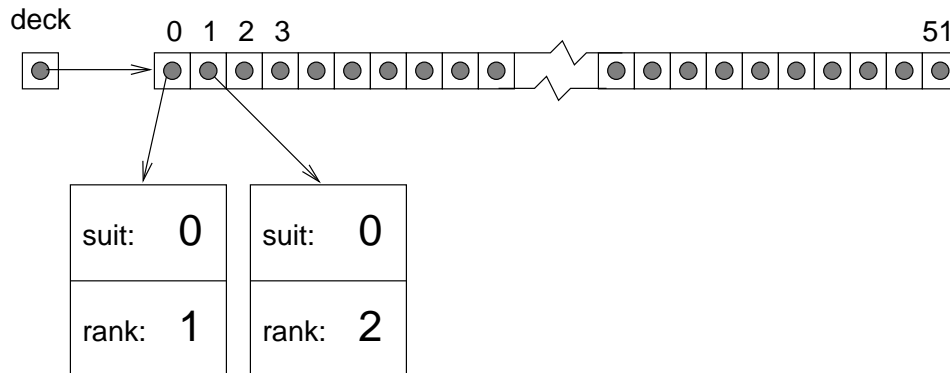
As an exercise, encapsulate this deck-building code in a method called buildDeck that takes no parameters and that returns a fully-populated array of Cards.

11.7   The printDeck method

Whenever you are working with arrays, it is convenient to have a method that will print the contents of the array. We have seen the pattern for traversing an array several times, so the following method should be familiar:

```
public static void printDeck (Card[] deck) {
  for (int i=0; i<deck.length; i++) {
    printCard (deck[i]);
  }
}
```

Since deck has type Card[], an element of deck has type Card. Therefore, deck[i] is a legal argument for printCard.

11.8   Searching

The next method I want to write is findCard, which searches through an array of Cards to see whether it contains a certain card. It may not be obvious why this method would be useful, but it gives me a chance to demonstrate two ways to go searching for things, a linear search and a bisection search.

Linear search is the more obvious of the two; it involves traversing the deck and comparing 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 static int findCard (Card[] deck, Card card) {
  for (int i = 0; i< deck.length; i++) {
    if (sameCard (deck[i], card)) return i;
  }
  return -1;
}
```

The arguments of findCard are named card and deck. It might seem odd to have a variable with the same name as a type (the card variable has type Card). This is legal and common, although it can sometimes make code hard to read. In this case, though, I think it works.

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 the loop terminates without finding the card, we know the card is not in the deck and return -1.

If the cards in the deck are not in order, there is no way to search that is faster than this. We have to look at every card, since otherwise there is no way to 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. The reason is that the words are in alphabetical order. As a result, you probably use an algorithm that is similar to a bisection 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 found 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. The only alternative is that your word has been misfiled somewhere, but that contradicts our assumption that the words are in alphabetical order.

In the case of a deck of cards, if we know that the cards are in order, we can write a version of `findCard` that is much faster. The best way to write a bisection search is with a recursive method. That's because bisection is naturally recursive.

The trick is to write a method called `findBisect` 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 in the range from `low` to `mid-1`.

4. If the card at `mid` is lower than your card, search in the range from `mid+1` to `high`.

Steps 3 and 4 look suspiciously like recursive invocations. Here's what this all looks like translated into Java code:

```
public static int findBisect (Card[] deck, Card card, int low, int high) {
  int mid = (high + low) / 2;
  int comp = compareCard (deck[mid], card);

  if (comp == 0) {
    return mid;
  } else if (comp > 0) {
    return findBisect (deck, card, low, mid-1);
  } else {
    return findBisect (deck, card, mid+1, high);
  }
}
```

Rather than call `compareCard` three times, I called it once and stored the result.

Although this code contains the kernel of a bisection search, it is still missing a piece. As it is currently written, if the card is not in the deck, it will recurse forever. We need a way to detect this condition and deal with it properly (by returning -1).

The easiest way to tell that your card is not in the deck is if there are *no* cards in the deck, which is the case if `high` is less than `low`. Well, there are still cards in the deck, of course, but what I mean is that there are no cards in the segment of the deck indicated by `low` and `high`.

With that line added, the method works correctly:

```
public static int findBisect
            (Card[] deck, Card card, int low, int high) {
  System.out.println (low + ", " + high);

  if (high < low) return -1;

  int mid = (high + low) / 2;
  int comp = deck[mid].compareCard (card);

  if (comp == 0) {
    return mid;
  } else if (comp > 0) {
    return findBisect (deck, card, low, mid-1);
  } else {
    return findBisect (deck, card, mid+1, high);
  }
}
```

I added a print statement at the beginning so I could watch the sequence of recursive calls and convince myself that it would eventually reach the base case. I tried out the following code:

```
Card card1 = new Card (1, 11);
System.out.println (findBisect (deck, 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 calls is fairly small, typically 6 or 7. That means we only had to invoke `compareCard` 6 or 7 times, compared to up to 52 times if we did a linear search. In general, bisection is much faster than a linear search, especially for large arrays.

Two common errors in recusive programs are forgetting to include a base case and writing the recursive call so that the base case is never reached. Either error will cause an infinite recursion, in which case Java will (eventually) throw a `StackOverflowException`.

11.9 Decks and subdecks

Looking at the interface to `findBisect`

```
public static int findBisect
    (Card[] deck, Card card, int low, int high)
```

it might make sense to treat three of the parameters, `deck`, `low` and `high`, as a single parameter that specifies a **subdeck**. We took a similar view in Section **??** when we were talking about bounding boxes. In that case I referred to `x`, `y`, `width` and `height` as if they were a single parameter, a bounding box.

This kind of thing is quite common, and I sometimes think of it 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 a method and pass an array and 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 sending a subset of the deck; you are really sending the whole deck. But as long as the recipient plays by the rules, it makes sense to think of it, abstractly, as a subdeck.

There is one other example of this kind of abstraction that you might have noticed in Section **??**, when I referred to an "empty" data structure. The reason I put "empty" in quotation marks was to suggest that it is not literally accurate. All variables have values all the time. When you create them, they are given default values. So there is no such thing as an empty object.

But if the program guarantees that the current value of a variable is never read before it is written, then the current value is irrelevant. Abstractly, it makes sense to think of such a variable as "empty."

This kind of thinking, in which a program comes to take on meaning beyond what is literally encoded, is a very important part of thinking like a computer scientist. Sometimes, 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 (as well as many other fields).

A more general definition of "abstraction" is "The process of modeling a complex system with a simplified description in order to suppress unnecessary details while capturing relevant behavior."

11.10   Glossary

encode: To represent one set of values using another set of values, by constructing a mapping between them.

shallow equality: Equality of references. Two references that point to the same object.

deep equality: Equality of values. Two references that point to objects that have the same value.

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.

Chapter 12

Objects of Arrays

In the previous chapter, we worked with an array of objects, but I also mentioned that it is possible to have an object that contains an array as an instance variable. In this chapter I am going to create a new object, called a `Deck`, that contains an array of `Cards` as an instance variable.

The class definition looks like this

```
class Deck {
  Card[] cards;

  public Deck (int n) {
    cards = new Card[n];
  }
}
```

The name of the instance variable is `cards` to help distinguish the `Deck` object from the array of `Cards` that it contains. Here is a state diagram showing what a `Deck` object looks like with no cards allocated:
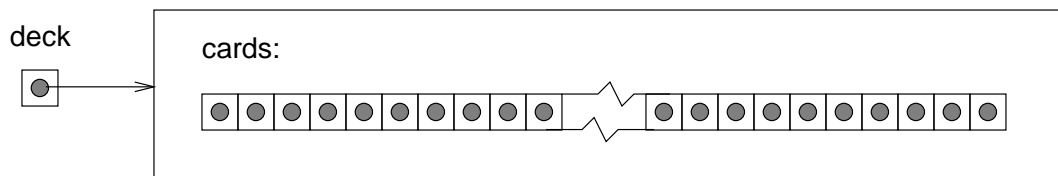
As usual, the constructor initializes the instance variable, but in this case it uses the `new` command to create the array of cards. It doesn't create any cards to go in it, though. For that we could write another constructor that creates a standard 52-card deck and populates it with `Card` objects:

```
  public Deck () {
    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++;
      }
    }
  }
```

Notice how similar this method is to `buildDeck`, except that we had to change the syntax to make it a constructor. To invoke it, we use the `new` command:

```
    Deck deck = new Deck ();
```

Now that we have a `Deck` class, it makes sense to put all 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 `printDeck` (Section **??**). Here's how it looks, rewritten to work with a `Deck` object:

```
  public static void printDeck (Deck deck) {
    for (int i=0; i<deck.cards.length; i++) {
      Card.printCard (deck.cards[i]);
    }
  }
```

The most obvious thing we have to change is the type of the parameter, from `Card[]` to `Deck`. The second change is that we can no longer use `deck.length` to get the length of the array, because `deck` is a `Deck` object now, not an array. It contains an array, but it is not, itself, an array. Therefore, we have to write `deck.cards.length` to extract the array from the `Deck` object and get the length of the array.

For the same reason, we have to use `deck.cards[i]` to access an element of the array, rather than just `deck[i]`. The last change is that the invocation of `printCard` has to say explicitly that `printCard` is defined in the `Card` class.

For some of the other methods, it is not obvious whether they should be included in the `Card` class or the `Deck` class. For example, `findCard` takes a `Card` and a `Deck` as arguments; you could reasonably put it in either class. As an exercise, move `findCard` into the `Deck` class and rewrite it so that the first parameter is a `Deck` object rather than an array of `Card`s.

12.1   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 **??** 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 reassembling the deck by 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 same order you started in. For a discussion of that claim, see `http://www.wiskit.com/marilyn/craig.html` or do a web search with the keywords "perfect shuffle."

A better shuffling algorithm is to traverse the deck one card at a time, and at each iteration choose two cards 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<deck.cards.length; i++) {
  // choose a random number between i and deck.cards.length
  // swap the ith card and the randomly-chosen card
}
```

The nice thing about using pseudocode is that it often makes it clear what methods you are going to need. In this case, we need something like `randomInt`, which chooses a random integer between the parameters `low` and `high`, and `swapCards` which takes two indices and switches the cards at the indicated positions.

You can probably figure out how to write `randomInt` by looking at Section **??**, although you will have to be careful about possibly generating indices that are out of range.

You can also figure out `swapCards` yourself. The only tricky thing is to decide whether to swap just the references to the cards or the contents of the cards. Does it matter which one you choose? Which is faster?

I will leave the remaining implementation of these methods as an exercise to the reader.

12.2 Sorting

Now that we have messed up the deck, we need a way to put it back in order. Ironically, there is an algorithm for sorting that is very similar to the algorithm for shuffling. This algorithm is sometimes called **selection sort** because it works by traversing the array repeatedly and selecting the lowest remaining card each time.

During the first iteration we find the lowest card and swap it with the card in the 0th position. During the `i`th, we find the lowest card to the right of `i` and swap it with the `i`th card.

Here is pseudocode for selection sort:

```
for (int i=0; i<deck.cards.length; i++) {
  // find the lowest card at or to the right of i
```

```
        // swap the ith card and the lowest card
    }
```

Again, the pseudocode helps with the design of the **helper methods**. In this case we can use `swapCards` again, so we only need one new one, called `findLowestCard`, that takes an array of cards and an index where it should start looking.

Once again, I am going to leave the implementation up to the reader.

12.3   Subdecks

How should we represent a hand or some other subset of a full deck? One good choice is to make a `Deck` object that has fewer than 52 cards.

We might want a method, `subdeck`, that takes an array of cards and a range of indices, and that returns a new array of cards that contains the specified subset of the deck:

```
public static Deck subdeck (Deck deck, int low, int high) {
  Deck sub = new Deck (high-low+1);

  for (int i = 0; i<sub.cards.length; i++) {
    sub.cards[i] = deck.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 the `new` command, the contructor that gets invoked will be the first one, 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.
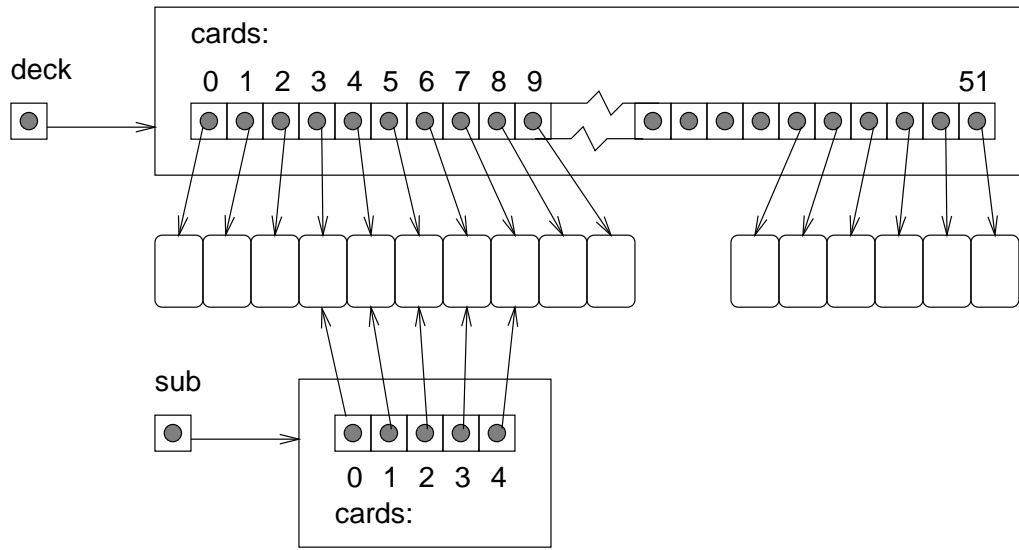
I have suggested that aliasing is not generally a good idea, since changes in one subdeck will be reflected in others, which is not the behavior you would expect from real cards and decks. But if the objects in question are immutable, then aliasing can be a reasonable choice. In this case, there is probably no reason ever to change the rank or suit of a card. Instead we will create each card once and then treat it as an immutable object. So for `Cards` aliasing is a reasonable choice.

As an exercise, write a version of `findBisect` that takes a subdeck as an argument, rather than a deck and an index range. Which version is more error-prone? Which version do you think is more efficient?

12.4   Shuffling and dealing

In Section **??** I wrote pseudocode for a shuffling algorithm. Assuming that we have a method called `shuffleDeck` that takes a deck as an argument and shuffles it, we can create and shuffle a deck:

```
Deck deck = new Deck ();
shuffleDeck (deck);
```

Then, to deal out several hands, we can use `subdeck`:

```
Deck hand1 = subdeck (deck, 0, 4);
Deck hand2 = subdeck (deck, 5, 9);
Deck pack = subdeck (deck, 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 out one card at a time 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. Beware of misleading analogies.

12.5   Mergesort

In Section **??**, we saw a simple sorting algorithm that turns out not to be very efficient. In order to sort n items, it has to traverse the array n times, and each traversal takes an amount of time that is proportional to n. The total time, therefore, is proportional to n^2.

In this section I will sketch a more efficient algorithm called **mergesort**. To sort n items, mergesort takes time proportional to $n \log n$. That may not seem impressive, but as n gets big, the difference between n^2 and $n \log n$ can be enormous. Try out a few values of n and see.

The basic idea behind mergesort is this: if you have two subdecks, each of which has been sorted, it is easy (and fast) to merge them into a single, sorted deck. Try this out with a deck of cards:

1. Form two subdecks with about 10 cards each and sort them so that when they are face up the lowest cards are on top. Place both decks face up in front of you.

2. Compare the top card from each deck and choose the lower one. Flip it over and add it to the merged deck.

3. Repeat step two until one of the decks is empty. Then take the remaining cards and add them to the merged deck.

The result should be a single sorted deck. Here's what this looks like in pseudocode:

```
public static Deck merge (Deck d1, Deck d2) {
  // create a new deck big enough for all the cards
  Deck result = new Deck (d1.cards.length + d2.cards.length);

  // use the index i to keep track of where we are in
  // the first deck, and the index j for the second deck
  int i = 0;
  int j = 0;
```

```
    // the index k traverses the result deck
    for (int k = 0; k<result.cards.length; k++) {

      // if d1 is empty, d2 wins; if d2 is empty, d1 wins;
      // otherwise, compare the two cards

      // add the winner to the new deck
    }
    return result;
  }
```

The best way to test `merge` is to build and shuffle a deck, use subdeck to form two (small) hands, and then use the sort routine from the previous chapter to sort the two halves. Then you can pass the two halves to `merge` to see if it works.

If you can get that working, try a simple implementation of `mergeSort`:

```
public static Deck mergeSort (Deck deck) {
  // find the midpoint of the deck
  // divide the deck into two subdecks
  // sort the subdecks using sortDeck
  // merge the two halves and return the result
}
```

Then, if you get that working, the real fun begins! The magical thing about mergesort is that it is recursive. At the point where you sort the subdecks, why should you invoke the old, slow version of `sort`? Why not invoke the spiffy new `mergeSort` you are in the process of writing?

Not only is that a good idea, it is *necessary* in order to achieve the performance advantage I promised. In order to make it work, though, you have to add a base case so that it doesn't recurse forever. A simple base case is a subdeck with 0 or 1 cards. If `mergesort` receives such a small subdeck, it can return it unmodified, since it is already sorted.

The recursive version of `mergesort` should look something like this:

```
public static Deck mergeSort (Deck deck) {
  // if the deck is 0 or 1 cards, return it

