Think Java

How to Think Like a Computer Scientist

Allen B. Downey and Chris Mayfield

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

- *Program development.* There are many strategies for writing programs, including bottom-up, top-down, and others. We demonstrate multiple program development techniques, allowing readers to choose methods that work best for them.
- *Multiple learning curves.* To write a program, you have to understand the algorithm, know the programming language, and be able to debug errors. We discuss these and other aspects throughout the book, and include an appendix that summarizes our advice.

Object-Oriented Programming

Some Java books introduce classes and objects immediately; others begin with procedural programming and transition to object-oriented more gradually.

Many of Java's object-oriented features are motivated by problems with previous languages, and their implementations are influenced by this history. Some of these features are hard to explain when people aren't familiar with the problems they solve.

We get to object-oriented programming as quickly as possible, limited by the requirement that we introduce concepts one at a time, as clearly as possible, in a way that allows readers to practice each idea in isolation before moving on. So it takes some time to get there.

But you can't write Java programs (even hello world) without encountering object-oriented features. In some cases we explain a feature briefly when it first appears, and then explain it more deeply later on.

This book is well suited to prepare students for the AP Computer Science A exam, which includes object-oriented design and implementation. (AP is a registered trademark of the College Board.) We introduce nearly every topic in the "AP Java subset" with a few exceptions. A mapping of *Think Java* section numbers to the current AP course description is available on our website: *http://thinkjava.org*.

Appendixes

The chapters of this book are meant to be read in order, because each one builds on the previous one. We also include three appendixes with material that can be read at any time:

Appendix A, Development Tools
: The steps for compiling, running, and debugging Java code depend on the details of the development environment and operating system. We avoided putting these details in the main text, because they can be distracting. Instead, we provide this appendix with a brief introduction to DrJava—an interactive development envi-

CHAPTER 1

The Way of the Program

The goal of this book is to teach you to think like a computer scientist. This way of thinking combines 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 trade-offs among alternatives. And 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**. It involves the ability to formulate problems, think creatively about solutions, and express solutions clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to develop problem solving skills. That's why this chapter is called, "The way of the program".

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

What Is Programming?

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. It can also be a symbolic computation, like searching and replacing text in a document or (strangely enough) compiling a program. The details look different in different languages, but a few basic instructions appear in just about every language.

that led to the results you see. Thinking about how to correct programs and improve their performance sometimes even leads to the discovery of new algorithms.

Programming Languages

The programming language you will learn is Java, which is a **high-level language**. Other high-level languages you may have heard of include Python, C and C++, Ruby, and JavaScript.

Before they can run, programs in high-level languages have to be translated into a **low-level language**, also called "machine language". This translation takes some time, which is a small disadvantage of high-level languages. But high-level languages have two advantages:

- It is *much* easier to program in a high-level language. Programs take less time to write, they are shorter and easier to read, and they are more likely to be correct.
- High-level languages are **portable**, meaning 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.

Two kinds of programs translate high-level languages into low-level languages: interpreters and compilers. An **interpreter** reads a high-level program and executes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations. Figure 1-1 shows the structure of an interpreter.

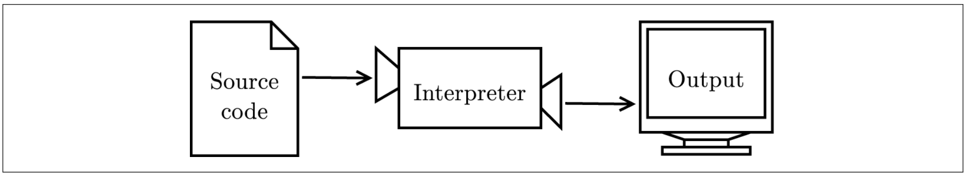

Figure 1-1. How interpreted languages are executed.

In contrast, a **compiler** reads the entire program and translates it completely before the program starts running. In this context, the high-level program is called the **source code**, and the translated program is called the **object code** or the **executable**. Once a program is compiled, you can execute it repeatedly without further translation. As a result, compiled programs often run faster than interpreted programs.