  // find the midpoint of the deck
  // divide the deck into two subdecks
  // sort the subdecks using mergesort
  // merge the two halves and return the result
}
```

As usual, there are two ways to think about recursive programs: you can think through the entire flow of execution, or you can make the "leap of faith." I

have deliberately constructed this example to encourage you to make the leap
of faith.

When you were using `sortDeck` to sort the subdecks, you didn't feel com-
pelled to follow the flow of execution, right? You just assumed that the
`sortDeck` method would work because you already debugged it. Well, all you
did to make `mergeSort` recursive was replace one sort algorithm with another.
There is no reason to read the program differently.

Well, actually you have to give some thought to getting the base case right
and making sure that you reach it eventually, but other than that, writing the
recursive version should be no problem. Good luck!

12.6   Glossary

pseudocode: A way of designing programs by writing rough drafts in a com-
bination of English and Java.

helper method: Often a small method that does not do anything enormously
useful by itself, but which helps another, more useful, method.

Chapter 13

Object-oriented programming

13.1 Programming languages and styles

There are many programming languages in the world, and almost as many programming styles (sometimes called paradigms). Three styles that have appeared in this book are procedural, functional, and object-oriented. Although Java is usually thought of as an object-oriented language, it is possible to write Java programs in any style. The style I have demonstrated in this book is pretty much procedural. Existing Java programs and the built-in Java packages are written in a mixture of all three styles, but they tend to be more object-oriented than the programs in this book.

It's not easy to define what object-oriented programming is, but here are some of its characteristics:

- Object definitions (classes) usually correspond to relevant real-world objects. For example, in Chapter **??**, the creation of the `Deck` class was a step toward object-oriented programming.

- The majority of methods are object methods (the kind you invoke on an object) rather than class methods (the kind you just invoke). So far all the methods we have written have been class methods. In this chapter we will write some object methods.

- The language feature most associated with object-oriented programming is **inheritance**. I will cover inheritance later in this chapter.

Recently object-oriented programming has become quite popular, and there are people who claim that it is superior to other styles in various ways. I hope that by exposing you to a variety of styles I have given you the tools you need to understand and evaluate these claims.

13.2   Object and class methods

There are two types of methods in Java, called **class methods** and **object methods**. So far, every method we have written has been a class method. Class methods are identified by the keyword `static` in the first line. Any method that does not have the keyword `static` is an object method.

Although we have not written any object methods, we have invoked some. Whenever you invoke a method "on" an object, it's an object method. For example, `drawOval` is an object method we invoked on `g`, which is a `Graphics` object. Also, the methods we invoked on `String`s in Chapter **??** were object methods.

Anything that can be written as a class method can also be written as an object method, and vice versa. Sometimes it is just more natural to use one or the other. For reasons that will be clear soon, object methods are often shorter than the corresponding class methods.

13.3   The current object

When you invoke a method on an object, that object becomes **the current object**. Inside the method, you can refer to the instance variables of the current object by name, without having to specify the name of the object.

Also, you can refer to the current object using the keyword `this`. We have already seen `this` used in constructors. In fact, you can think of constructors as being a special kind of object method.

13.4   Complex numbers

As a running example for the rest of this chapter we will consider a class definition for complex numbers. Complex numbers are useful for many branches of mathematics and engineering, and many computations are performed using complex arithmetic. A complex number is the sum of a real part and an imaginary part, and is usually written in the form $x + yi$, where x is the real part, y is the imaginary part, and i represents the square root of -1. Thus, $i \cdot i = -1$.

The following is a class definition for a new object type called `Complex`:

```
class Complex
{
  // instance variables
  double real, imag;

  // constructor
  public Complex () {
    this.real = 0.0;  this.imag = 0.0;
  }
```

```
  // constructor
  public Complex (double real, double imag) {
    this.real = real;  this.imag = imag;
  }
}
```

There should be nothing surprising here. The instance variables are two `doubles` that contain the real and imaginary parts. The two constructors are the usual kind: one takes no parameters and assigns default values to the instance variables, the other takes parameters that are identical to the instance variables. As we have seen before, the keyword `this` is used to refer to the object being initialized.

In `main`, or anywhere else we want to create `Complex` objects, we have the option of creating the object and then setting the instance variables, or doing both at the same time:

```
    Complex x = new Complex ();
    x.real = 1.0;
    x.imag = 2.0;
    Complex y = new Complex (3.0, 4.0);
```

13.5   A function on `Complex` numbers

Let's look at some of the operations we might want to perform on complex numbers. The absolute value of a complex number is defined to be $\sqrt{x^2 + y^2}$. The `abs` method is a pure function that computes the absolute value. Written as a class method, it looks like this:

```
  // class method
  public static double abs (Complex c) {
    return Math.sqrt (c.real * c.real + c.imag * c.imag);
  }
```

This version of `abs` calculates the absolute value of `c`, the `Complex` object it receives as a parameter. The next version of `abs` is an object method; it calculates the absolute value of the current object (the object the method was invoked on). Thus, it does not receive any parameters:

```
  // object method
  public double abs () {
    return Math.sqrt (real*real + imag*imag);
  }
```

I removed the keyword `static` to indicate that this is an object method. Also, I eliminated the unnecessary parameter. Inside the method, I can refer to the instance variables `real` and `imag` by name without having to specify an object. Java knows implicitly that I am referring to the instance variables of the current object. If I wanted to make it explicit, I could have used the keyword `this`:

```
// object method
public double abs () {
   return Math.sqrt (this.real * this.real + this.imag * this.imag);
}
```

But that would be longer and not really any clearer. To invoke this method, we invoke it on an object, for example

```
Complex y = new Complex (3.0, 4.0);
double result = y.abs ();
```

13.6   Another function on `Complex` numbers

Another operation we might want to perform on complex numbers is addition. You can add complex numbers by adding the real parts and adding the imaginary parts. Written as a class method, that looks like:

```
public static Complex add (Complex a, Complex b) {
   return new Complex (a.real + b.real, a.imag + b.imag);
}
```

To invoke this method, we would pass both operands as arguments:

```
Complex sum = add (x, y);
```

Written as an object method, it would take only one argument, which it would add to the current object:

```
public Complex add (Complex b) {
   return new Complex (real + b.real, imag + b.imag);
}
```

Again, we can refer to the instance variables of the current object implicitly, but to refer to the instance variables of b we have to name b explicitly using dot notation. To invoke this method, you invoke it on one of the operands and pass the other as an argument.

```
Complex sum = x.add (y);
```

From these examples you can see that the current object (`this`) can take the place of one of the parameters. For this reason, the current object is sometimes called an **implicit** parameter.

13.7 A modifier

As yet another example, we'll look at `conjugate`, which is a modifier method that transforms a `Complex` number into its complex conjugate. The complex conjugate of $x + yi$ is $x - yi$.

As a class method, this looks like:

```
public static void conjugate (Complex c) {
  c.imag = -c.imag;
}
```

As an object method, it looks like

```
public void conjugate () {
  imag = -imag;
}
```

By now you should be getting the sense that converting a method from one kind to another is a mechanical process. With a little practice, you will be able to do it without giving it much thought, which is good because you should not be constrained to writing one kind of method or the other. You should be equally familiar with both so that you can choose whichever one seems most appropriate for the operation you are writing.

For example, I think that `add` should be written as a class method because it is a symmetric operation of two operands, and it makes sense for both operands to appear as parameters. It just seems odd to invoke the method on one of the operands and pass the other as an argument.

On the other hand, simple operations that apply to a single object can be written most concisely as object methods (even if they take some additional arguments).

13.8 The `toString` method

There are two object methods that are common to many object types: `toString` and `equals`. `toString` converts the object to some reasonable string representation that can be printed. `equals` is used to compare objects.

When you print an object using `print` or `println`, Java checks to see whether you have provided an object method named `toString`, and if so it invokes it. If not, it invokes a default version of `toString` that produces the output described in Section **??**.

Here is what `toString` might look like for the `Complex` class:

```
public String toString () {
  return real + " + " + imag + "i";
}
```

The return type for `toString` is `String`, naturally, and it takes no parameters. You can invoke `toString` in the usual way:

```
Complex x = new Complex (1.0, 2.0);
String s = x.toString ();
```

or you can invoke it indirectly through `print`:

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

Whenever you pass an object to `print` or `println`, Java invokes the `toString` method on that object and prints the result. In this case, the output is `1.0 + 2.0i`.

This version of `toString` does not look good if the imaginary part is negative. As an exercise, fix it.

13.9   The equals method

When you use the `==` operator to compare two objects, what you are really asking is, "Are these two things the same object?" That is, do both objects refer to the same location in memory.

For many types, that is not the appropriate definition of equality. For example, two complex numbers are equal if their real parts are equal and their imaginary parts are equal.

When you create a new object type, you can provide your own definition of equality by providing an object method called `equals`. For the `Complex` class, this looks like:

```
public boolean equals (Complex b) {
  return (real == b.real && imag == b.imag);
}
```

By convention, `equals` is always an object method. The return type has to be `boolean`.

The documentation of `equals` in the `Object` class provides some guidelines you should keep in mind when you make up your own definition of equality:

> The `equals` method implements an equivalence relation:
>
> - It is reflexive: for any reference value x, `x.equals(x)` should return `true`.
>
> - It is symmetric: for any reference values x and y, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
>
> - It is transitive: for any reference values x, y, and z, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.

- It is consistent: for any reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`.
- For any reference value `x`, `x.equals(null)` should return `false`.

The definition of `equals` I provided satisfies all these conditions except one. Which one? As an exercise, fix it.

13.10 Invoking one object method from another

As you might expect, it is legal and common to invoke one object method from another. For example, to normalize a complex number, you divide through (both parts) by the absolute value. It may not be obvious why this is useful, but it is.

Let's write the method `normalize` as an object method, and let's make it a modifier.

```
public void normalize () {
  double d = this.abs();
  real = real/d;
  imag = imag/d;
}
```

The first line finds the absolute value of the current object by invoking `abs` on the current object. In this case I named the current object explicitly, but I could have left it out. If you invoke one object method within another, Java assumes that you are invoking it on the current object.

As an exercise, rewrite `normalize` as a pure function. Then rewrite it as a class method.

13.11 Oddities and errors

If you have both object methods and class methods in the same class definition, 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 might 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 might get "Can't make a static reference to nonstatic variable..." This is not one of the better error messages, since it uses some non-standard language. For example, by "nonstatic variable" it means "instance variable." But once you know what it means, you know what it means.

13.12 Inheritance

The language feature that is most often associated with object-oriented programming is **inheritance**. Inheritance is the ability to define a new class that is a modified version of a previously-defined class (including built-in classes).

The primary advantage of this feature is that you can add new methods or instance variables to an existing class without modifying the existing class. This is particularly useful for built-in classes, since you can't modify them even if you want to.

The reason inheritance is called "inheritance" is that the new class inherits all the instance variables and methods of the existing class. Extending this metaphor, the existing class is sometimes called the **parent** class.

13.13 Drawable rectangles

An an example of inheritance, we are going to take the existing `Rectangle` class and make it "drawable." That is, we are going to create a new class called `DrawableRectangle` that will have all the instance variables and methods of a `Rectangle`, plus an additional method called `draw` that will take a `Graphics` object as a parameter and draw the rectangle.

The class definition looks like this:

```
import java.awt.*;

class DrawableRectangle extends Rectangle {

  public void draw (Graphics g) {
    g.drawRect (x, y, width, height);
  }
}
```

Yes, that's really all there is in the whole class definition. The first line imports the `java.awt` package, which is where `Rectangle` and `Graphics` are defined.

The next line indicates that `DrawableRectangle` inherits from `Rectangle`. The keyword `extends` is used to identify the parent class.

The rest is the definition of the `draw` method, which refers to the instance variables `x`, `y`, `width` and `height`. It might seem odd to refer to instance variables that don't appear in this class definition, but remember that they are inherited from the parent class.

To create and draw a `DrawableRectangle`, you could use the following:

```
public static void draw
    (Graphics g, int x, int y, int width, int height) {
  DrawableRectangle dr = new DrawableRectangle ();
  dr.x = 10;  dr.y = 10;
  dr.width = 200;  dr.height = 200;
  dr.draw (g);
}
```

The parameters of `draw` are a `Graphics` object and the bounding box of the drawing area (not the coordinates of the rectangle).

It might seem odd to use the `new` command for a class that has no constructors. `DrawableRectangle` inherits the default constructor of its parent class, so there is no problem there.

We can set the instance variables of `dr` and invoke methods on it in the usual way. When we invoke `draw`, Java invokes the method we defined in `DrawableRectangle`. If we invoked `grow` or some other `Rectangle` method on `dr`, Java would know to use the method defined in the parent class.

13.14   The class hierarchy

In Java, all classes extend some other class. The most basic class is called `Object`. It contains no instance variables, but it does provide the methods `equals` and `toString`, among others.

Many classes extend `Object`, including almost all of the classes we have written and many of the built-in classes, like `Rectangle`. Any class that does not explicitly name a parent inherits from `Object` by default.

Some inheritance chains are longer, though. For example, `Slate` extends `Frame` (see Appendix **??**), which extends `Window`, which extends `Container`, which extends `Component`, which extends `Object`. No matter how long the chain, `Object` is the ultimate parent of all classes.

All the classes in Java can be organized into a "family tree" that is called the class hierarchy. `Object` usually appears at the top, with all the "child" classes below. If you look at the documentation of `Frame`, for example, you will see the part of the hierarchy that makes up `Frame`'s pedigree.

13.15   Object-oriented design

Inheritance is a powerful feature. Some programs that would be complicated without inheritance can be written concisely and simply with it. Also, inheri-

tance can facilitate code reuse, since you can customize the behavior of build-in classes without having to modify them.

On the other hand, inheritance can make programs difficult to read, since it is sometimes not clear, when a method is invoked, where to find the definition. For example, one of the methods you can invoke on a `Slate` is `getBounds`. Can you find the documentation for `getBounds`? It turns out that `getBounds` is defined in the parent of the parent of the parent of the parent of `Slate`.

Also, many of the things that can be done using inheritance can be done almost as elegantly (or more so) without it.

13.16   Glossary

object method: A method that is invoked on an object, and that operates on that object, which is referred to by the keyword `this` in Java or "the current object" in English. Object methods do not have the keyword `static`.

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 object method is invoked. Inside the method, the current object is referred to by `this`.

this: The keyword that refers to the current object.

implicit: Anything that is left unsaid or implied. Within an object method, you can refer to the instance variables implicitly (without naming the object).

explicit: Anything that is spelled out completely. Within a class method, all references to the instance variables have to be explicit.

Chapter 14

Linked lists

14.1 References in objects

In the last chapter we saw that the instance variables of an object can be arrays, and I mentioned that they can be objects, too.

One of the more interesting possibilities is that an object can contain a reference to another object of the same type. There is a common data structure, the **list**, that takes advantage of this feature.

Lists are made up of **nodes**, where each node contains a reference to the next node in the list. In addition, each node usually contains a unit of data called the **cargo**. In our first example, the cargo will be a single integer, but later we will write a **generic** list that can contain objects of any type.

14.2 The `Node` class

As usual when we write a new class, we'll start with the instance variables, one or two constructors and `toString` so that we can test the basic mechanism of creating and displaying the new type.

```
public class Node {
    int cargo;
    Node next;

    public Node () {
        cargo = 0;
        next = null;
    }

    public Node (int cargo, Node next) {
        this.cargo = cargo;
        this.next = next;
```

```
    }

    public String toString () {
        return cargo + "";
    }
}
```

The declarations of the instance variables follow naturally from the specification, and the rest follows mechanically from the instance variables. The expression `cargo + ""` is an awkward but concise way to convert an integer to a String.

   To test the implementation so far, we would put something like this in `main`:

```
    Node node = new Node (1, null);
    System.out.println (node);
```

The result is simply

```
1
```

To make it interesting, we need a list with more than one node!

```
    Node node1 = new Node (1, null);
    Node node2 = new Node (2, null);
    Node node3 = new Node (3, null);
```

This code creates three nodes, but we don't have a list yet because the nodes are not **linked**. The state diagram looks like this:
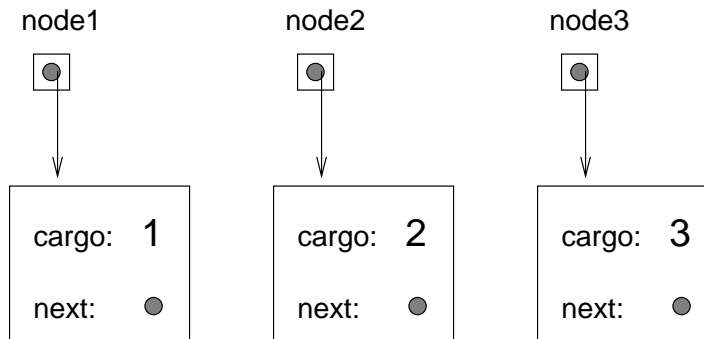
   To link up the nodes, we have to make the first node refer to the second and the second node refer to the third.

```
    node1.next = node2;
    node2.next = node3;
    node3.next = null;
```

The reference of the third node is `null`, which indicates that it is the end of the list. Now the state diagram looks like:
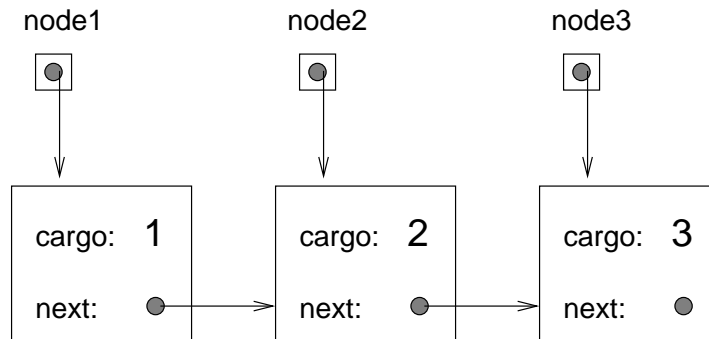
Now we know how to create nodes and link them into lists. What might be less clear at this point is why.

14.3   Lists as collections

The thing that makes lists useful is that they are a way of assembling multiple objects into a single entity, sometimes called a collection. In the example, the first node of the list serves as a reference to the entire list.

If we want to pass the list as a parameter, all we have to pass is a reference to the first node. For example, the method `printList` takes a single node as an argument. Starting with the head of the list, it prints each node until it gets to the end (indicated by the `null` reference).