Java is *both* compiled and interpreted. Instead of translating programs directly into machine language, the Java compiler generates **byte code**. Similar to machine language, byte code is easy and fast to interpret. But it is also portable, so it is possible to compile a Java program on one machine, transfer the byte code to another machine,

Table 1-1. Common escape sequences

`\n`	newline
`\t`	tab
`\"`	double quote
`\\`	backslash

Formatting Code

In Java programs, some spaces are required. For example, you need at least one space between words, so this program is not legal:

```
publicclassGoodbye{

    publicstaticvoidmain(String[] args) {
        System.out.print("Goodbye, ");
        System.out.println("cruel world");
    }
}
```


But most other spaces are optional. For example, this program is legal:

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

The newlines are optional, too. So we could just write:

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

It still works, but the program is getting harder and harder to read. Newlines and spaces are important for organizing your program visually, making it easier to understand the program and find errors when they occur.

Many editors will automatically format source code with consistent indenting and line breaks. For example, in DrJava (see Appendix A) you can indent the code by selecting all text (*Ctrl+A*) and pressing the *Tab* key.

Organizations that do a lot of software development usually have strict guidelines on how to format source code. For example, Google publishes its Java coding standards for use in open-source projects: *http://google.github.io/styleguide/javaguide.html*.

If you spend some time learning this vocabulary, you will have an easier time reading the following chapters.

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

program:
A sequence of instructions that specifies how to perform tasks on a computer.

programming:
The application of problem solving to creating executable computer programs.

computer science:
The scientific and practical approach to computation and its applications.

algorithm:
A procedure or formula for solving a problem, with or without a computer.

bug:
An error in a program.

debugging:
The process of finding and removing errors.

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

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

portable:
The ability of a program to run on more than one kind of computer.

interpret:
To run a program in a high-level language by translating it one line at a time and immediately executing the corresponding instructions.

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.

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

This example declares two variables with type `String` and two with type `int`. When a variable name contains more than one word, like `firstName`, it is conventional to capitalize the first letter of each word except the first. Variable names are case-sensitive, so `firstName` is not the same as `firstname` or `FirstName`.

This example also demonstrates the syntax for declaring multiple variables with the same type on one line: `hour` and `minute` are both integers. Note that each declaration statement ends with a semicolon.

You can use any name you want for a variable. But there are about 50 reserved words, called **keywords**, that you are not allowed to use as variable names. These words include `public`, `class`, `static`, `void`, and `int`, which are used by the compiler to analyze the structure of the program.

You can find the complete list of keywords at *http://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html*, but you don't have to memorize them. Most programming editors provide “syntax highlighting”, which makes different parts of the program appear in different colors.

Assignment

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

```java
message = "Hello!";  // give message 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 illustrate 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 update its value.

As a general rule, a variable has to have the same type as the value you assign to it. For example, you cannot store a string in `minute` or an integer in `message`. We will see some examples that seem to break this rule, but we'll get to that later.

A common source of confusion is that some strings *look* like integers, but they are not. For example, `message` 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 integer `123`.

Rounding Errors

Most floating-point numbers are only *approximately* correct. Some numbers, like reasonably-sized integers, can be represented exactly. But repeating fractions, like 1/3, and irrational numbers, like π, cannot. To represent these numbers, computers have to round off to the nearest floating-point number.

The difference between the number we want and the floating-point number we get is called **rounding error**. For example, the following two statements should be equivalent:

```
System.out.println(0.1 * 10);
System.out.println(0.1 + 0.1 + 0.1 + 0.1 + 0.1
                 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1);
```

But on many machines, the output is:

```
1.0
0.9999999999999999
```

The problem is that `0.1`, which is a terminating fraction in base 10, is a repeating fraction in base 2. So its floating-point representation is only approximate. When we add up the approximations, the rounding errors accumulate.

For many applications, like computer graphics, encryption, statistical analysis, and multimedia rendering, floating-point arithmetic has benefits that outweigh the costs. But if you need *absolute* precision, use integers instead. For example, consider a bank account with a balance of $123.45:

```
double balance = 123.45;  // potential rounding error
```

In this example, balances will become inaccurate over time as the variable is used in arithmetic operations like deposits and withdrawals. The result would be angry customers and potential lawsuits. You can avoid the problem by representing the balance as an integer:

```
int balance = 12345;      // total number of cents
```

This solution works as long as the number of cents doesn't exceed the largest integer, which is about 2 billion.

Operators for Strings

In general, you cannot perform mathematical operations on strings, even if the strings look like numbers. The following expressions are illegal:

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

```
cm = inch * 2.54;
System.out.print(inch + " in = ");
System.out.println(cm + " cm");
```

This code works correctly, but it has a minor problem. If another programmer reads this code, they might wonder where 2.54 comes from. For the benefit of others (and yourself in the future), it would be better to assign this value to a variable with a meaningful name. We'll demonstrate in the next section.

Literals and Constants

A value that appears in a program, like 2.54 (or `" in ="`), is called a **literal**. In general, there's nothing wrong with literals. But when numbers like 2.54 appear in an expression with no explanation, they make code hard to read. And if the same value appears many times, and might have to change in the future, it makes code hard to maintain.

Values like that are sometimes called **magic numbers** (with the implication that being "magic" is not a good thing). A good practice is to assign magic numbers to variables with meaningful names, like this:


```
double cmPerInch = 2.54;
cm = inch * cmPerInch;
```

This version is easier to read and less error-prone, but it still has a problem. Variables can vary, but the number of centimeters in an inch does not. Once we assign a value to `cmPerInch`, it should never change. Java provides a language feature that enforces that rule, the keyword `final`.

```
final double CM_PER_INCH = 2.54;
```

Declaring that a variable is `final` means that it cannot be reassigned once it has been initialized. If you try, the compiler reports an error. Variables declared as `final` are called **constants**. By convention, names for constants are all uppercase, with the underscore character (_) between words.

Formatting Output

When you output a `double` using `print` or `println`, it displays up to 16 decimal places:

```
System.out.print(4.0 / 3.0);
```

The result is:

```
1.3333333333333333
```

Using division and modulus, we can convert to feet and inches like this:

```
quotient = 76 / 12;   // division
remainder = 76 % 12;  // modulus
```

The first line yields 6. The second line, which is pronounced "76 mod 12", yields 4. So 76 inches is 6 feet, 4 inches.

The modulus operator looks like a percent sign, but you might find it helpful to think of it as a division sign (÷) rotated to the left.

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`. You can use modulus to "extract" digits from a number: `x \% 10` yields the rightmost digit of `x`, and `x \% 100` yields the last two digits. Also, many encryption algorithms use the modulus operator extensively.

Putting It All Together

At this point, you have seen enough Java to write useful programs that solve everyday problems. You can (1) import Java library classes, (2) create a `Scanner`, (3) get input from the keyboard, (4) format output with `printf`, and (5) divide and mod integers. Now we will put everything together in a complete program:

```
import java.util.Scanner;

/**
 * Converts centimeters to feet and inches.
 */
public class Convert {

    public static void main(String[] args) {
        double cm;
        int feet, inches, remainder;
        final double CM_PER_INCH = 2.54;
        final int IN_PER_FOOT = 12;
        Scanner in = new Scanner(System.in);

        // prompt the user and get the value
        System.out.print("Exactly how many cm? ");
        cm = in.nextDouble();

        // convert and output the result
        inches = (int) (cm / CM_PER_INCH);
        feet = inches / IN_PER_FOOT;
        remainder = inches % IN_PER_FOOT;
        System.out.printf("%.2f cm = %d ft, %d in\n",
                          cm, feet, remainder);
    }
}
```

veniently, the `Math` class provides a constant double named `PI` that contains an approximation of π:

```
double degrees = 90;
double angle = degrees / 180.0 * Math.PI;
```

Notice that `PI` is in capital letters. Java does not recognize `Pi`, `pi`, or `pie`. Also, `PI` is the name of a variable, not a method, so it doesn't have parentheses. The same is true for the constant `Math.E`, which approximates Euler's number.

Converting to and from radians is a common operation, so the `Math` class provides methods that do it for you.