```
public static void printList (Node list) {
    Node node = list;

    while (node != null) {
        System.out.print (node);
        node = node.next;
    }
    System.out.println ();
}
```

To invoke this method we just have to pass a reference to the first node:

```
printList (node1);
```

Inside `printList` we have a reference to the first node of the list, but there is no variable that refers to the other nodes. We have to use the `next` value from each node to get to the next node.

This diagram shows the value of `list` and the values that `node` takes on:
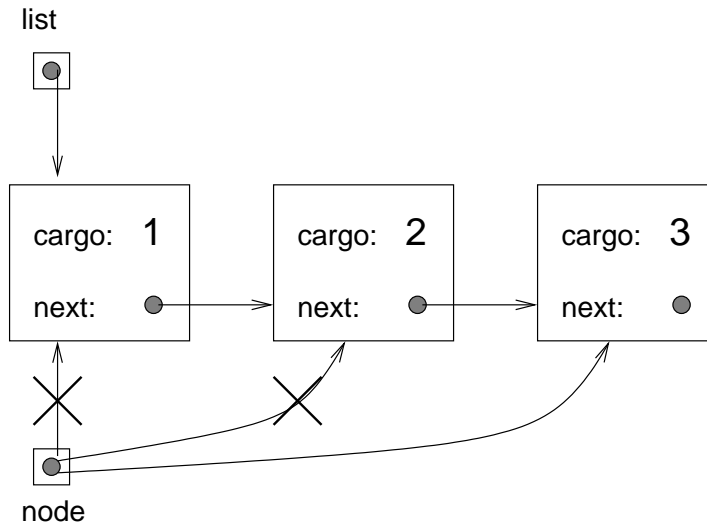
This way of moving through a list is called a **traversal**, just like the similar pattern of moving through the elements of an array. It is common to use a loop variable like `node` to refer to each of the nodes in the list in succession.

The output of this method is

```
123
```

By convention, lists are printed in parentheses with commas between the elements, as in (1, 2, 3). As an exercise, modify `printList` so that it generates output in this format.

As another exercise, rewrite `printList` using a `for` loop instead of a `while` loop.

14.4   Lists and recursion

Recursion and lists go together like fava beans and a nice Chianti. For example, here is a recursive algorithm for printing a list backwards:

1. Separate the list into two pieces: the first node (called the head) and the rest (called the tail).

2. Print the tail backwards.

3. Print the head.

Of course, Step 2, the recursive call, assumes that we have a way of printing a list backwards. But *if* we assume that the recursive call works—the leap of faith—then we can convince ourselves that this algorithm works.

All we need is a base case, and a way of proving that for any list we will eventually get to the base case. A natural choice for the base case is a list with a single element, but an even better choice is the empty list, represented by null.

```java
public static void printBackward (Node list) {
    if (list == null) return;

    Node head = list;
    Node tail = list.next;

    printBackward (tail);
    System.out.print (head);
}
```

The first line handles the base case by doing nothing. The next two lines split the list into `head` and `tail`. The last two lines print the list.

We invoke this method exactly as we invoked `printList`:

```java
printBackward (node1);
```

The result is a backwards list.

Can we prove that this method will always terminate? In other words, will it always reach the base case? In fact, the answer is no. There are some lists that will make this method crash.

14.5 Infinite lists

There is nothing to prevent a node from referring back to an earlier node in the list, including itself. For example, this figure shows a list with two nodes, one of which refers to itself.
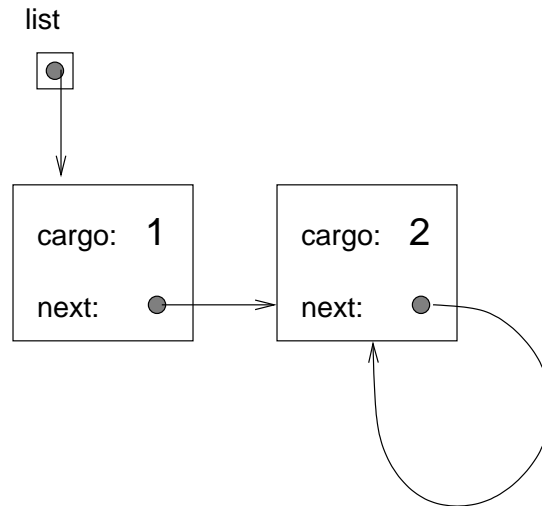
If we invoke `printList` on this list, it will loop forever. If we invoke `printBackward` it will recurse infinitely. This sort of behavior makes infinite lists difficult to work with.

Nevertheless, they are occasionally useful. For example, we might represent a number as a list of digits and use an infinite list to represent a repeating fraction.

Regardless, it is problematic that we cannot prove that `printList` and `printBackward` terminate. The best we can do is the hypothetical statement, "If the list contains no loops, then these methods will terminate." This sort of claim is called a **precondition**. It imposes a constraint on one of the parameters and describes the behavior of the method if the constraint is satisfied. We will see more examples soon.

14.6   The fundamental ambiguity theorem

There is a part of `printBackward` that might have raised an eyebrow:

```
Node head = list;
Node tail = list.next;
```

After the first assignment, `head` and `list` have the same type and the same value. So why did I create a new variable?

The reason is that the two variables play different roles. We think of `head` as a reference to a single node, and we think of `list` as a reference to the first node of a list. These "roles" are not part of the program; they are in the mind of the programmer.

The second assignment creates a new reference to the second node in the list, but in this case we think of it as a list. So, even though `head` and `tail` have the same type, they play different roles.

This ambiguity is useful, but it can make programs with lists difficult to read. I often use variable names like `node` and `list` to document how I intend to use a variable, and sometimes I create additional variables to disambiguate.

I could have written `printBackward` without `head` and `tail`, but I think it makes it harder to understand:

```
public static void printBackward (Node list) {
    if (list == null) return;

    printBackward (list.next);
    System.out.print (list);
}
```

Looking at the two function calls, we have to remember that `printBackward` treats its argument as a list and `print` treats its argument as a single object.

Always keep in mind the **fundamental ambiguity theorem**:

A variable that refers to a node might treat the node as a single
object or as the first in a list of nodes.

14.7 Object methods for nodes

You might have wondered why `printList` and `printBackward` are class meth-
ods. I have made the claim that anything that can be done with class methods
can also be done with object methods; it's just a question of which form is
cleaner.

In this case there is a legitimate reason to choose class methods. It is legal
to send `null` as an argument to a class method, but it is not legal to invoke an
object method on a null object.

```
Node node = null;
printList (node);        // legal
node.printList ();       // NullPointerException
```

This limitation makes it awkward to write list-manipulating code in a clean,
object-oriented style. A little later we will see a way to get around this, though.

14.8 Modifying lists

Obviously one way to modify a list is to change the cargo of one on the nodes,
but the more interesting operations are the ones that add, remove, or reorder
the nodes.

As an example, we'll write a method that removes the second node in the
list and returns a reference to the removed node.

```
public static Node removeSecond (Node list) {
    Node first = list;
    Node second = list.next;

    // make the first node refer to the third
    first.next = second.next;

    // separate the second node from the rest of the list
    second.next = null;
    return second;
}
```

Again, I am using temporary variables to make the code more readable. Here
is how to use this method.

```
printList (node1);
Node removed = removeSecond (node1);
printList (removed);
printList (node1);
```

The output is

```
(1, 2, 3)            the original list
(2)                  the removed node
(1, 3)               the modified list
```

Here is a state diagram showing the effect of this operation.
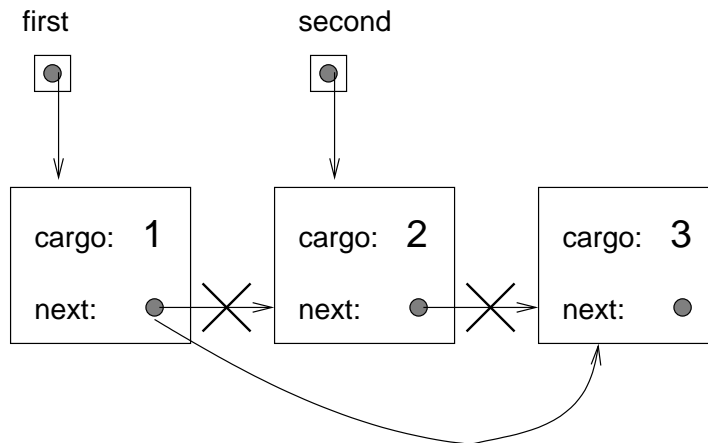
What happens if we invoke this method and pass a list with only one element (a **singleton**)? What happens if we pass the empty list as an argument? Is there a precondition for this method?

14.9   Wrappers and helpers

For some list operations it is useful to divide the labor into two methods. For example, to print a list backwards in the conventional list format, (3, 2, 1) we can use the `printBackwards` method to print 3, 2, but we need a separate method to print the parentheses and the first node. We'll call it `printBackwardNicely`.

```
public static void printBackwardNicely (Node list) {
    System.out.print ("(");

    if (list != null) {
        Node head = list;
        Node tail = list.next;
        printBackward (tail);
        System.out.print (head);
    }
    System.out.println (")");
}
```

Again, it is a good idea to check methods like this to see if they work with special cases like an empty list or a singleton.

Elsewhere in the program, when we use this method, we will invoke `printBackwardNicely` directly and it will invoke `printBackward` on our behalf. In that sense, `printBackwardNicely` acts as a **wrapper**, and it uses `printBackward` as a helper.

14.10 The `LinkedList` class

There are a number of subtle problems with the way we have been implementing lists. In a reversal of cause and effect, I will propose an alternative implementation first and then explain what problems it solves.

First, we will create a new class called `LinkedList`. Its instance variables are an integer that contains the length of the list and a reference to the first node in the list. LinkedList objects serve as handles for manipulating lists of Node objects.

```
public class LinkedList {
    int length;
    Node head;

    public LinkedList () {
        length = 0;
        head = null;
    }
}
```

One nice thing about the `LinkedList` class is that it gives us a natural place to put wrapper functions like `printBackwardNicely`, which we can make an object method in the `LinkedList` class.

```
    public void printBackward () {
        System.out.print ("(");

        if (head != null) {
            Node tail = head.next;
            Node.printBackward (tail);
            System.out.print (head);
        }
        System.out.println (")");
    }
```

Just to make things confusing, I renamed `printBackwardNicely`. Now there are two methods named `printBackward`: one in the `Node` class (the helper) and one in the `LinkedList` class (the wrapper). In order for the wrapper to invoke the helper, it has to identify the class explicitly (`Node.printBackward`).

So, one of the benefits of the `LinkedList` class is that it provides a nice place to put wrapper functions. Another is that it makes it easier to add or remove the first element of a list. For example, `addFirst` is an object method for `LinkedLists`; it takes an `int` as an argument and puts it at the beginning of the list.

```
public void addFirst (int i) {
    Node node = new Node (i, head);
    head = node;
    length++;
}
```

As always, to check code like this it is a good idea to think about the special cases. For example, what happens if the list is initially empty?

14.11   Invariants

Some lists are "well-formed;" others are not. For example, if a list contains a loop, it will cause many of our methods to crash, so we might want to require that lists contain no loops. Another requirement is that the `length` value in the `LinkedList` object should be equal to the actual number of nodes in the list.

Requirements like this are called **invariants** because, ideally, they should be true of every object all the time. Specifying invariants for objects is a useful programming practice because it makes it easier to prove the correctness of code, check the integrity of data structures, and detect errors.

One thing that is sometimes confusing about invariants is that there are some times when they are violated. For example, in the middle of `addFirst`, after we have added the node, but before we have incremented `length`, the invariant is violated. This kind of violation is acceptable; in fact, it is often impossible to modify an object without violating an invariant for at least a little while. Normally the requirement is that every method that violates an invariant must restore the invariant.

If there is any significant stretch of code in which the invariant is violated, it is important for the comments to make that clear, so that no operations are performed that depend on the invariant.

14.12   Glossary

list: A data structure that implements a collection using a sequence of linked nodes.

node: An element of a list, usually implemented as an object that contains a reference to another object of the same type.

cargo: An item of data contained in a node.

link: An object reference embedded in an object.

generic data structure: A kind of data structure that can contain data of any type.

precondition: An assertion that must be true in order for a method to work correctly.

invariant: An assertion that should be true of an object at all times (except maybe while the object is being modified).

wrapper method: A method that acts as a middle-man between a caller and a helper method, often offering an interface that is cleaner than the helper method's.

Chapter 15

Stacks

15.1 Abstract data types

The data types we have looked at so far are all concrete, in the sense that we have completely specified how they are implemented. For example, the `Card` class represents a card using two integers. As I discussed at the time, that is not the only way to represent a card; there are many alternative implementations.

An **abstract data type**, or ADT, specifies a set of operations (or methods) and the semantics of the operations (what they do) but it does not not specify the implementation of the operations. That's what makes it abstract.

Why is that useful?

- It simplifies the task of specifying an algorithm if you can denote the operations you need without having to think at the same time about how the operations are performed.

- Since there are usually many ways to implement an ADT, it might be useful to write an algorithm that can be used with any of the possible implementations.

- Well-known ADTs, like the `Stack` ADT in this chapter, are often implemented in standard libraries so they can be written once and used by many programmers.

- The operations on ADTs provide a common high-level language for specifying and talking about algorithms.

When we talk about ADTs, we often distinguish the code that uses the ADT, called the **client** code, from the code that implements the ADT, called **provider** code because it provides a standard set of services.

15.2   The Stack ADT

In this chapter we will look at one common ADT, the stack. A stack is a collection, meaning that it is a data structure that contains multiple elements. Other collections we have seen include arrays and lists.

As I said, an ADT is defined by the operations you can perform on it. Stacks can perform only the following operations:

constructor: Create a new, empty stack.

`push`: Add a new item to the stack.

`pop`: Remove and return an item from the stack. The item that is returned is always the last one that was added.

`empty`: Check whether the stack is empty.

A stack is sometimes called a "last in, first out," or LIFO data structure, because the last item added is the first to be removed.

15.3   The Java Stack Object

Java provides a built-in object type called `Stack` that implements the Stack ADT. You should make some effort to keep these two things—the ADT and the Java implementation—straight. Before using the `Stack` class, we have to import it from `java.util`.

Then the syntax for constructing a new `Stack` is

```
Stack stack = new Stack ();
```

Initially the stack is empty, as we can confirm with the `empty` method, which returns a `boolean`:

```
System.out.println (stack.empty ());
```

A stack is a generic data structure, which means that we can add any type of item to it. In the Java implementation, though, we can only add object types. For our first example, we'll use `Node` objects, as defined in the previous chapter. Let's start by creating and printing a short list.

```
LinkedList list = new LinkedList ();
list.addFirst (3);
list.addFirst (2);
list.addFirst (1);
list.print ();
```

The output is (1, 2, 3). To put a `Node` object onto the stack, use the `push` method:

```
stack.push (list.head);
```

The following loop traverses the list and pushes all the nodes onto the stack:

```
for (Node node = list.head; node != null; node = node.next) {
    stack.push (node);
}
```

We can remove an element from the stack with the pop method.

```
Object obj = stack.pop ();
```

The return type from pop is Object! That's because the stack implementation doesn't really know the type of the objects it contains. When we pushed the Node objects, they were automatically converted to Objects. When we get them back from the stack, we have to cast them back to Nodes.

```
Node node = (Node) obj;
System.out.println (node);
```

Unfortunately, the burden falls on the programmer to keep track of the objects in the stack and cast them back to the right type when they are removed. If you try to cast an object to the wrong type, you get a ClassCastException.

The following loop is a common idiom for popping all the elements from a stack, stopping when it is empty:

```
while (!stack.empty ()) {
    Node node = (Node) stack.pop ();
    System.out.print (node + " ");
}
```

The output is 3 2 1. In other words, we just used a stack to print the elements of a list backwards! Granted, it's not the standard format for printing a list, but using a stack it was remarkably easy to do.

You should compare this code to the implementations of printBackward in the previous chapter. There is a natural parallel between the recursive version of printBackward and the stack algorithm here. The difference is that printBackward uses the run-time stack to keep track of the nodes while it traverses the list, and then prints them on the way back from the recursion. The stack algorithm does the same thing, just using a Stack object instead of the run-time stack.

15.4   Wrapper classes

For every primitive type in Java, there is a built-in object type called a **wrapper class**. For example, the wrapper class for int is called Integer; for double it is called Double.

Wrapper classes are useful for several reasons:

- You can instantiate wrapper classes and create objects that contain primitive values. In other words, you can wrap a primitive value up in an object, which is useful if you want to invoke a method that requires an object type.

- Each wrapper class contains special values (like the minimum and maximum values for the type), and methods that are useful for converting between types.

15.5   Creating wrapper objects

The most straightforward way to create a wrapper object is to use its constructor:

```
Integer i = new Integer (17);
Double d = new Double (3.14159);
Character c = new Character ('b');
```

Technically `String` is not a wrapper class, because there is no corresponding primitive type, but the syntax for creating a `String` object is the same:

```
String s = new String ("fred");
```

On the other hand, no one ever uses the constructor for `String` objects, because you can get the same effect with a simple `String` value:

```
String s = "fred";
```

15.6   Creating more wrapper objects

Some of the wrapper classes have a second constructor that takes a `String` as an argument and tries to convert to the appropriate type. For example:

```
Integer i = new Integer ("17");
Double d = new Double ("3.14159");
```

The type conversion process is not very robust. For example, if the `String`s are not in the right format, they will cause a `NumberFormatException`. Any non-numeric character in the `String`, including a space, will cause the conversion to fail.

```
Integer i = new Integer ("17.1");       // WRONG!!
Double d = new Double ("3.1459 ");      // WRONG!!
```

It is usually a good idea to check the format of the `String` before you try to convert it.

15.7 Getting the values out

Java knows how to print wrapper objects, so the easiest way to extract a value is just to print the object:

```
Integer i = new Integer (17);
Double d = new Double (3.14159);
System.out.println (i);
System.out.println (d);
```

Alternatively, you can use the `toString` method to convert the contents of the wrapper object to a String

```
String istring = i.toString();
String dstring = d.toString();
```

Finally, if you just want to extract the primitive value from the object, there is an object method in each wrapper class that does the job:

```
int iprim = i.intValue ();
double dprim = d.doubleValue ();
```

There are also methods for converting wrapper objects into different primitive types. You should check out the documentation for each wrapper class to see what is available.

15.8 Useful methods in the wrapper classes

As I mentioned, the wrapper classes contain useful methods that pertain to each type. For example, the `Character` class contains lots of methods for converting characters to upper and lower case, and for checking whether a character is a number, letter, or symbol.

The `String` class also contains methods for converting to upper and lower case. Keep in mind, though, that they are functions, not modifiers (see Section **??**).

As another example, the `Integer` class contains methods for interpreting and printing integers in different bases. If you have a `String` that contains a number in base 6, you can convert to base 10 using `parseInt`.

```
String base6 = "12345";
int base10 = Integer.parseInt (base6, 6);
System.out.println (base10);
```

Since `parseInt` is a class method, you invoke it by naming the class and the method in dot notation.

Base 6 might not be all that useful, but hexadecimal (base 16) and octal (base 8) are common for computer science related things.

15.9   Postfix expressions

In most programming languages, mathematical expressions are written with the operator between the two operands, as in 1+2. This format is called **infix**. An alternate format used by some calculators is called **postfix**. In postfix, the operator follows the operands, as in 1 2+.

The reason postfix is sometimes useful is that there is a natural way to evaluate a postfix expression using a stack.

- Starting at the beginning of the expression, get one term (operator or operand) at a time.

    - If the term is an operand, push it on the stack.
    - If the term is an operator, pop two operands off the stack, perform the operation on them, and push the result back on the stack.

- When we get to the end of the expression, there should be exactly one operand left on the stack. That operand is the result.

As an exercise, apply this algorithm to the expression 1 2 + 3 *.

This example demonstrates one of the advantages of postfix: there is no need to use parentheses to control the order of operations. To get the same result in infix, we would have to write (1 + 2) * 3. As an exercise, write a postfix expression that is equivalent to 1 + 2 * 3?

15.10   Parsing

In order to implement the algorithm from the previous section, we need to be able to traverse a string and break it into operands and operators. This process is an example of **parsing**, and the results—the individual chunks of the string—are called **tokens**.

Java provides a built-in class called a `StringTokenizer` that parses strings and breaks them into tokens. To use it, you have to import it from `java.util`.

In its simplest form, the `StringTokenizer` uses spaces to mark the boundaries between tokens. A character that marks a boundary is called a **delimiter**.

We can create a `StringTokenizer` in the usual way, passing as an argument the string we want to parse.

```
StringTokenizer st = new StringTokenizer ("Here are four tokens.");
```

The following loop is a standard idiom for extracting the tokens from a `StringTokenizer`.

```
while (st.hasMoreTokens ()) {
    System.out.println (st.nextToken());
}
```

The output is

```
Here
are
four
tokens.
```

For parsing expressions, we have the option of specifying additional characters that will be used as delimiters:

```
    StringTokenizer st = new StringTokenizer ("11 22+33*", " +-*/");
```

The second argument is a `String` that contains all the characters that will be used as delimiters. Now the output is:

```
11
22
33
```

This succeeds at extracting all the operands but we have lost the operators. Fortunately, there is one more option for `StringTokenizers`.

```
    StringTokenizer st = new StringTokenizer ("11 22+33*", " +-*/", true);
```

The third argument says, "Yes, we would like to treat the delimiters as tokens." Now the output is

```
11

22
+
33
*
```

This is just the stream of tokens we would like for evaluating this expression.

15.11 Implementing ADTs

One of the fundamental goals of an ADT is to separate the interests of the provider, who writes the code that implements the ADT, and the client, who uses the ADT. The provider only has to worry about whether the implementation is correct—in accord with the specification of the ADT—and not how it will be used.

Conversely, the client *assumes* that the implementation of the ADT is correct and doesn't worry about the details. When you are using one of Java's built-in classes, you have the luxury of thinking exclusively as a client.

When you implement an ADT, on the other hand, you also have to write client code to test it. In that case, you sometimes have to think carefully about which role you are playing at a given instant.

In the next few sections we will switch gears and look at one way of implementing the Stack ADT, using an array. Start thinking like a provider.

15.12   Array implementation of the Stack ADT

The instance variables for this implementation are an array of `Objects`, which
will contain the items on the stack, and an integer index which will keep track
of the next available space in the array. Initially, the array is empty and the
index is 0.

To add an element to the stack (`push`), we'll copy a reference to it onto
the stack and increment the index. To remove an element (`pop`) we have to
decrement the index first and then copy the element out.

Here is the class definition:

```
public class Stack {
    Object[] array;
    int index;

    public Stack () {
        this.array = new Object[128];
        this.index = 0;
    }
}
```

As usual, once we have chosen the instance variables, it is a mechanical process
to write a constructor. For now, the default size is 128 items. Later we will
consider better ways of handling this.

Checking for an empty stack is trivial.

```
    public boolean empty () {
        return index == 0;
    }
```

It it important to remember, though, that the number of elements in the stack
is not the same as the size of the array. Initially the size is 128, but the number
of elements is 0.

The implementations of `push` and `pop` follow naturally from the specification.

```
    public void push (Object item) {
        array[index] = item;
        index++;
    }

    public Object pop () {
        index--;
        return array[index];
    }
```

To test these methods, we can take advantage of the client code we used
to exercise the built-in Stack. All we have to do is comment out the line

`import java.util.Stack`. Then, instead of using the stack implementation from `java.util` the program will use the implementation we just wrote.

If everything goes according to plan, the program should work without any additional changes. Again, one of the strengths of using an ADT is that you can change implementations without changing client code.

15.13 Resizing arrays

A weakness of this implementation is that it chooses an arbitrary size for the array when the `Stack` is created. If the user pushes more than 128 items onto the stack, it will cause an `ArrayIndexOutOfBounds` exception.

An alternative is to let the client code specify the size of the array. This alleviates the problem, but it requires the client to know ahead of time how many items are needed, and that is not always possible.

A better solution is to check whether the array is full and make it bigger when necessary. Since we have no idea how big the array needs to be, it is a reasonable strategy to start with a small size and double it each time it overflows.

Here's the improved version of `push`:

```
public void push (Object item) {
    if (full ()) resize ();

    // at this point we can prove that index < array.length

    array[index] = item;
    index++;
}
```

Before putting the new item in the array, we check if the array is full. If so, we invoke `resize`. After the `if` statement, we know that either (1) there was room in the array, or (2) the array has been resized and there is room. If `full` and `resize` are correct, then we can prove that `index < array.length`, and therefore the next statement cannot cause an exception.

Now all we have to do is implement `full` and `resize`.

```
private boolean full () {
    return index == array.length;
}

private void resize () {
    Object[] newArray = new Object[array.length * 2];

    // we assume that the old array is full
    for (int i=0; i<array.length; i++) {
        newArray[i] = array[i];
    }
```

```
        array = newArray;
    }
```

Both methods are declared `private`, which means that they cannot be invoked from another class, only from within this one. This is acceptable, since there is no reason for client code to use these functions, and desirable, since it enforces the boundary between the implementation and the client.

The implementation of `full` is trivial; it just checks whether the index has gone beyond the range of valid indices.

The implementation of `resize` is straightforward, with the caveat that it assumes that the old array is full. In other words, that assumption is a precondition of this method. It is easy to see that this precondition is satisfied, since the only way `resize` is invoked is if `full` returns true, which can only happen if `index == array.length`.

At the end of `resize`, we replace the old array with the new one (causing the old to be garbage collected). The new `array.length` is twice as big as the old, and `index` hasn't changed, so now it must be true that `index < array.length`. This assertion is a **postcondition** of `resize`: something that must be true when the method is complete (as long as its preconditions were satisfied).

Preconditions, postconditions, and invariants are useful tools for analyzing programs and demonstrating their correctness. In this example I have demonstrated a programming style that facilitates program analysis and a style of documentation that helps demonstrate correctness.

15.14   Glossary

abstract data type (ADT): A data type (usually a collection of objects) that is defined by a set of operations, but that can be implemented in a variety of ways.

client: A program that uses an ADT (or the person who wrote the program).

provider: The code that implements an ADT (or the person who wrote it).

wrapper class: One of the Java classes, like `Double` and `Integer` that provide objects to contain primitive types, and methods that operate on primitives.

private: A Java keyword that indicates that a method or instance variable cannot be accessed from outside the current class definition.

infix: A way of writing mathematical expressions with the operators between the operands.

postfix: A way of writing mathematical expressions with the operators after the operands.

parse: To read a string of characters or tokens and analyze their grammatical structure.

token: A set of characters that are treated as a unit for purposes of parsing, like the words in a natural language.

delimiter: A character that is used to separate tokens, like the punctuation in a natural language.

predicate: A mathematical statement that is either true or false.

postcondition: A predicate that must be true at the end of a method (provided that the preconditions were true at the beginning).

Chapter 16

Queues and Priority Queues

This chapter presents two ADTs: Queues and Priority Queues. In real life a **queue** is a line of customers waiting for service of some kind. In most cases, the first customer in line is the next customer to be served. There are exceptions, though. For example, at airports customers whose flight is leaving imminently are sometimes taken from the middle of the queue. Also, at supermarkets a polite customer might let someone with only a few items go first.

The rule that determines who goes next is called a **queueing discipline**. The simplest queueing discipline is called **FIFO**, for "first-in-first-out." The most general queueing discipline is **priority queueing**, in which each customer is assigned a priority, and the customer with the highest priority goes first, regardless of the order of arrival. The reason I say this is the most general discipline is that the priority can be based on anything: what time a flight leaves, how many groceries the customer has, or how important the customer is. Of course, not all queueing disciplines are "fair," but fairness is in the eye of the beholder.

The Queue ADT and the Priority Queue ADT have the same set of operations and their interfaces are the same. The difference is in the semantics of the operations: a Queue uses the FIFO policy, and a Priority Queue (as the name suggests) uses the priority queueing policy.

As with most ADTs, there are a number of ways to implement queues Since a queue is a collection of items, we can use any of the basic mechanisms for storing collections: arrays, lists, or vectors. Our choice among them will be based in part on their performance— how long it takes to perform the operations we want to perform— and partly on ease of implementation.

16.1 The queue ADT

The queue ADT is defined by the following operations:

constructor: Create a new, empty queue.

`insert:` Add a new item to the queue.

`remove:` Remove and return an item from the queue. The item that is returned is the first one that was added.

`empty:` Check whether the queue is empty.

To demonstrate a queue implementation, I will take advantage of the `LinkedList` class from Chapter **??**. Also, I will assume that we have a class named `Customer` that defines all the information about each customer, and the operations we can perform on customers.

As far as our implementation goes, it does not matter what kind of object is in the `Queue`, so we can make it generic. Here is what the implementation looks like.

```
public class Queue {
    public LinkedList list;

    public Queue () {
        list = new List ();
    }

    public boolean empty () {
        return list.empty ();
    }

    public void insert (Object obj) {
        list.addLast (obj);
    }

    public Object remove () {
        return list.removeFirst ();
    }
}
```

A queue object contains a single instance variable, which is the list that implements it. For each of the other methods, all we have to do is invoke one of the methods from the `LinkedList` class.

16.2   Veneer

An implementation like this is called a **veneer**. In real life, veneer is a thin coating of good quality wood used in furniture-making to hide lower quality wood underneath. Computer scientists use this metaphor to describe a small piece of code that hides the details of an implementation and provides a simpler, or more standard, interface.

This example demonstrates one of the nice things about a veneer, which is that it is easy to implement, and one of the dangers of using a veneer, which is the **performance hazard**!

Normally when we invoke a method we are not concerned with the details of its implementation. But there is one "detail" we might want to know—the performance characteristics of the method. How long does it take, as a function of the number of items in the list?

First let's look at `removeFirst`.

```
public Object removeFirst () {
    Object result = head;
    if (head != null) {
        head = head.next;
    }
    return result;
}
```

There are no loops or function calls here, so that suggests that the run time of this method is the same every time. Such a method is called a **constant time** operation. In reality, the method might be slightly faster when the list is empty, since it skips the body of the conditional, but that difference is not significant.

The performance of `addLast` is very different.

```
public void addLast (Object obj) {
    // special case: empty list
    if (head == null) {
        head = new Node (obj, null);
        return;
    }
    Node last;
    for (last = head; last.next != null; last = last.next) {
        // traverse the list to find the last node
    }
    last.next = new Node (obj, null);
}
```

The first conditional handles the special case of adding a new node to an empty list. In this case, again, the run time does not depend on the length of the list. In the general case, though, we have to traverse the list to find the last element so we can make it refer to the new node.

This traversal takes time proportional to the length of the list. Since the run time is a linear function of the length, we would say that this method is **linear time**. Compared to constant time, that's very bad.

16.3   Linked Queue

We would like an implementation of the Queue ADT that can perform all operations in constant time. One way to accomplish that is to implement a **linked queue**, which is similar to a linked list in the sense that it is made up of zero or more linked `Node` objects. The difference is that the queue maintains a reference to both the first and the last node, as shown in the figure.
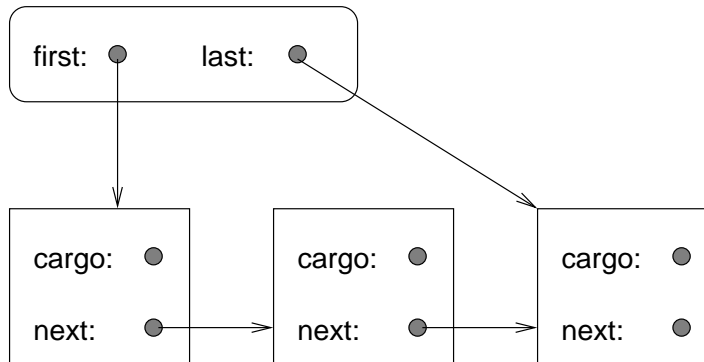
Here's what a linked `Queue` implementation looks like:

```java
public class Queue {
    public Node first, last;

    public Queue () {
        first = null;
        last = null;
    }

    public boolean empty () {
        return first == null;
    }
}
```

So far it is straightforward. In an empty queue, both `first` and `last` are null. To check whether a list is empty, we only have to check one of them.

`insert` is a little more complicated because we have to deal with several special cases.

```java
    public void insert (Object obj) {
        Node node = new Node (obj, null);
        if (last != null) {
            last.next = node;
        }
        last = node;
```

```
        if (first == null) {
            first = last;
        }
    }
```

The first condition checks to make sure that `last` refers to a node; if it does then we have to make it refer to the new node.

The second condition deals with the special case where the list was initially empty. In this case both `first` and `last` refer to the new node.

`remove` also deals with several special cases.

```
    public Object remove () {
        Node result = first;
        if (first != null) {
            first = first.next;
        }
        if (first == null) {
            last = null;
        }
        return result;
    }
```

The first condition checks whether there were any nodes in the queue. If so, we have to copy the `next` node into `first`. The second condition deals with the special case that the list is now empty, in which case we have to make `last` null.

As an exercise, draw diagrams showing both operations in both the normal case and in the special cases, and convince yourself that they are correct.

Clearly, this implementation is more complicated than the veneer implementation, and it is more difficult to demonstrate that it is correct. The advantage is that we have achieved the goal: both `insert` and `remove` are constant time.

16.4 Circular buffer

Another common implementation of a queue is a **circular buffer**. "Buffer" is a general name for a temporary storage location, although it often refers to an array, as it does in this case. What it means to say a buffer is "circular" should become clear in a minute.

The implementation of a circular buffer is similar to the array implementation of a stack, as in Section **??**. The queue items are stored in an array, and we use indices to keep track of where we are in the array. In the stack implementation, there was a single index that pointed to the next available space. In the queue implementation, there are two indices: `first` points to the space in the array that contains the first customer in line and `next` points to the next available space.

The following figure shows a queue with two items (represented by dots).

first   next

●	●				

first   next

| 0 | | 2 |

first   next
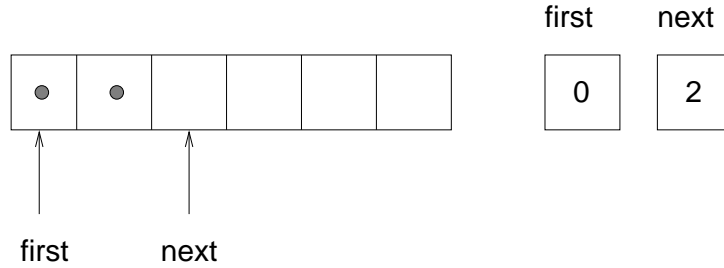
There are two ways to think of the variables `first` and `last`. Literally, they are integers, and their values are shown in boxes on the right. Abstractly, though, they are indices of the array, and so they are often drawn as arrows pointing to locations in the array. The arrow representation is convenient, but you should remember that the indices are not references; they are just integers.

Here is an incomplete array implementation of a queue:

```
public class Queue {
    public Object[] array;
    public int first, next;

    public Queue () {
        array = new Object[128];
        first = 0;
        next = 0;
    }

    public boolean empty () {
        return first == next;
    }
```

The instance variables and the constructor are straightforward, although again we have the problem that we have to choose an arbitrary size for the array. Later we will solve that problem, as we did with the stack, by resizing the array if it gets full.

The implementation of `empty` is a little surprising. You might have thought that `first == 0` would indicate an empty queue, but that neglects the fact that the head of the queue is not necessarily at the beginning of the array. Instead, we know that the queue is empty if `head` equals `next`, in which case there are no items left. Once we see the implementation of `insert` and `remove`, that situation will more more sense.