```
double radians = Math.toRadians(180.0);
double degrees = Math.toDegrees(Math.PI);
```

Another useful method is `round`, which rounds a floating-point value to the nearest integer and returns a `long`. A `long` is like an `int`, but bigger. More specifically, an `int` uses 32 bits; the largest value it can hold is $2^{31} - 1$, which is about 2 billion. A `long` uses 64 bits, so the largest value is $2^{63} - 1$, which is about 9 quintillion.

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

The result is 63 (rounded up from 62.8319).

Take a minute to read the documentation for these and other methods in the `Math` class. The easiest way to find documentation for Java classes is to do a web search for "Java" and the name of the class.

Composition Revisited

Just as with mathematical functions, Java methods can be *composed*. That means you can 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 / 2.0);
```

This statement divides `Math.PI` by two, adds the result to `angle`, and computes the cosine of the sum. You can also take the result of one method and pass it as an argument to another:

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

In Java, the `log` method always uses base *e*. So this statement finds the log base *e* of 10, and then raises *e* to that power. The result gets assigned to `x`.

Some math methods take more than one argument. For example, `Math.pow` takes two arguments and raises the first to the power of the second. This line of code assigns the value `1024.0` to the variable `x`:

```java
double x = Math.pow(2.0, 10.0);
```

When using `Math` methods, it is a common error to forget the `Math`. For example, if you try to invoke `pow(2.0, 10.0)`, you get an error message like:

```
File: Test.java  [line: 5]
Error: cannot find symbol
  symbol:   method pow(double,double)
  location: class Test
```

The message "cannot find symbol" is confusing, but the last line provides a useful hint. The compiler is looking for `pow` in the same class where it is used, which is `Test`. If you don't specify a class name, the compiler looks in the current class.

Adding New Methods

You have probably guessed by now that you can define more than one method in a class. Here's an example:


```java
public class NewLine {

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

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

The name of the class is `NewLine`. By convention, class names begin with a capital letter. `NewLine` contains two methods, `newLine` and `main`. Remember that Java is case-sensitive, so `NewLine` and `newLine` are not the same.

Method names should begin with a lowercase letter and use "camel case", which is a cute name for `jammingWordsTogetherLikeThis`. You can use any name you want for methods, except `main` or any of the Java keywords.

`newLine` and `main` are `public`, which means they can be invoked from other classes. They are both `static`, but we can't explain what that means yet. And they are both `void`, which means that they don't yield a result (unlike the `Math` methods, for example).

The parentheses after the method name contain a list of variables, called **parameters**, where the method stores its arguments. `main` has a single parameter, called `args`, which has type `String[]`. That means that whoever invokes `main` must provide an array of strings (we'll get to arrays in a later chapter).

```
File: Test.java  [line: 10]
Error: method printTwice in class Test cannot be applied
       to given types;
  required: java.lang.String
  found: int
  reason: actual argument int cannot be converted to
          java.lang.String by method invocation conversion
```

Sometimes Java can convert an argument from one type to another automatically. For example, `Math.sqrt` requires a `double`, but if you invoke `Math.sqrt(25)`, the integer value `25` is automatically converted to the floating-point value `25.0`. But in the case of `printTwice`, Java can't (or won't) convert the integer `17` to a `String`.

Parameters and other variables only exist inside their own methods. Inside `main`, there is no such thing as `s`. If you try to use it there, you'll get a compiler error. Similarly, inside `printTwice` there is no such thing as `argument`. That variable belongs to `main`.

Because variables only exist inside the methods where they are defined, they are often called **local variables**.

Multiple Parameters

Here is an example of a method that takes two parameters:

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

In the parameter list, it may be tempting to write:

```java
public static void printTime(int hour, minute) {
    ...
```

But that format (without the second `int`) is only legal for variable declarations. In parameter lists, you need to specify the type of each variable separately.

To invoke this method, we have to provide two integers as arguments:

```java
int hour = 11;
int minute = 59;
printTime(hour, minute);
```

A common error is to declare the types of the arguments, like this:

```java
int hour = 11;
int minute = 59;
printTime(int hour, int minute);  // syntax error
```

The class comment explains the purpose of the class. The method comment explains what the method does.

Notice that this example also includes an inline comment, beginning with `//`. In general, inline comments are short phrases that help explain complex parts of a program. They are intended for other programmers reading and maintaining the source code.

In contrast, Javadoc comments are longer, usually complete sentences. They explain what each method does, but they omit details about how the method works. And they are intended for people who will use the methods without looking at the source code.

Appropriate comments and documentation are essential for making source code readable. And remember that the person most likely to read your code in the future, and appreciate good documentation, is you.

Vocabulary

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

invoke:
: To cause a method to execute. Also known as "calling" a method.

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

flow of execution:
: The order in which Java executes methods and statements. It may not necessarily be from top to bottom, left to right.

parameter passing:
: The process of assigning an argument value to a parameter variable.

local variable:
: A variable declared inside a method. Local variables cannot be accessed from outside their method.

stack diagram:
: A graphical representation of the variables belonging to each method. The method calls are "stacked" from top to bottom, in the flow of execution.

frame:
: In a stack diagram, a representation of the variables and parameters for a method, along with their current values.