```
    public void insert (Object item) {
        array[next] = item;
        next++;
    }
```

```
public Object remove () {
    Object result = array[first];
    first++;
    return result;
}
```

`insert` looks very much like `push` in Section **??**; it puts the new item in the next available space and then increments the index.

`remove` is similar. It takes the first item from the queue and then increments `first` so it refers to the new head of the queue. The following figure shows what the queue looks like after both items have been removed.
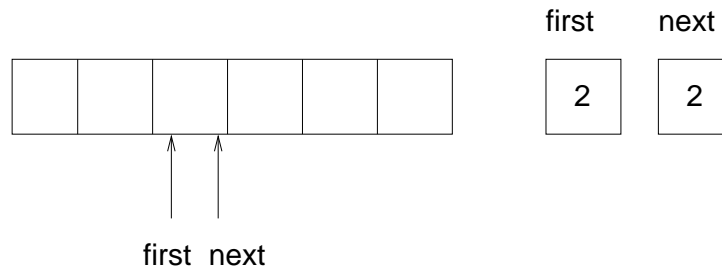
It is always true that `next` points to an available space. If `first` catches up with `next` and points to the same space, then `first` is referring to an "empty" location, and the queue is empty. I put "empty" in quotation marks because it is possible that the location that `first` points to actually contains a value (we do nothing to ensure that empty locations contain `null`); on the other hand, since we know the queue is empty, we will never read this location, so we can think of it, abstractly, as empty.

As an exercise, fix `remove` so that it returns `null` if the queue is empty.

The next problem with this implementation is that eventually it will run out of space. When we add an item we increment `next` and when we remove an item we increment `first`, but we never decrement either. What happens when we get to the end of the array?

The following figure shows the queue after we add four more items:
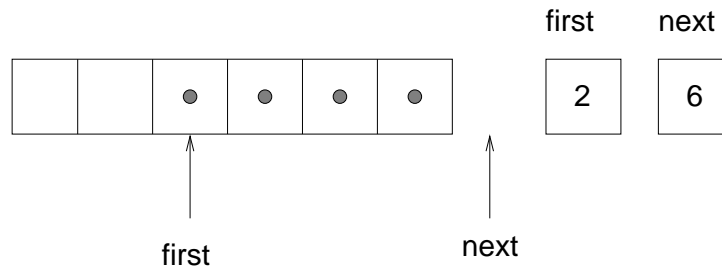
The array is now full. There is no "next available space," so there is nowhere for `next` to point. One possibility is that we could resize the array, as we did with the stack implementation. But in that case the array would keep getting

bigger regardless of how many items were actually in queue. A better solution is to wrap around to the beginning of the array and reuse the spaces there. This "wrap around" is the reason this implementation is called a circular buffer.

One way to wrap the index around is to add a special case whenever we increment an index:

```
next++;
if (next == array.length) next = 0;
```

A fancy alternative is to use the modulus operator:

```
next = (next + 1) % array.length;
```

Either way, we have one last problem to solve. How do we know if the queue is *really* full, meaning that we cannot insert another item? The following figure shows what the queue looks like when it is "full."
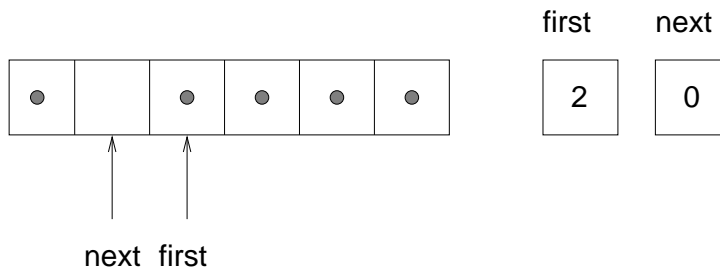
There is still one empty space in the array, but the queue is full because if we insert another item, then we have to increment `next` such that `next == first`, and in that case it would appear that the queue was empty!

To avoid that, we sacrifice one space in the array. So how can we tell if the queue is full?

```
if ((next + 1) % array.length == first)
```

And what should we do if the array is full? In that case resizing the array is probably the only option.

As an exercise, put together all the code from this section and write an implementation of a queue using a circular buffer that resizes itself when necessary.

16.5   Priority queue

The Priority Queue ADT has the same interface as the Queue ADT, but different semantics. The interface is:

constructor: Create a new, empty queue.

insert: Add a new item to the queue.

`remove:` Remove and return an item from the queue. The item that is returned is the one with the highest priority.

`empty:` Check whether the queue is empty.

The semantic difference is that the item that is removed from the queue is not necessarily the first one that was added. Rather, it is whatever item in the queue has the highest priority. What the priorities are, and how they compare to each other, are not specified by the Priority Queue implementation. It depends on what the items are that are in the queue.

For example, if the items in the queue have names, we might choose them in alphabetical order. If they are bowling scores, we might choose from highest to lowest, but if they are golf scores, we would go from lowest to highest.

So we face a new problem. We would like an implementation of Priority Queue that is generic—it should work with any kind of object—but at the same time the code that implements Priority Queue needs to have the ability to compare the objects it contains.

We have seen a way to implement generic data structures using `Objects`, but that does not solve this problem, because there is no way to compare `Objects` unless we know what type they are.

The answer lies in a new Java feature called an **abstract class**.

16.6   Abstract class

An abstract class is a set of classes. The abstract class definition specifies the requirements a class must satisfy to be a member.

Often abstract classes have names that end in "able" to indicate the fundamental capability the abstract class requires. For example, any class that provides a method named `draw` can be a member of the abstract class named `Drawable`. Any class that contains a method named `start` can be a member of the abstract class `Runnable`.

As of Java 2, Java provides a built-in abstract class that we can use in an implementation of a Priority Queue. It is called `Comparable`, and it means what it says. Any class that belongs to the `Comparable` abstract class has to provide a method named `compareTo` that compares two objects and returns a value indicating whether one is larger or smaller than the other, or whether they are the same.

Many of the built-in Java classes are members of the `Comparable` abstract class, including numeric wrapper classes like `Integer` and `Double`.

In the next section I will show how to write an ADT that manipulates an abstract class. Then we will see how to write a new (concrete) class that belongs to an existing abstract class. Then we will see how to write a new abstract class.

16.7   Array implementation of Priority Queue

In the implementation of the Priority Queue, every time we specify the type of
the items in the queue, we specify the abstract class `Comparable`. For example,
the instance variables are an array of `Comparables` and an integer:

```
public class PriorityQueue {
    private Comparable[] array;
    private int index;
}
```

As usual, `index` is the index of the next available location in the array. The
instance variables are declared `private` so that other classes cannot have direct
access to them.

The constructor and `empty` are similar to what we have seen before. I chose
the initial size for the array arbitrarily.

```
    public PriorityQueue () {
        array = new Comparable [16];
        index = 0;
    }

    public boolean empty () {
        return index == 0;
    }
```

`insert` is similar to `push`:

```
    public void insert (Comparable item) {
        if (index == array.length) {
            resize ();
        }
        array[index] = item;
        index++;
    }
```

I omitted the implementation of `resize`. The only substantial method in the
class is `remove`, which has to traverse the array to find and remove the largest
item:

```
    public Comparable remove () {
        if (index == 0) return null;

        int maxIndex = 0;

        // find the index of the item with the highest priority
        for (int i=1; i<index; i++) {
```

```
            if (array[i].compareTo (array[maxIndex]) > 0) {
                maxIndex = i;
            }
        }
        Comparable result = array[maxIndex];

        // move the last item into the empty slot
        index--;
        array[maxIndex] = array[index];
        return result;
    }
```

As we traverse the array, `maxIndex` keeps track of the index of the largest element we have seen so far. What it means to be the "largest" is determined by `compareTo`. In this case the `compareTo` method is provided by the `Integer` class, and it does what we expect—larger (more positive) numbers win.

16.8   A Priority Queue client

The implementation of Priority Queue is written entirely in terms of `Comparable` objects, but there is no such thing as a `Comparable` object! Go ahead, try to create one:

```
    Comparable comp = new Comparable ();        // ERROR
```

You'll get a compile-time message that says something like "java.lang.Comparable is an interface. It can't be instantiated." In Java, abstract classes are called **interfaces**. I have avoided this word so far because it also means several other things, but now you have to know.

Why can't abstract classes be instantiated? Because an abstract class only specifies requirements (you must have a `compareTo` method); it does not provide an implementation.

To create a `Comparable` object, you have to create one of the objects that belongs to the `Comparable` set, like `Integer`. Then you can use that object anywhere a `Comparable` is called for.

```
        PriorityQueue pq = new PriorityQueue ();
        Integer item = new Integer (17);
        pq.insert (item);
```

This code creates a new, empty Priority Queue and a new `Integer` object. Then it inserts the `Integer` into the queue. `insert` is expecting a `Comparable` as a parameter, so it is perfectly happy to take an `Integer`. If we try to pass a `Rectangle`, which does not belong to `Comparable`, we get a compile-time message like, "Incompatible type for method. Explicit cast needed to convert java.awt.Rectangle to java.lang.Comparable."

That's the compiler telling us that if we want to make that conversion, we have to do it explicitly. We might try to do what it says:

```
Rectangle rect = new Rectangle ();
pq.insert ((Comparable) rect);
```

But in that case we get a run-time error, a `ClassCastException`. When the `Rectangle` tries to pass as a `Comparable`, the run-time system checks whether it satisfies the requirements, and rejects it. So that's what we get for following the compiler's advise.

To get items out of the queue, we have to reverse the process:

```
while (!pq.empty ()) {
    item = (Integer) pq.remove ();
    System.out.println (item);
}
```

This loop removes all the items from the queue and prints them. It assumes that the items in the queue are `Integers`. If they were not, we would get a `ClassCastException`.

16.9   The `Golfer` class

Finally, let's looks at how we can make a new class that belongs to `Comparable`. As an example of something with an unusual definition of "highest" priority, we'll use golfers:

```
public class Golfer implements Comparable {
    String name;
    int score;

    public Golfer (String name, int score) {
        this.name = name;
        this.score = score;
    }
}
```

The class definition and the constructor are pretty much the same as always; the difference is that we have to declare that `Golfer implements Comparable`. In this case the keyword `implements` means that `Golfer` implements the interface specified by `Comparable`.

If we try to compile `Golfer.java` at this point, we get something like "class Golfer must be declared abstract.   It does not define int compareTo(java.lang.Object) from interface java.lang.Comparable." In other words, to be a `Comparable`, `Golfer` has to provide a method named `compareTo`. So let's write one:

```
    public int compareTo (Object obj) {
        Golfer that = (Golfer) obj;

        int a = this.score;
        int b = that.score;

        // for golfers, low is good!
        if (a<b) return 1;
        if (a>b) return -1;
        return 0;
    }
```

Two things here are a little surprising. First, the parameter is an `Object`. That's because in general the caller doesn't know what type the objects are that are being compared. For example, in `PriorityQueue.java` when we invoke `compareTo`, we pass a `Comparable` as a parameter. We don't have to know whether it is an `Integer` or a `Golfer` or whatever.

Inside `compareTo` we have to convert the parameter from an `Object` to a `Golfer`. As usual, there is a risk when we do this kind of cast: if we cast to the wrong type we get an exception.

Finally, we can create some golfers:

```
        Golfer tiger = new Golfer ("Tiger Woods", 61);
        Golfer phil = new Golfer ("Phil Mickelson", 72);
        Golfer hal = new Golfer ("Hal Sutton", 69);
```

And put them in the queue:

```
        pq.insert (tiger);
        pq.insert (phil);
        pq.insert (hal);
```

When we pull them out:

```
        while (!pq.empty ()) {
            golfer = (Golfer) pq.remove ();
            System.out.println (golfer);
        }
```

They appear in descending order (for golfers):

```
        Tiger Woods     61
        Hal Sutton      69
        Phil Mickelson  72
```

When we switched from `Integers` to `Golfers`, we didn't have to make any changes in `PriorityQueue.java` at all. So we succeeded in maintaining a barrier between `PriorityQueue` and the classes that use it, allowing us to reuse the code without modification. Furthermore, we were able to give the client code control over the definition of `compareTo`, making this implementation of `PriorityQueue` more versatile.

16.10   Glossary

queue: An ordered set of objects waiting for a service of some kind.

queueing discipline: The rules that determine which member of a queue is removed next.

FIFO: "first in, first out," a queueing discipline in which the first member to arrive is the first to be removed.

priority queue: A queueing discipline in which each member has a priority determined by external factors. The member with the highest priority is the first to be removed.

Priority Queue: An ADT that defines the operations one might perform on a priority queue.

veneer: A class definition that implements an ADT with method definitions that are invocations of other methods, sometimes with simple transformations. The veneer does no significant work, but it improves or standardizes the interface seen by the client.

performance hazard: A danger associated with a veneer that some of the methods might be implemented inefficiently in a way that is not apparent to the client.

constant time: An operation whose run time does not depend on the size of the data structure.

linear time: An operation whose run time is a linear function of the size of the data structure.

linked queue: An implementation of a queue using a linked list and references to the first and last nodes.

circular buffer: An implementation of a queue using an array and indices of the first element and the next available space.

abstract class: A set of classes. The abstract class specification lists the requirements a class must satisfy to be included in the set.

interface: The Java word for an abstract class. Not to be confused with the more broad meaning of the word interface.

Chapter 17

Trees

This chapter presents a new data structure called a tree, some of its uses and two ways to implement it.

A possible source of confusion is the distinction between an ADT, a data structure, and an implementation of an ADT or data structure. There is no universal answer, because something that is an ADT at one level might in turn be the implementation of another ADT.

To help keep some of this straight, it is sometimes useful to draw a diagram showing the relationship between an ADT and its possible implementations. This figure shows that there are two implementations of a tree:

Tree	
linked implementation	array implementation
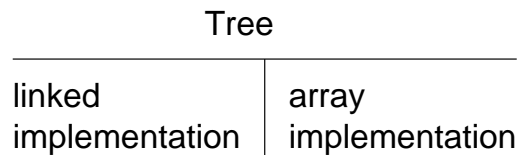
The horizontal line in the figure represents the barrier of abstraction between the ADT and its implementations.

17.1 A tree node

Like lists, trees are made up of nodes. A common kind of tree is a **binary tree**, in which each node contains a reference to two other nodes (possibly null). The class definition looks like this:

```
public class Tree {
    Object cargo;
    Tree left, right;
}
```

Like list nodes, tree nodes contain cargo: in this case a generic `Object`. The other instance variables are named `left` and `right`, in accordance with a standard way to represent trees graphically:
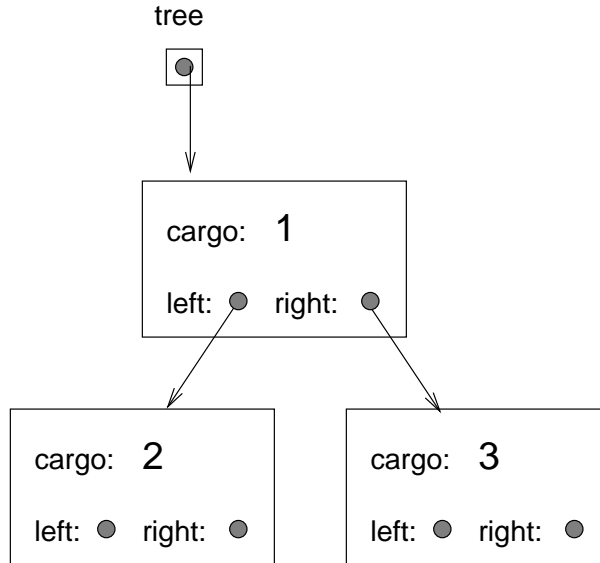
tree

The top of the tree (the node referred to by `tree`) is called the **root**. In keeping with the tree metaphor, the other nodes are called branches and the nodes at the tips with null references are called **leaves**. It may seem odd that we draw the picture with the root at the top and the leaves at the bottom, but that is not the strangest thing.

To make things worse, computer scientists mix in yet another metaphor: the family tree. The top node is sometimes called a **parent** and the nodes it refers to are its **children**. Nodes with the same parent are called **siblings**, and so on.

Finally, there is also a geometric vocabulary for taking about trees. I already mentioned left and right, but there is also "up" (toward the parent/root) and down (toward the children/leaves). Also, all the nodes that are the same distance from the root comprise a **level** of the tree.

I don't know why we need three metaphors for talking about trees, but there it is.

17.2   Building trees

The process of assembling tree nodes is similar to the process of assembling lists. We have a constructor for tree nodes that initializes the instance variables.