```
if (x > 0)
    System.out.println("x is positive");
    System.out.println("x is not zero");
```

This code is misleading because it's not indented correctly. Since there are no braces, only the first `println` is part of the `if` statement. Here is what the compiler actually sees:

```
if (x > 0) {
    System.out.println("x is positive");
}
    System.out.println("x is not zero");
```

As a result, the second `println` runs no matter what. Even experienced programmers make this mistake; search the web for Apple's "goto fail" bug.

Chaining and Nesting

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

```
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 braces lined up, you are less likely to make syntax errors.

In addition to chaining, you can also make complex decisions by **nesting** one conditional statement inside 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");
    }
}
```

The outer conditional has two branches. The first branch contains a `print` statement, and the second branch contains another conditional statement, which has two branches of its own. These two branches are also `print` statements, but they could have been conditional statements as well.

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

If there is no base case in a recursive method, or if the base case is never reached, the stack would grow forever, at least in theory. In practice, the size of the stack is limited; if you exceed the limit, you get a `StackOverflowError`.

For example, here is a recursive method without a base case:

```
public static void forever(String s) {
    System.out.println(s);
    forever(s);
}
```

This method displays the string until the stack overflows, at which point it throws an exception.

Binary Numbers

The `countdown` example has three parts: (1) it checks the base case, (2) displays something, and (3) makes a recursive call. What do you think happens if you reverse steps 2 and 3, making the recursive call *before* displaying?

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

The stack diagram is the same as before, and the method is still called *n* times. But now the `System.out.println` happens just before each recursive call returns. As a result, it counts up instead of down:

```
Blastoff!
1
2
3
```

This behavior comes in handy when it is easier to compute results in reverse order. For example, to convert a decimal integer into its **binary** representation, you repeatedly divide the number by two:

```
23 / 2 is 11 remainder 1
11 / 2 is  5 remainder 1
 5 / 2 is  2 remainder 1
 2 / 2 is  1 remainder 0
 1 / 2 is  0 remainder 1
```

Compared to void methods, value methods differ in two ways:

- They declare the type of the return value (the **return type**);
- They use at least one `return` statement to provide a **return value**.

Here's an example: `calculateArea` takes a `double` as a parameter and returns the area of a circle with that radius:

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

As usual, this method is `public` and `static`. But in the place where we are used to seeing `void`, we see `double`, which means that the return value from this method is a `double`.

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


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

On the other hand, **temporary variables** like `result` often make debugging easier, especially when you are stepping through code using an interactive debugger (see "Tracing with a Debugger" on page 207).

The type of the expression in the `return` statement must match the return type of the method. When you declare that the return type is `double`, you are making a promise that this method will eventually produce a `double` value. If you try to `return` with no expression, or an expression with the wrong type, the compiler will generate an error.

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

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

Since these `return` statements are in a conditional statement, only one will be executed. As soon as either of them executes, the method terminates without executing any more statements.

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

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

The first step is to consider what a `distance` method should look like in Java. In other words, what are the inputs (parameters) and what is the output (return value)? In this case, the two points are the parameters, and it is natural to represent them using four `double` values. The return value is the distance, which should also have type `double`.

Already we can write an outline for the method, which is sometimes called a **stub**. The stub includes the method signature and a `return` statement:

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

The return statement is a placeholder that is necessary for the program to compile. At this stage the program doesn't do anything useful, but it is good to compile it so we can find any syntax errors before we add more code.

It's usually a good idea to think about testing *before* you develop new methods; doing so can help you figure out how to implement them. To test the method, we can invoke it from `main` using sample values:

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

With these values, the horizontal distance is 3.0 and the vertical distance is 4.0. So the result should be 5.0, the hypotenuse of a 3-4-5 triangle. When you are testing a method, it is helpful to know the right answer.

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

The next step is to find the differences $x_2 - x_1$ and $y_2 - y_1$. We 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;
}
```

Conditional statements often invoke `boolean` methods and use the result as the condition:

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

Examples like this one almost read like English: "If is single digit z, print ... else print ...".