```
public Tree (Object cargo, Tree left, Tree right) {
    this.cargo = cargo;
    this.left = left;
```

```
        this.right = right;
    }
```

We allocate the child nodes first:

```
    Tree left = new Tree (new Integer(2), null, null);
    Tree right = new Tree (new Integer(3), null, null);
```

We can create the parent node and link it to the children at the same time:

```
    Tree tree = new Tree (new Integer(1), left, right);
```

This code produces the state shown in the previous figure.

17.3 Traversing trees

By now, any time you see a new data structure, your first question should be, "How can I traverse it?" The most natural way to traverse a tree is recursively. For example, to add up all the integers in a tree, we could write this class method:

```
    public static int total (Tree tree) {
        if (tree == null) return 0;
        Integer cargo = (Integer) tree.cargo;
        return cargo.intValue() + total (tree.left) + total (tree.right);
    }
```

This is a class method because we would like to use `null` to represent the empty tree, and make the empty tree the base case of the recursion. If the tree is empty, the method returns 0. Otherwise it makes two recursive calls to find the total value of its two children. Finally, it adds in its own cargo and returns the total.

Although this method works, there is some difficulty fitting it into an object-oriented design. It should not appear in the `Tree` class because it requires the cargo to be `Integer` objects. If we make that assumption in `Tree.java` then we lose the advantages of a generic data structure.

On the other hand, this code accesses the instance variables of the `Tree` nodes, so it "knows" more than it should about the implementation of the tree. If we changed that implementation later (and we will) this code would break.

Later in this chapter we will develop ways to solve this problem, allowing client code to traverse trees containing any kinds of objects without breaking the abstraction barrier between the client code and the implementation. Before we get there, let's look at an application of trees.

17.4   Expression trees

A tree is a natural way to represent the structure of an expression. Unlike other notations, it can represent the comptation unambiguously. For example, the infix expression 1 + 2 * 3 is ambiguous unless we know that the multiplication happens before the addition.

The following figure represents the same computation:
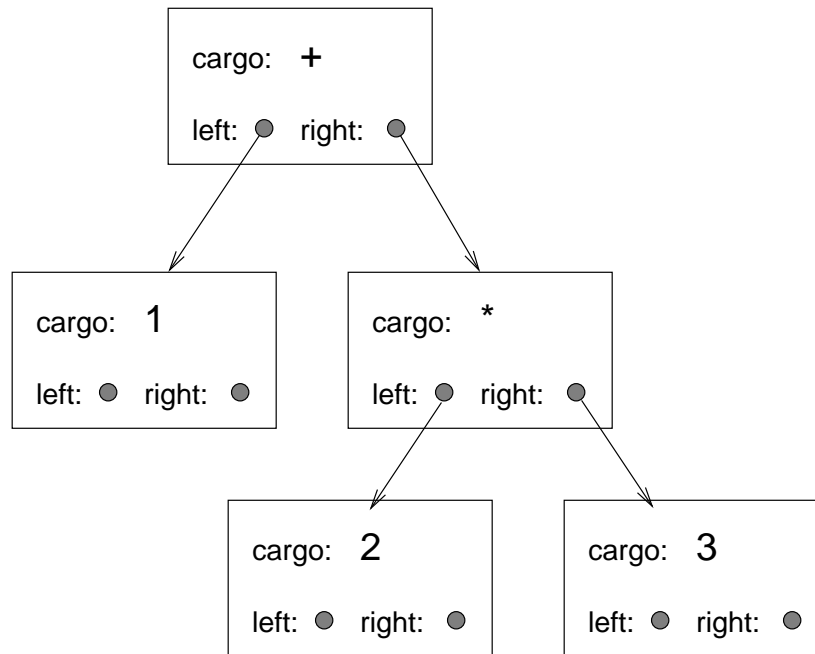
The nodes can be operands like 1 and 2 or operators like + and *. Operands are leaf nodes; operator nodes contain references to their operands (all of these operators are **binary**, meaning they have exactly two operands).

Looking at this figure, there is no question what the order of operations is: the multiplication happens first in order to compute the first operand of the addition.

Expression trees like this have many uses. The example we are going to look at is translation from one format (postfix) to another (infix). Similar trees are used inside compilers to parse, optimize and translate programs.

17.5   Traversal

I already pointed out that recursion provides a natural way to traverse a tree. We can print the contents of an expression tree like this:

```
public static void print (Tree tree) {
```

```
            if (tree == null) return;
            System.out.print (tree + " ");
            print (tree.left);
            print (tree.right);
      }
```

In other words, to print a tree, first print the contents of the root, then print the entire left subtree, then print the entire right subtree. This way of traversing a tree is called a **preorder**, because the contents of the root appear before the contents of the children.

For the example expression the output is + 1 * 2 3. This is different from both postfix and infix; it is a new notation called **prefix**, in which the operators appear before their operands.

You might suspect that if we traverse the tree in a different order we get the expression in a different notation. For example, if we print the subtrees first, and then the root node:

```
      public static void printPostorder (Tree tree) {
            if (tree == null) return;
            printPostorder (tree.left);
            printPostorder (tree.right);
            System.out.print (tree + " ");
      }
```

We get the expression in postfix (1 2 3 * +)! As the name of the previous method implies, this order of traversal is called **postorder**. Finally, to traverse a tree **inorder**, we print the left tree, then the root, then the right tree:

```
      public static void printInorder (Tree tree) {
            if (tree == null) return;
            printInorder (tree.left);
            System.out.print (tree + " ");
            printInorder (tree.right);
      }
```

The result is 1 + 2 * 3, which is the expression in infix.

To be fair, I have to point out that I have omitted an important complication. Sometimes when we write an expression in infix we have to use parentheses to preserve the order of operations. So an inorder traversal is not quite sufficient to generate an infix expression.

Nevertheless, with a few improvements, the expression tree and the three recursive traversals provide a general way to translate expressions from one format to another.

17.6 Encapsulation

As I mentioned before, there is a problem with the way we have been traversing trees: it breaks down the barrier between the client code (the application that

uses the tree) and the provider code (the Tree implementation). Ideally, tree code should be general; it shouldn't know anything about expression trees. And the code that generates and traverses the expression tree shouldn't know about the implementation of the trees. This design criterion is called **object encapsulation** to distinguish it from the encapsulation we saw in Section **??**, which we might call **method encapsulation**.

In the current version, the `Tree` code knows too much about the client. Instead, the `Tree` class should provide the general capability of traversing a tree in various ways. As it traverses, it should perform operations on each node that are specified by the client.

To facilitate this separation of interests, we will create a new abstract class, called `Visitable`. The items stored in a tree will be required to be visitable, which means that they define a method named `visit` that does whatever the client wants done to each node. That way the Tree can perform the traversal and the client can perform the node operations.

Here are the steps we have to perform to wedge an abstract class between a client and a provider:

1. Define an abstract class that specifies the methods the provider code will need to invoke on its components.

2. Write the provider code in terms of the new abstract class, as opposed to generic `Objects`.

3. Define a concrete class that belongs to the abstract class and that implements the required methods as appropriate for the client.

4. Write the client code to use the new concrete class.

The next few sections demonstrate these steps.

17.7   Defining an abstract class

An abstract class definition looks a lot like a concrete class definition, except that it only specifies the interface of each method and not an implementation. The definition of `Visitable` is

```
public interface Visitable {
    public void visit ();
}
```

That's it! The word `interface` is Java's keyword for an abstract class. The definition of `visit` looks like any other method definition, except that it has no body. This definition specifies that any class that implements `Visitable` has to have a method named `visit` that takes no parameters and that returns `void`.

Like other class definitions, abstract class definitions go in a file with the same name as the class (in this case `Visitable.java`).

17.8 Implementing an abstract class

If we are using an expression tree to generate infix, then "visiting" a node means printing its contents. Since the contents of an expression tree are tokens, we'll create a new concrete class called `Token` that implements `Visitable`

```
public class Token implements Visitable {
    String str;

    public Token (String str) {
        this.str = str;
    }

    public void visit () {
        System.out.print (str + " ");
    }
}
```

When we compile this class definition (which is in a file named `Token.java`), the compiler checks whether the methods provided satisfy the requirements specified by the abstract class. If not, it will produce an error message. For example, if we misspell the name of the method that is supposed to be `visit`, we might get something like, "class Token must be declared abstract. It does not define void visit() from interface Visitable."

The next step is to modify the parser to put `Token` objects into the tree instead of `Strings`. Here is a small example:

```
    String expr = "1 2 3 * +";
    StringTokenizer st = new StringTokenizer (expr, " +-*/", true);
    String token = st.nextToken();
    Tree tree = new Tree (new Token (token), null, null));
```

This code takes the first token in the string and wraps it in a `Token` object, then puts the `Token` into a tree node. If the `Tree` requires the cargo to be `Visitable`, it will convert the `Token` to be a `Visitable` object. When we remove the `Visitable` from the tree, we will have to cast it back into a `Token`.

As an exercise, write a version of `printPreorder` called `visitPreorder` that traverses the tree and invokes `visit` on each node in preorder.

17.9 Array implementation of trees

What does it mean to "implement" a tree? So far we have only seen one implementation of a tree, a linked data structure similar to a linked list. But there are other structures we would like to identify as trees. Anything that can perform the basic set of tree operations should be recognized as a tree.

So what are the tree operation? In other words, how do we define the Tree ADT?

constructor: Build an empty tree.

empty: Is this tree the empty tree?

left: Return the left child of this node, or an empty tree if there is none.

right: Return the left child of this node, or an empty tree if there is none.

parent: Return the parent of this node, or an empty tree if this node is the root.

In the implementation we have seen, the empty tree is represented by the special value `null`. `left` and `right` are performed by accessing the instance variables of the node. We have not implemented `parent` yet (you might think about how to do it).

There is another implementation of trees that uses arrays and indices instead of objects and references. To see how it works, we will start by looking at a hybrid implementation that uses both arrays and objects.

This figure shows a tree like the ones we have been looking at, although it is laid out sideways, with the root at the left and the leaves on the right. At the bottom there is an array of references that refer to the objects in the trees.
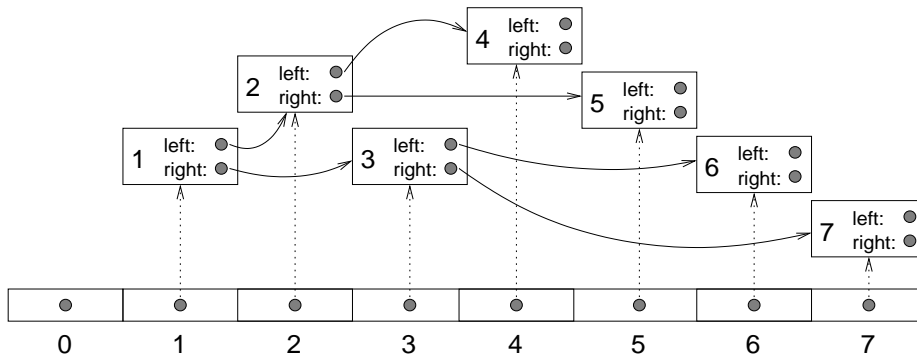
In this tree the cargo of each node is the same as the array index of the node, but of course that is not true in general. You might notice that array index 1 refers to the root node and array index 0 is empty. The reason for that will become clear soon.

So now we have a tree where each node has a unique index. Furthermore, the indices have been assigned to the nodes according to a deliberate pattern, in order to achieve the following results:

1. The left child of the node with index i has index $2i$.

2. The right child of the node with index i has index $2i + 1$.

3. The parent of the node with index i has index $i/2$ (rounded down).

Using these formulas, we can implement `left`, `right` and `parent` just by
doing arithmetic; we don't have to use the references at all!

Since we don't use the references, we can get rid of them, which means that
what used to be a tree node is now just cargo and nothing else. That means we
can implement the tree as an array of cargo objects; we don't need tree nodes
at all.

Here's what one implementation looks like:

```
public class Tree {
    Object[] array;

    public Tree () {
        array = new Object [128];
    }
```

No surprises so far. The instance variable is an array of `Objects`. The con-
structor initializes this array with an arbitrary initial size (we can always resize
it later).

To check whether a tree is empty, we check whether the root node is `null`.
Again, the root node is located at index 1.

```
    public boolean empty () {
        return (array[1] == null);
    }
```

The implementation of `left`, `right` and `parent` is just arithmetic:

```
    public int left (int i) {  return 2*i;  }
    public int right (int i) {  return 2*i + 1;  }
    public int parent (int i) {  return i/2;  }
```

Only one problem remanins. The node "references" we have are not really
references; they are integer indices. To access the cargo itself, we have to get or
set an element of the array. For that kind of operation, it is often a good idea to
provide methods that perform simple error checking before accessing the data
structure.

```
    public Object getCargo (int i) {
        if (i < 0 || i >= array.length) return null;
        return array[i];
    }

    public void setCargo (int i, Object obj) {
        if (i < 0 || i >= array.length) return;
        array[i] = obj;
    }
```

Methods like this are often called **accessor methods** because they provide
access to a data structure (the ability to get and set elements) without letting
the client see the details of the implementation.

Finally we are ready to build a tree. In another class (the client), we would
write

```
Tree tree = new Tree ();
int root = 1;
tree.setCargo (root, "cargo for root");
```

The constructor builds an empty tree. In this case we assume that the client
knows that the index of the root is 1 although it would be preferable for the
tree implementation to provide that information. Anyway, invoking `setCargo`
puts the string `"cargo for root"` into the root node.

To add children to the root node:

```
tree.setCargo (tree.left (root), "cargo for left");
tree.setCargo (tree.right (root), "cargo for right");
```

In the tree class we could provide a method that prints the contents of the tree
in preorder.

```
public void printPreorder (int i) {
    if (getNode (i) == null) return;
    System.out.println (getNode (i));
    printPreorder (left (i));
    printPreorder (right (i));
}
```

We invoke this method from the client by passing the root as a parameter.

```
tree.print (root);
```

The output is

```
cargo for root
cargo for left
cargo for right
```

This implementation provides the basic operations required to be a tree, but it
leaves a lot to be desired. As I pointed out, we expect the client to have a lot
of information about the implementation, and the interface the client sees, with
indices and all, is not very pretty.

Also, we have the usual problem with array implementations, which is that
the initial size of the array is arbitrary and it might have to be resized. This
last problem can be solved by replacing the array with a `Vector`.

17.10 The Vector class

The Vector is a built-in Java class in the java.util package. It is an implementation of an array of Objects, with the added feature that it can resize itself automatically, so we don't have to.

The Vector class provides methods named get and set that are similar to the getCargo and setCargo methods we wrote for the Tree class. You should review the other Vector operations by consulting the online documentation.

Before using the Vector class, you should understand a few concepts. Every Vector has a capacity, which is the amount of space that has been allocated to store values, and a size, which is the number of values that are actually in the vector.

The following figure is a simple diagram of a Vector that contains three elements, but it has a capacity of seven.
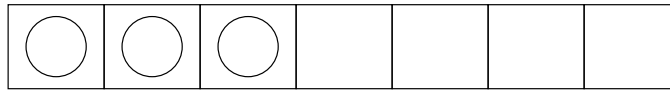
In general, it is the responsibility of the client code to make sure that the vector has sufficient *size* before invoking set or get. If you try to access an element that does not exist (in this case the elements with indices 3 through 6), you will get an ArrayIndexOutOfBounds exception.

The Vector methods use the add and insert automatically increase the size of the Vector, but set does not. The resize method adds null elements to the end of the Vector to get to the given size.

Most of the time the client doesn't have to worry about capacity. Whenever the size of the Vector changes, the capacity is updated automatically. For performance reasons, some applications might want to take control of this function, which is why there are additional methods for increasing and decreasing capacity.

Because the client code has no access to the implementation of a vector, it is not clear how we should traverse one. Of course, one possibility is to use a loop variable as an index into the vector:

```
for (int i=0; i<v.size(); i++) {
    System.out.println (v.get(i));
}
```

There's nothing wrong with that, but there is another way that serves to demonstrate the Iterator class. Vectors provide a method named iterator that returns an Iterator object that makes it possible to traverse the vector.

17.11 The Iterator class

Iterator is an abstract class in the java.util package. It specifies three methods:

`hasNext:` Does this iteration have more elements?

`next:` Return the next element, or throw an exception if there is none.

`remove:` Remove from the collection the last element that was returned.

The following example uses an iterator to traverse and print the elements of a vector.

```
Iterator iterator = vector.iterator ();
while (iterator.hasNext ()) {
    System.out.println (iterator.next ());
}
```

Once the `Iterator` is created, it is a separate object from the original `Vector`. Subsequent changes in the `Vector` are not reflected in the `Iterator`. In fact, if you modify the `Vector` after creating an `Iterator`, the `Iterator` becomes invalid. If you access the `Iterator` again, it will cause a `ConcurrentModification` exception.

In a previous section we used the `Visitable` abstract class to allow a client to traverse a data structure without knowing the details of its implementation. Iterators provide another way to do the same thing. In the first case, the provider performs the iteration and invokes client code to "visit" each element. In the second case the provider gives the client an object that it can use to select elements one at a time (albeit in an order controlled by the provider).

As an exercise, write a concrete class named `PreIterator` that implements the `Iterator` interface, and write a method named `preorderIterator` for the `Tree` class that returns a `PreIterator` that selects the elements of the `Tree` in preorder.

17.12   Glossary

binary tree: A tree in which each node refers to 0, 1, or 2 dependent nodes.

root: The top-most node in a tree, to which no other nodes refer.

leaf: A bottom-most node in a tree, which refers to no other nodes.

parent: The node that refers to a given node.

child: One of the nodes referred to by a node.

level: The set of nodes equidistant from the root.

prefix notation: A way of writing a mathematical expression with each operator appearing before its operands.

preorder: A way to traverse a tree, visiting each node before its children.

postorder: A way to traverse a tree, visiting the children of each node before the node itself.

inorder: A way to traverse a tree, visiting the left subtree, then the root, then the right subtree.

class variable: A `static` variable declared outside of any method. It is accessible from any method.

binary operator: An operator that takes two operands.

object encapsulation: The design goal of keeping the implementations of two objects as separate as possible. Neither class should have to know the details of the implementation of the other.

method encapsulation: The design goal of keeping the interface of a method separate from the details of its implementation.

Chapter 18

Heap

18.1 The Heap

A heap is a special kind of tree that happens to be an efficient implementation of a priority queue. This figure shows the relationships among the data structures in this chapter.

PriorityQueue	
Heap	
tree	
linked implementation	array implementation
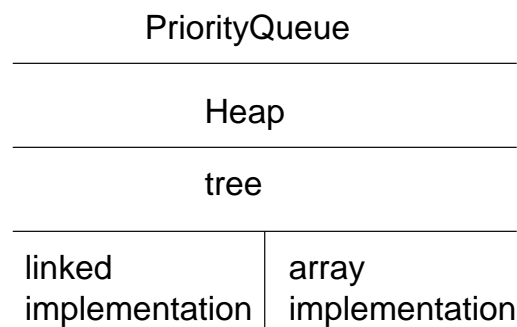
Ordinarily we try to maintain as much distance as possible between an ADT and its implementation, but in the case of the Heap, this barrier breaks down a little. The reason is that we are interested in the performance of the operations we implement. For each implementation there are some operations that are easy to implement and efficient, and others that are clumsy and slow.

It turns out that the array implementation of a tree works particularly well as an implementation of a Heap. The operations the array performs well are exactly the operations we need to implement a Heap.

To understand this relationship, we will proceed in a few steps. First, we need to develop ways of comparing the performance of various implementations. Next, we will look at the operations Heaps perform. Finally, we will compare the Heap implementation of a Priority Queue to the others (arrays and lists) and see why the Heap is considered particularly efficient.

18.2   Performance analysis

When we compare algorithms, we would like to have a way to tell when one
is faster than another, or takes less space, or uses less of some other resource.
It is hard to answer those questions in detail, because the time and space used
by an algorithm depend on the implementation of the algorithm, the particular
problem being solved, and the hardware the program runs on.

The objective of this section is to develop a way of talking about performance
that is independent of all of those things, and only depends on the algorithm
itself. To start, we will focus on run time; later we will talk about other re-
sources.

Our decisions are guided by a series of constraints:

1. First, the performance of an algorithm depends on the hardware it runs
   on, so we usually don't talk about run time in absolute terms like seconds.
   Instead, we usually count the number of abstract operations the algorithm
   performs.

2. Second, performance often depends on the particular problem we are try-
   ing to solve – some problems are easier than others. To compare algo-
   rithms, we usually focus on either the worst-case scenario or an average
   (or common) case.

3. Third, performance depends on the size of the problem (usually, but not
   always, the number of elements in a collection). We address this depen-
   dence explicitly by expressing run time as a function of problem size.

4. Finally, performance depends on details of the implementation like object
   allocation overhead and method invocation overhead. We usually ignore
   these details because they don't affect the rate at which the number of
   abstract operations increases with problem size.

To make this process more concrete, consider two algorithms we have already
seen for sorting an array of integers. The first is **selection sort**, which we saw
in Section **??**. Here is the pseudocode we used there.

```
selectionsort (array) {
    for (int i=0; i<array.length; i++) {
      // find the lowest item at or to the right of i
      // swap the ith item and the lowest item
    }
}
```

To perform the operations specified in the pseudocode, we wrote helper methods
named `findLowest` and `swap`. In pseudocode, `findLowest` looks like this

```
// find the index of the lowest item between
// i and the end of the array
```

```
findLowest (array, i) {
    // lowest contains the index of the lowest item so far
    lowest = i;
    for (int j=i+1; j<array.length; j++) {
      // compare the jth item to the lowest item so far
      // if the jth item is lower, replace lowest with j
    }
    return lowest;
}
```

And `swap` looks like this:

```
swap (i, j) {
    // store a reference to the ith card in temp
    // make the ith element of the array refer to the jth card
    // make the jth element of the array refer to temp
}
```

To analyze the performance of this algorithm, the first step is to decide what operations to count. Obviously, the program does a lot of things: it increments i, compares it to the length of the deck, it searches for the largest element of the array, etc. It is not obvious what the right thing is to count.

It turns out that a good choice is the number of times we compare two items. Many other choices would yield the same result in the end, but this is easy to do and we will find that it allows us to compare most easily with other sort algorithms.

The next step is to define the "problem size." In this case it is natural to choose the size of the array, which we'll call n.

Finally, we would like to derive an expression that tells us how many abstract operations (specifically, comparisons) we have to do, as a function of n.

We start by analyzing the helper methods. `swap` copies several references, but it doesn't perform any comparisons, so we ignore the time spent performing swaps. `findLowest` starts at `i` and traverses the array, comparing each item to `lowest`. The number of items we look at is $n - i$, so the total number of comparisons is $n - i - 1$.

Next we consider how many times `findLowest` gets invoked and what the value of i is each time. The last time it is invoked, i is $n - 2$ so the number of comparisons is 1. The previous iteration performs 2 comparisons, and so on. During the first iteration, i is 0 and the number of comparisons is $n - 1$.

So the total number of comparisons is $1 + 2 + \cdots + n - 1$. This sum is equal to $n^2/2 - n/2$. To describe this algorithm, we would typically ignore the lower order term $(n/2)$ and say that the total amount of work is proportional to n^2. Since the leading order term is quadratic, we might also say that this algorithm is **quadratic time**.
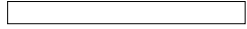
18.3 Analysis of mergesort

In Section **??** I claimed that mergesort takes time that is proportional to $n \log n$, but I didn't explain how or why. Now I will.

Again, we start by looking at pseudocode for the algorithm. For mergesort, it's

```
mergeSort (array) {
  // find the midpoint of the array
  // divide the array into two halves
  // sort the halves recursively
  // merge the two halves and return the result
}
```

At each level of the recursion, we split the array in half, make two recursive calls, and then merge the halves. Graphically, the process looks like this:

	# arrays	items per array	# merges	comparisons per merge	total work
	1	n	1	n–1	~n
	2	n/2	2	n/2–1	~n
	⋮	⋮	⋮	⋮	⋮
	n/2	2	n/2	2–1	~n
	n	1	0	0	

Each line in the diagram is a level of the recursion. At the top, a single array divides into two halves. At the bottom, n arrays (with one element each) are merged into $n/2$ arrays (with 2 elements each).

The first two columns of the table show the number of arrays at each level and the number of items in each array. The third column shows the number of merges that take place at each level of recursion. The next column is the one that takes the most thought: it shows the number of comparisons each merge performs.

If you look at the pseudocode (or your implementation) of merge, you should convince yourself that in the worst case it takes $m - 1$ comparisons, where m is the total number items being merged.

The next step is to multiply the number of merges at each level by the amount of work (comparisons) per merge. The result is the total work at each level. At this point we take advantage of a small trick. We know that in the end we are only interested in the leading-order term in the result, so we can go ahead and ignore the -1 term in the comparisons per merge. If we do that, then the total work at each level is simply n.

Next we need to know the number of levels as a function of n. Well, we start with an array of n items and divide it in half until it gets to 1. That's the same as starting at 1 and multiplying by 2 until we get to n. In other words,

we want to know how many times we have to multiply 2 by itself before we get to n. The answer is that the number of levels, l, is the logarithm, base 2, of n.

Finally, we multiply the amount of work per level, n, by the number of levels, $\log_2 n$ to get $n \log_2 n$, as promised. There isn't a good name for this functional form; most of the time people just say, "en log en."

It might not be obvious at first that $n \log_2 n$ is better than n^2, but for large values of n, it is. As an exercise, write a program that prints $n \log_2 n$ and n^2 for a range of values of n.

18.4 Overhead

Performance analysis takes a lot of handwaving. First we ignored most of the operations the program performs and counted only comparisons. Then we decided to consider only worst case performance. During the analysis we took the liberty of rounding a few things off, and when we finished, we casually discarded the lower-order terms.

When we interpret the results of this analysis, we have to keep all this handwaving in mind. Because mergesort is $n \log_2 n$, we consider it a better algorithm than selection sort, but that doesn't mean that mergesort is *always* faster. It just means that eventually, if we sort bigger and bigger arrays, mergesort will win.

How long that takes depends on the details of the implementation, including the additional work, besides the comparisons we counted, that each algorithm performs. This extra work is sometimes called **overhead**. It doesn't affect the performance analysis, but it does affect the run time of the algorithm.

For example, our implementation of mergesort actually allocates subarrays before making the recursive calls and then lets them get garbage collected after they are merged. Looking again at the diagram of mergesort, we can see that the total amount of space that gets allocated is proportional to $n \log_2 n$, and the total number of objects that get allocated is about $2n$. All that allocating takes time.

Even so, it is most often true that a bad implementation of a good algorithm is better than a good implementation of a bad algorithm. The reason is that for large values of n the good algorithm is better and for small values of n it doesn't matter because both algorithms are good enough.

As an exercise, write a program that prints values of $1000n \log_2 n$ and n^2 for a range of values of n. For what value of n are they equal?

18.5 Priority Queue implementations

In Chapter **??** we looked at an implementation of a Priority Queue based on an array. The items in the array are unsorted, so it is easy to add a new item (at the end), but harder to remove an item, because we have to search for the item with the highest priority.

An alternative is an implementation based on a sorted list. In this case when we insert a new item we traverse the list and put the new item in the right spot. This implementation takes advantage of a property of lists, which is that it is easy to insert a new node into the middle. Similarly, removing the item with the highest priority is easy, provided that we keep it at the beginning of the list.

Performance analysis of these operations is straightforward. Adding an item to the end of an array or removing a node from the beginning of a list takes the same amount of time regardless of the number of items. So both operations are constant time.

Any time we traverse an array or list, performing a constant-time operation on each element, the run time is proportional to the number of items. Thus, removing something from the array and adding something to the list are both linear time.

So how long does it take to insert and then remove n items from a Priority Queue? For the array implementation, n insertions takes time proportional to n, but the removals take longer. The first removal has to traverse all n items; the second has to traverse $n - 1$, and so on, until the last removal, which only has to look at 1 item. Thus, the total time is $1 + 2 + \cdots + n$, which is (still) $n^2/2 - n/2$. So the total for the insertions and the removals is the sum of a linear function and a quadratic function. The leading term of the result is quadratic.

The analysis of the list implementation is similar. The first insertion doesn't require any traversal, but after that we have to traverse at least part of the list each time we insert a new item. In general we don't know how much of the list we will have to traverse, since it depends on the data and what order they are inserted, but we can assume that on average we have to traverse half of the list. Unfortunately, even traversing half of the list is still a linear operation.

So, once again, to insert and remove n items takes time proportional to n^2. Thus, based on this analysis we cannot say which implementation is better; both the array and the list yield quadratic run times.

If we implement a Priority Queue using a heap, we can perform both insertions and removals in time proportional to $log n$. Thus the total time for n items is $n \log n$, which is better than n^2. That's why, at the beginning of the chapter, I said that a heap is a particularly efficient implementation of a Priority Queue.

18.6   Definition of a Heap

A heap is a special kind of tree. It has two properties that are not generally true for other trees:

completeness: The tree is complete, which means that nodes are added from top to bottom, left to right, without leaving any spaces.

heapness: The item in the tree with the highest priority is at the top of the tree, and the same is true for every subtree.

Both of these properties bear a little explaining. This figure shows a number of trees that are considered complete or not complete:

Complete trees   Not complete trees
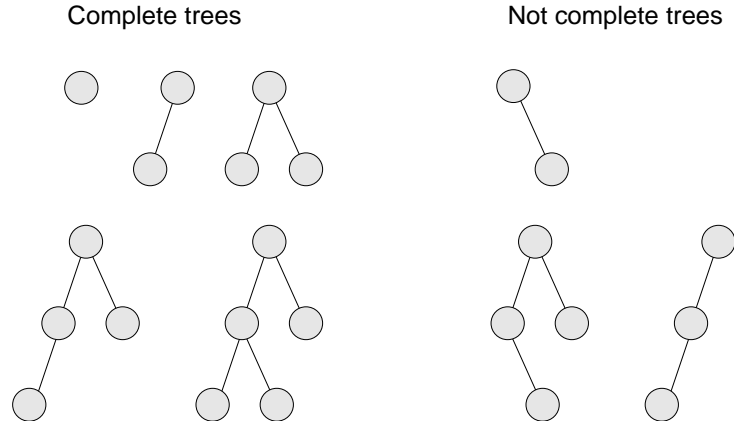
An empty tree is also considered complete. We can define completeness more rigorously by comparing the height of the subtrees. Recall that the **height** of a tree is the number of levels.

Starting at the root, if the tree is complete, then the height of the left subtree and the height of the right subtree should be equal, or the left subtree may be taller by one. In any other case, the tree cannot be complete.

Furthermore, if the tree is complete, then the height relationship between the subtrees has to be true for every node in the tree.

It is natural to write these rules as a recursive method:

```
public static boolean isComplete (Tree tree) {
    // the null tree is complete
    if (tree == null) return true;

    int leftHeight = height (tree.left);
    int rightHeight = height (tree.right);
    int diff = leftHeight - rightHeight)

    // check the root node
    if (diff < 0 || diff > 1) return false;

    // check the children
    if (!isComplete (tree.left)) return false;
    return isComplete (tree.right);
}
```

For this example I used the linked implementation of a tree. As an exercise, write the same method for the array implementation. Also as an exercise, write the `height` method. The height of a `null` tree is 0 and the height of a leaf node is 1.

The **heap property** is similarly recursive. In order for a tree to be a heap, the largest value in the tree has to be at the root, *and* the same has to be true

for each subtree. As another exercise, write a method that checks whether a
tree has the heap property.

18.7   Heap remove

It might seem odd that we are going to remove things from the heap before we
insert any, but I think removal is easier to explain.

At first glance, we might think that removing an item from the heap is a
constant time operation, since the item with the highest priority is always at
the root. The problem is that once we remove the root node, we are left with
something that is no longer a heap. Before we can return the result, we have to
restore the heap property. We call this operation `reheapify`.

The situation is shown in the following figure:
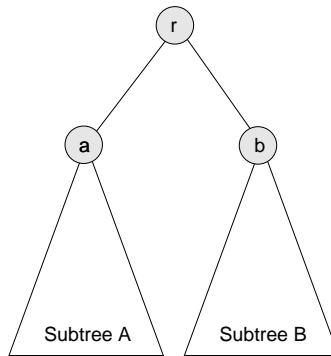
The root node has priority `r` and two subtrees, A and B. The value at the
root of Subtree A is `a` and the value at the root of Subtree B is `b`.

We assume that before we remove `r` from the tree, the tree is a heap. That
implies that `r` is the largest value in the heap and that `a` and `b` are the largest
values in their respective subtrees.

Once we remove `r`, we have to make the resulting tree a heap again. In other
words we need to make sure it has the properties of completeness and heapness.

The best way to ensure completeness is to remove the bottom-most, right-
most node, which we'll call `c` and put its value at the root. In a general tree
implementation, we would have to traverse the tree to find this node, but in the
array implementation, we can find it in constant time because it is always the
last (non-null) element of the array.

Of course, the chances are that the last value is not the highest, so putting
it at the root breaks the heapness property. Fortunately it is easy to restore.
We know that the largest value in the heap is either `a` or `b`. Therefore we can
select whichever is larger and swap it with the value at the root.

Arbitrarily, let's say that `b` is larger. Since we know it is the highest value
left in the heap, we can put it at the root and put `c` at the top of Subtree B.
Now the situation looks like this:
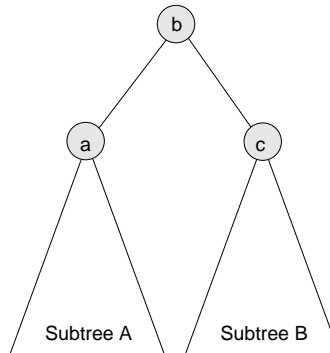
Again, `c` is the value we copied from the last entry in the array and `b` is the highest value left in the heap. Since we haven't changed Subtree A at all, we know that it is still a heap. The only problem is that we don't know if Subtree B is a heap, since we just stuck a (probably low) value at its root.

Wouldn't it be nice if we had a method that could `reheapify` Subtree B? Wait... we do!

18.8   Heap `insert`

Inserting a new item in a heap is a similar operation, except that instead of trickling a value down from the top, we trickle it up from the bottom.

Again, to guarantee completeness, we add the new element at the bottom-most, rightmost position in the tree, which is the next available space in the array.

Then to restore the heap property, we compare the new value with its neighbors. The situation looks like this:
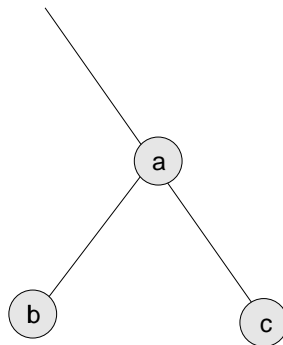
The new value is `c`. We can restore the heap property of this subtree by comparing `c` to `a`. If `c` is smaller, then the heap property is satisfied. If `c` is larger, then we swap `c` and `a`. The swap satisfies the heap property because we know that `c` must also be bigger than `b`, because `c > a` and `a > b`.

Now that the subtree is reheapified, we can work our way up the tree until we reach the root.

18.9 Performance of heaps

For both insert and remove, we perform a constant time operation to do the actual insertion and removal, but then we have to reheapify the tree. In one case we start at the root and work our way down, comparing items and then recursively reheapifying one of the subtrees. In the other case we start at a leaf and work our way up, again comparing elements at each level of the tree.

As usual, there are several operations we might want to count, like comparisons and swaps. Either choice would work; the real issue is the number of levels of the tree we examine and how much work we do at each level. In both cases we keep examining levels of the tree until we restore the heap property, which means we might only visit one, or in the worst case we might have to visit them all. Let's consider the worst case.

At each level, we perform only constant time operations like comparisons and swaps. So the total amount of work is proportional to the number of levels in the tree, a.k.a. the height.

So we might say that these operations are linear with respect to the height of the tree, but the "problem size" we are interested in is not height, it's the number of items in the heap.

As a function of n, the height of the tree is $log_2 n$. This is not true for all trees, but it is true for complete trees. To see why, think of the number of nodes on each level of the tree. The first level contains 1, the second contains 2, the third contains 4, and so on. The ith level contains 2^i nodes, and the total number in all levels up to i is $2^i - 1$. In other words, $2^h = n$, which means that $h = log_2 n$.

Thus, both insertion and removal take **logarithmic** time. To insert and remove n items takes time proportional to $n \log_2 n$.

18.10 Heapsort

The result of the previous section suggests yet another algorithm for sorting. Given n items, we insert them into a Heap and then remove them. Because of the Heap semantics, they come out in order. We have already shown that this algorithm, which is called **heapsort**, takes time proportional to $n \log_2 n$, which is better than selection sort and the same as mergesort.

As the value of n gets large, we expect heapsort to be faster than selection sort, but performance analysis gives us no way to know whether it will be faster than mergesort. We would say that the two algorithms have the same **order of growth** because they grow with the same functional form. Another way to say the same thing is that they belong to the same **complexity class**.

Complexity classes are sometimes written in "big-O notation". For example, $\mathcal{O}(n^2)$, pronounced "oh of en squared" is the set of all functions that grow no faster than n^2 for large values of n. To say that an algorithm is $\mathcal{O}(n^2)$ is the same as saying that it is quadratic. The other complexity classes we have seen, in decreasing order of performance, are:

$\mathcal{O}(1)$	constant time
$\mathcal{O}(\log n)$	logarithmic
$\mathcal{O}(n)$	linear
$\mathcal{O}(n \log n)$	"en log en"
$\mathcal{O}(n^2)$	quadratic
$\mathcal{O}(2^n)$	exponential

So far none of the algorithms we have looked at are **exponential**. For large values of n, these algorithms quickly become impractical. Nevertheless, the phrase "exponential growth" appears frequently in even non-technical language. It is frequently misused so I wanted to include its technical meaning.

People often use "exponential" to describe any curve that is increasing and accelerating (that is, one that has positive slope and curvature). Of course, there are many other curves that fit this description, including quadratic functions (and higher-order polynomials) and even functions as undramatic as $n \log n$. Most of these curves do not have the (often detrimental) explosive behavior of exponentials.

As an exercise, compare the behavior of $1000n^2$ and 2^n as the value of n increases.

18.11 Glossary

selection sort: The simple sorting algorithm in Section **??**.

mergesort: A better sorting algorithm from Section **??**.

heapsort: Yet another sorting algorithm.

complexity class: A set of algorithms whose performance (usually run time) has the same order of growth.

order of growth: A set of functions with the same leading-order term, and therefore the same qualitative behavior for large values of n.

overhead: Additional time or resources consumed by a programming performing operations other than the abstract operations considered in performance analysis.

Chapter 19

Table

19.1 Arrays, Vectors and Tables

Arrays are a generally useful data structure, but they suffer from two important limitations:

- The size of the array does not depend on the number of items in it. If the array is too big, it wastes space. If it is too small it might cause an error, or we might have to write code to resize it.

- Although the array can contain any type of item, the indices of the array have to be integers. We cannot, for example, use a String to specify an element of an array.

In Section **??** we saw how the built-in `Vector` class solves the first problem. As the user adds items it expands automatically. It is also possible to shrink a `Vector` so that the capacity is the same as the current size.

But `Vectors` don't help with the second problem. The indices are still integers.

That's where the Table ADT comes in. The Table is a generalization of the `Vector` that can use any type as an index. These generalized indices are called **keys**.

Just as you would use an index to access a value in an array, you use a key to access a value in a Table. So each key is associated with a value, which is why Tables are sometimes called **associative arrays**.

A common example of a table is a dictionary, which is a table that associates words (the keys) with their definitions (the values). Because of this example Tables are also sometimes called Dictionaries. Also, the association of a particular key with a particular value is called an **entry**.

19.2   The Table ADT

Like the other ADTs we have looked at, Tables are defined by the set of operations they support:

constructor: Make a new, empty table.

`put:` Create an entry that associates a value with a key.

`get:` For a given key, find the corresponding value.

`containsKey:` Return `true` if there is an entry in the Table with the given Key.

`keys` : Return a collection that contains all the keys in the Table.

19.3   The built-in Hashtable

Java provides an implementation of the Table ADT called `Hashtable`. It is in the `java.util` package. Later in the chapter we'll see why it is called `Hashtable`.

To demonstrate the use of the `Hashtable` we'll write a short program that traverses a String and counts the number of times each word appears.

We'll create a new class called `WordCount` that will build the Table and then print its contents. Naturally, each `WordCount` object contains a `Hashtable`:

```
public class WordCount {
    Hashtable ht;

    public WordCount () {
        ht = new Hashtable ();
    }
}
```

The only public methods for `WordCount` are `processLine`, which takes a String and adds its words to the Table, and `print`, which prints the results at the end.

`processLine` breaks the String into words using a `StringTokenizer` and passes each word to `processWord`.