Javadoc Tags

In "Writing Documentation" on page 53, we discussed how to write documentation comments using `/**`. It's generally a good idea to document each class and method, so that other programmers can understand what they do without having to read the code.

To organize the documentation into sections, Javadoc supports optional **tags** that begin with the at sign (`@`). For example, we can use `@param` and `@return` to provide additional information about parameters and return values.

```
/**
 * Tests whether x is a single digit integer.
 *
 * @param x the integer to test
 * @return true if x has one digit, false otherwise
 */
public static boolean isSingleDigit(int x) {
```

Figure 6-1 shows part of the resulting HTML page generated by Javadoc. Notice the relationship between the source code and the documentation.

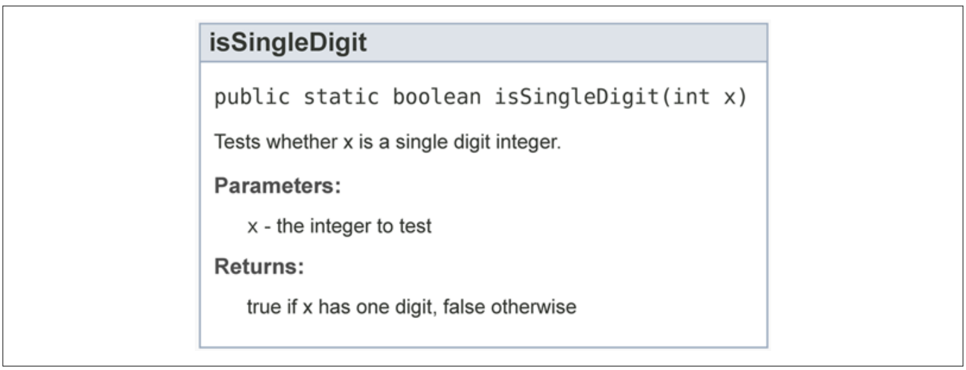

Figure 6-1. HTML documentation for `isSingleDigit`.

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

$$\sin\frac{\pi}{4} + \frac{\cos\frac{\pi}{4}}{2}$$

$$\log 10 + \log 20$$

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

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

Hint: The method for raising e to a power is `Math.exp`.

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

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

Exercise 6-5.

What is the output of the following program?

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

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

This type of flow is called a **loop**, because the last step loops back around to the first.

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 terminates when `n` is positive. But in general, it is not so easy to tell whether a loop terminates. For example, this loop continues until `n` is 1 (which makes the condition `false`):

```
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;
        }
    }
}
```


Each time through the loop, the program displays 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 and that the program will ever terminate. For some values of `n`, we can prove that it terminates. 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 when `n` is 16.

The hard question is whether this program terminates for *all* values of n. So far, no one has been able to prove it *or* disprove it! For more information, see *https://en.wikipedia.org/wiki/Collatz_conjecture*.

Generating Tables

Loops are good for generating and displaying tabular data. 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 of tables where you could look up values of various functions. Creating these tables by hand was slow and boring, and the results were often full of errors.

In words, the length of each row is the same as its row number. The result is a triangular multiplication table.

```
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
```

Generalization makes code more versatile, more likely to be reused, and sometimes easier to write.

The for Statement

The loops we have written so far have several elements in common. They start by initializing a variable, they have a condition that depends on that variable, and inside the loop they do something to update that variable. This type of loop is so common that there is another statement, the `for` loop, that expresses it more concisely.

For example, we could rewrite `printTable` like this:

```
public static void printTable(int rows) {
    for (int i = 1; i <= rows; i = i + 1) {
        printRow(i, rows);
    }
}
```

`for` loops have three components in parentheses, separated by semicolons: the initializer, the condition, and the update.

1. The *initializer* runs once at the very beginning of the loop.
2. The *condition* is checked each time through the loop. If it is `false`, the loop ends. Otherwise, the body of the loop is executed (again).
3. At the end of each iteration, the *update* runs, and we go back to step 2.

The `for` loop is often easier to read because it puts all the loop-related statements at the top of the loop.

There is one difference between `for` loops and `while` loops: if you declare a variable in the initializer, it only exists inside the `for` loop. For example, here is a version of `printRow` that uses a `for` loop:

```
public static void printRow(int n, int cols) {
    for (int i = 1; i <= cols; i = i + 1) {
        System.out.printf("%4d", n * i);
    }
    System.out.println(i);  // compiler error
}
```

Although `break` and `continue` statements give you more control of the loop execution, they can make code difficult to understand and debug. Use them sparingly.

Vocabulary

iteration:
Executing a sequence of statements repeatedly.

loop:
A statement that executes a sequence of statements repeatedly.

loop body:
The statements inside the loop.

infinite loop:
A loop whose condition is always true.

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

encapsulate:
To wrap a sequence of statements in a method.

generalize:
To replace something unnecessarily specific (like a constant value) with something appropriately general (like a variable or parameter).

loop variable:
A variable that is initialized, tested, and updated in order to control a loop.

increment:
Increase the value of a variable.

decrement:
Decrease the value of a variable.

pretest loop:
A loop that tests the condition before each iteration.

posttest loop:
A loop that tests the condition after each iteration.

Exercises

The code for this chapter is in the `ch07` directory of `ThinkJavaCode`. See “Using the Code Examples” on page xi for instructions on how to download the repository.

Exercises

The code for this chapter is in the `ch08` directory of `ThinkJavaCode`. See "Using the Code Examples" on page xi for instructions on how to download the repository. Before you start the exercises, we recommend that you compile and run the examples.

Exercise 8-1.

The goal of this exercise is to practice encapsulation with some of the examples in this chapter.

1. Starting with the code in "Array Traversal" on page 107, write a method called `powArray` that takes a `double` array, `a`, and returns a new array that contains the elements of `a` squared. Generalize it to take a second argument and raise the elements of `a` to the given power.
2. Starting with the code in "The Enhanced for Loop" on page 111, write a method called `histogram` that takes an `int` array of scores from 0 to (but not including) 100, and returns a histogram of 100 counters. Generalize it to take the number of counters as an argument.

Exercise 8-2.

The purpose of this exercise is to practice reading code and recognizing the traversal patterns in this chapter. The following methods are hard to read, because instead of using meaningful names for the variables and methods, they use names of fruit.

```
public static int banana(int[] a) {
    int kiwi = 1;
    int i = 0;
    while (i < a.length) {
        kiwi = kiwi * a[i];
        i++;
    }
    return kiwi;
}

public static int grapefruit(int[] a, int grape) {
    for (int i = 0; i < a.length; i++) {
        if (a[i] == grape) {
            return i;
        }
    }
    return -1;
}
```

```java
public static int pineapple(int[] a, int apple) {
    int pear = 0;
    for (int pine: a) {
        if (pine == apple) {
            pear++;
        }
    }
    return pear;
}
```

For each method, write one sentence that describes what the method does, without getting into the details of how it works. For each variable, identify the role it plays.

Exercise 8-3.

What is the output of the following program? Draw a stack diagram that shows the state of the program just before `mus` returns. Describe in a few words what `mus` does.


```java
public static int[] make(int n) {
    int[] a = new int[n];
    for (int i = 0; i < n; i++) {
        a[i] = i + 1;
    }
    return a;
}

public static void dub(int[] jub) {
    for (int i = 0; i < jub.length; i++) {
        jub[i] *= 2;
    }
}

public static int mus(int[] zoo) {
    int fus = 0;
    for (int i = 0; i < zoo.length; i++) {
        fus += zoo[i];
    }
    return fus;
}

public static void main(String[] args) {
    int[] bob = make(5);
    dub(bob);
    System.out.println(mus(bob));
}
```

Exercise 8-4.

Write a method called `indexOfMax` that takes an array of integers and returns the index of the largest element. Can you write this method using an enhanced `for` loop? Why or why not?