```
    public void processLine (String s) {
        StringTokenizer st = new StringTokenizer (s, " ,.");
        while (st.hasMoreTokens()) {
            String word = st.nextToken();
            processWord (word.toLowerCase ());
        }
    }
```

The interesting work is in `processWord`.

```
    public void processWord (String word) {
        if (ht.containsKey (word)) {
            Integer i = (Integer) ht.get (word);
            Integer j = new Integer (i.intValue() + 1);
            ht.put (word, j);
        } else {
            ht.put (word, new Integer (1));
        }
    }
```

If the word is already in the table, we `get` its counter, increment it, and `put` the new value. Otherwise, we just `put` a new entry in the table with the counter set to 1.

To print the entries in the table, we need to be able to traverse the keys in the table. Fortunately, the `Hashtable` implementation provides a method, `keys`, that returns an `Enumeration` object we can use. `Enumerations` are very similar to the `Iterators` we saw in Section **??**. Both are abstract classes in the `java.util` package; you should review the documentation of both. Here's how to use `keys` to print the contents of the `Hashtable`:

```
    public void print () {
        Enumeration enum = ht.keys ();
        while (enum.hasMoreElements ()) {
            String key = (String) enum.nextElement ();
            Integer value = (Integer) ht.get (key);
            System.out.println ("{ " + key + ", " + value + " }");
        }
    }
```

Each of the elements of the `Enumeration` is an `Object`, but since we know they are keys, we typecast them to be `Strings`. When we get the values from the Table, they are also `Objects`, but we know they are counters, so we typecast them to be `Integers`.

Finally, to count the words in a string:

```
        WordCount wc = new WordCount ();
        wc.processLine ("da doo ron ron ron, da doo ron ron");
        wc.print ();
```

The output is

```
{ ron, 5 }
{ doo, 2 }
{ da, 2 }
```

The elements of the `Enumeration` are not in any particular order. The only guarantee is that all the keys in the table will appear.

19.4   A Vector implementation

An easy way to implement the Table ADT is to use a `Vector` of entries, where
each entry is an object that contains a key and a value. These objects are called
key-value pairs.

A class definition for a `KeyValuePair` might look like this:

```
class KeyValuePair {
    Object key, value;

    public KeyValuePair (Object key, Object value) {
        this.key = key;
        this.value = value;
    }

    public String toString () {
        return "{ " + key + ", " + value + " }";
    }
}
```

Then the implementation of Table looks like this:

```
public class Table {
    Vector v;

    public Table () {
        v = new Vector ();
    }
}
```

To put a new entry in the table, we just `add` a new `KeyValuePair` to the `Vector`:

```
    public void put (Object key, Object value) {
        KeyValuePair pair = new KeyValuePair (key, value);
        v.add (pair);
    }
```

Then to look up a key in the Table we have to traverse the `Vector` and find a
`KeyValuePair` with a matching key:

```
    public Object get (Object key) {
        Iterator iterator = v.iterator ();
        while (iterator.hasNext ()) {
            KeyValuePair pair = (KeyValuePair) iterator.next ();
            if (key.equals (pair.key)) {
                return pair.value;
            }
```

```
        }
        return null;
    }
```

The idiom to traverse a `Vector` is the one we saw in Section **??**. When we compare keys, we use deep equality (the `equals` method) rather than shallow equality (the `==` operator). This allows the key class to specify the definition of equality. In our example, the keys are Strings, so it will use the built-in `equals` method in the `String` class.

For most of the built-in classes, the `equals` method implements deep equality. For some classes, though, it is not easy to define what that means. For example, see the documentation of `equals` for `Doubles`.

Because `equals` is an object method, this implementation of `get` does not work if `key` is null. We could handle null as a special case, or we could do what the build-in Hashtable does—simply declare that `null` is not a legal key.

Speaking of the built-in `Hashtable`, it's implementation of `put` is a bit different from ours. If there is already an entry in the table with the given key, `put` updates it (give it a new value), and returns the old value (or `null` if there was none. Here is an implementation of their version:

```
    public Object put (Object key, Object value) {
        Object result = get (key);
        if (result == null) {
            KeyValuePair pair = new KeyValuePair (key, value);
            v.add (pair);
        } else {
            update (key, value);
        }
        return result;
    }
```

The `update` method is not part of the Table ADT, so it is declared `private`. It traverses the vector until it finds the right `KeyValuePair` and then it updates the `value` field. Notice that we don't have to modify the `Vector` itself, just one of the objects it contains.

```
    private void update (Object key, Object value) {
        Iterator iterator = v.iterator ();
        while (iterator.hasNext ()) {
            KeyValuePair pair = (KeyValuePair) iterator.next ();
            if (key.equals (pair.key)) {
                pair.value = value;
                break;
            }
        }
    }
```

The only methods we haven't implemented are `containsKey` and `keys`. The `containsKey` method is almost identical to `get` except that it returns `true` or `false` instead of an object reference or `null`.

As an exercise, implement `keys` by building a `Vector` of keys and returning the `elements` of the vector. See the documentation of `elements` in the `Vector` class for more information.

19.5   The `List` abstract class

The `java.util` package defines an abstract class called `List` that specifies the set of operations a class has to implement in order to be considered (very abstractly) a list. This does not mean, of course, that every class that implements `List` has to be a linked list.

Not surprisingly, the built-in `LinkedList` class is a member of the `List` abstract class. Surprisingly, so is `Vector`.

The methods in the `List` definition include `add`, `get` and `iterator`. In fact, all the methods from the `Vector` class that we used to implement Table are defined in the `List` abstract class.

That means that instead of a `Vector`, we could have used any `List` class. In `Table.java` we can replace `Vector` with `LinkedList`, and the program still works!

This kind of type generality can be useful for tuning the performance of a program. You can write the program in terms of an abstract class like `List` and then test the program with several different implementations to see which yields the best performance.

19.6   Hash table implementation

The reason that the built-in implementation of the Table ADT is called `Hashtable` is that it uses a particularly efficient implementation of a Table called a hashtable.

Of course, the whole point of defining an ADT is that it allows us to use an implementation without knowing the details. So it is probably a bad thing that the people who wrote the Java library named this class according to its implementation rather than its ADT, but I suppose of all the bad things they did, this one is pretty small.

Anyhoo, you might be wondering what a hashtable is, and why I say it is particularly efficient. We'll start by analyzing the performance of the `List` implementation we just did.

Looking at the implementation of `put`, we see that there are two cases. If the key is not already in the table, then we only have to create a new key-value pair and `add` it to the `List`. Both of these are constant-time operations.

In the other case, we have to traverse the `List` to find the existing key-value pair. That's a linear time operation. For the same reason, `get` and `containsKey`

are also linear.

Although linear operations are often good enough, we can do better. It turns out that there is a way to implement the Table ADT so that both `put` and `get` are constant time operations!

The key is to realize that traversing a list takes time proportional to the length of the list. If we can put an upper bound on the length of the list, then we can put an upper bound on the traverse time, and anything with a fixed upper bound is considered constant time.

But how can we limit the length of the lists without limiting the number of items in the table? By increasing the number of lists. Instead of one long list, we'll keep many short lists.

As long as we know which list to search, we can put a bound on the amount of searching.

19.7 Hash Functions

And that's where hash functions come in. We need some way to look at a key and know, without searching, which list it will be in. We'll assume that the lists are in an array (or `Vector`) so we can refer to them by index.

The solution is to come up with some mapping—almost any mapping—between the key values and the indices of the lists. For every possible key there has to be a single index, but there might be many keys that map to the same index.

For example, imagine an array of 8 lists and a table made up of keys that are `Integers` and values that are `Strings`. It might be tempting to use the `intValue` of the `Integers` as indices, since they are the right type, but there are a whole lot of integers that do not fall between 0 and 7, which are the only legal indices.

The modulus operator provides a simple (in terms of code) and efficient (in terms of run time) way to map *all* the integers into the range $(0, 7)$. The expression

```
    key.intValue() % 8
```

is guaranteed to produce a value in the range from -7 to 7 (including both). If you take its absolute value (using `Math.abs`) you will get a legal index.

For other types, we can play similar games. For example, to convert a `Character` to an integer, we can use the built-in method `Character.getNumericValue` and for `Doubles` there is `intValue`.

For `Strings` we could get the numeric value of each character and add them up, or instead we might use a **shifted sum**. To calculate a shifted sum, alternate between adding new values to the accumulator and shifting the accumulator to the left. By "shift to the left" I mean "multiply by a constant."

To see how this works, take the list of numbers $1, 2, 3, 4, 5, 6$ and compute their shifted sum as follows. First, initialize the accumulator to 0. Then,

1. Multiply the accumulator by 10.

2. Add the next element of the list to the accumulator.

3. Repeat until the list is finished.

As an exercise, write a method that calculates the shifted sum of the numeric values of the characters in a `String` using a multiplier of 32.

For each type, we can come up with a function that takes values of that type and generates a corresponding integer value. These functions are called **hash functions**, because they often involve making a hash of the components of the object. The integer value for each object is called its **hash code**.

There is one other way we might generate a hash code for Java objects. Every Java object provides a method called `hashCode` that returns an integer that corresponds to that object. For the built-in types, the `hashCode` method is implemented so that if two objects contain the same data, they will have the same hash code (as in deep equality). The documentation of these methods explains what the hash function is. You should check them out.

For user-defined types, it is up to the implementor to provide an appropriate hash function. The default hash function, provided in the `Object` class, often uses the location of the object to generate a hash code, so its notion of "sameness" is shallow equality. Most often when we are searching a hash table for a key, shallow equality is not what we want.

Regardless of how the hash code is generated, the last step is to use modulus and absolute value to map the hash code into the range of legal indices.

19.8   Resizing a hash table

Let's review. A Hash table consists of an array (or Vector) of `List`s, where each `List` contains a small number of key-value pairs. To add a new entry to a table, we calculate the hash code of the new key and add the entry to the corresponding `List`.

To look up a key, we hash it again and search the corresponding list. If the lengths of the lists are bounded then the search time is bounded.

So how do we keep the lists short? Well, one goal is to keep them as balanced as possible, so that there are no very long lists at the same time that others are empty. This is not easy to do perfectly—it depends on how well we chose the hash function—but we can usually do a pretty good job.

Even with perfect balance, the average list length grows linearly with the number of entries, and we have to put a stop to that.

The solution is to keep track of the average number of entries per list, which is called the **load factor**; if the load factor gets too high, we have to resize the table.

To resize, we create a new table, usually twice as big as the original, take all the entries out of the old one, hash them again, and put them in the new table. Usually we can get away with using the same hash function; we just use a different value for the modulus operator.

19.9 Performance of resizing

How long does it take to resize the table? Clearly it is linear with the number of entries. That means that *most* of the time `put` takes constant time, but every once in a while —when we resize—it takes linear time.

At first that sounds bad. Doesn't that undermine my claim that we can perform `put` in constant time? Well, frankly, yes. But with a little wheedling, I can fix it.

Since some `put` operations take longer than others, let's figure out the *average* time for a put operation. The average is going to be c, the constant time for a simple `put`, plus an additional term of p, the percentage of the time I have to resize, times kn, the cost of resizing.

$$t(n) = c + p \cdot kn \tag{19.1}$$

I don't know what c and k are, but we can figure out what p is. Imagine that we have just resized the hash table by doubling its size. If there are n entries, then we can add an addition n entries before we have to resize again. So the percentage of the time we have to resize is $1/n$.

Plugging into the equation, we get

$$t(n) = c + 1/n \cdot kn = c + k \tag{19.2}$$

In other words, $t(n)$ is constant time!

19.10 Glossary

table: An ADT that defines operations on a collection of entries.

entry: An element in a table that contains a key-value pair.

key: An index, of any type, used to look up values in a table.

value: An element, of any type, stored in a table.

dictionary: Another name for a table.

associative array: Another name for a dictionary.

hash table: A particularly efficient implementation of a table.

hash function: A function that maps values of a certain type onto integers.

hash code: The integer value that corresponds to a given value.

shifted sum: A simple hash function often used for compounds objects like `Strings`.

load factor: The number of entries in a hashtable divided by the number of lists in the hashtable; i.e. the average number of entries per list.

Appendix A

The Slate Class

```
import java.awt.*;

public class Example {

    // demonstrate simple use of the Slate class

  public static void main (String[] args) {
    int width = 500;
    int height = 500;

    Slate slate = Slate.makeSlate (width, height);
    Graphics g = Slate.getGraphics (slate);

    g.setColor (Color.blue);
    draw (g, 0, 0, width, height);

    anim (slate, 0, 0, width, height);
  }

    // draw is taken from Section 4.14 of the book

  public static void draw (Graphics g, int x, int y, int width, int height) {
    if (height < 3) return;

    g.drawOval (x, y, width, height);

    draw (g, x, y+height/2, width/2, height/2);
    draw (g, x+width/2, y+height/2, width/2, height/2);
  }

    // anim demonstrates a simple animation
```

```java
  public static void anim (Slate slate, int x, int y, int width, int height) {
    Graphics g = slate.image.getGraphics ();
    g.setColor (Color.red);

    for (int i=-100; i<500; i+=8) {
      g.drawOval (i, 100, 100, 100);
      slate.repaint ();
      try {
        Thread.sleep(10);
      } catch (InterruptedException e) {
      }
    }
  }
}

class Slate extends Frame {

    // image is a buffer: when Slate users draw things, they
    // draw on the buffer.  When the Slate gets painted, we
    // copy the image onto the screen.
  Image image;

  public static Slate makeSlate (int width, int height) {
    Slate s = new Slate ();
    s.setSize (width, height);
    s.setBackground (Color.white);
    s.setVisible (true);
    s.image = s.createImage (width, height);
    return s;
  }

    // when a Slate user asks for a Graphics object, we give
    // them one from the off-screen buffer.

  public static Graphics getGraphics (Slate s) {
    return s.image.getGraphics ();
  }

    // normally update erases the screen and invokes paint, but
    // since we are overwriting the whole screen anyway, it is
    // slightly faster to override update and avoid clearing the
    // screen

  public void update (Graphics g) {
    paint (g);
```

```
    }

      // paint copies the off-screen buffer onto the screen

    public void paint (Graphics g) {
      g.drawImage (image, 0, 0, null);
    }
}
```

Index

abstract class, 181, 183, 186, 191, 197
    Comparable, 181
    defining, 192
    implementing, 184, 193
    List, 218
    Visitable, 192
abstract data type, *see* ADT
abstract parameter, 129, 130
Abstract Window Toolkit, *see* AWT
abstraction, 130
accessor method, 195
ADT, 161, 167, 170, 187, 201
    Priority Queue, 173, 180
    Queue, 173
    Stack, 162
    Table, 214
    Tree, 193
algorithm, 105, 106
algorithm analysis, 202, 204
aliasing, 90, 93, 123, 135
ambiguity, 6, 122
    fundamental theorem, 154
analysis
    Hashtable, 221
    hashtable, 219
    Heap, 210
    heapsort, 210
    mergesort, 204
    Priority Queue, 205
argument, 23, 28, 31
arithmetic
    char, 80
    complex, 140
    floating-point, 22, 104
    integer, 16
array, 107, 117, 213

    compared to object, 110
    copying, 108
    element, 108
    length, 110
    of Cards, 131
    of object, 124
    of String, 121
    resizing, 169
    traverse, 113
ArrayIndexOutOfBounds, 169, 197
assignment, 13, 19, 63
associative array, 213, 221
AWT, 85, 93, 146

big-O notation, 210
binary tree, 187, 198
bisection search, 127
body
    loop, 65
    method, 24, 192
boolean, 55, 56, 61
bounding box, 38, 47
bucket, 114
buffer
    circular, 177
bug, 4

Card, 119
cargo, 149, 158, 187
Cartesian coordinate, 39
char, 75, 80
charAt, 75
Chianti, 108, 152
child node, 188, 198
circular buffer, 177, 186
class, 26, 31, 106
    abstract, 181, 183, 191