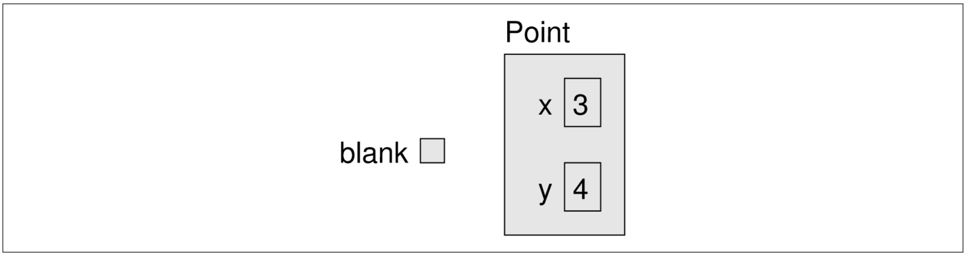

Figure 10-7. State diagram showing the effect of setting a variable to `null`.

As your program runs, the system automatically looks for stranded objects and reclaims them; then the space can be reused for new objects. This process is called **garbage collection**.

You don't have to do anything to make garbage collection happen, and in general you don't have to be aware of it. But in high-performance applications, you may notice a slight delay every now and then when Java reclaims space from discarded objects.

Class Diagrams

To summarize what we've learned so far, `Point` and `Rectangle` objects each have their own attributes and methods. Attributes are an object's *data*, and methods are an object's *code*. An object's class defines which attributes and methods it will have.

In practice, it's more convenient to look at high-level pictures than to examine the source code. **Unified Modeling Language** (UML) defines a standard way to summarize the design of a class.

As shown in Figure 10-8, a **class diagram** is divided into two sections. The top half lists the attributes, and the bottom half lists the methods. UML uses a language-independent format, so rather than showing `int x`, the diagram uses `x: int`.

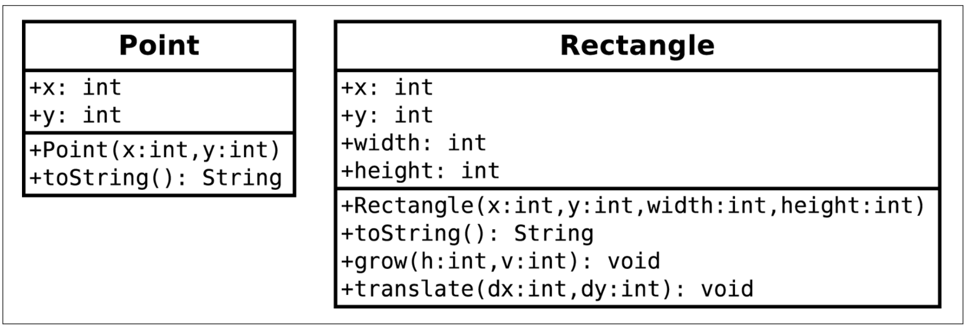

Figure 10-8. UML class diagrams for `Point` and `Rectangle`.

Now take a look at `Rectangle`'s `grow` and `translate` methods. There is more to them than you may have realized, but that doesn't limit your ability to use these methods in a program.

To summarize the whole chapter, objects encapsulate data and provide methods to access and modify the data directly. Object-oriented programming makes it possible to hide messy details so that you can more easily use and understand code that other people wrote.

Vocabulary

attribute:
One of the named data items that make up an object.

dot notation:
Use of the dot operator (`.`) to access an object's attributes or methods.

object-oriented:
A way of organizing code and data into objects, rather than independent methods.

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

UML:
Unified Modeling Language, a standard way to draw diagrams for software engineering.

class diagram:
An illustration of the attributes and methods for a class.

Exercises

The code for this chapter is in the `ch10` directory of `ThinkJavaCode`. See "Using the Code Examples" on page xi for instructions on how to download the repository. Before you start the exercises, we recommend that you compile and run the examples.

CHAPTER 11

Classes

Whenever you define a new class, you also create a new type with the same name. So way back in "The Hello World Program" on page 4, when we defined the class `Hello`, we created a type named `Hello`. We didn't declare any variables with type `Hello`, and we didn't use `new` to create a `Hello` object. It wouldn't have done much if we had—but we could have!

In this chapter, we will define classes that represent *useful* object types. We will also clarify the difference between classes and objects. Here are the most important ideas:

- Defining a **class** creates a new object type with the same name.
- Every object belongs to some object type; that is, it is an **instance** of some class.
- A class definition is like a template for objects: it specifies what attributes the objects have and what methods can operate on them.
- Think of a class like a blueprint for a house: you can use the same blueprint to build any number of houses.
- The methods that operate on an object type are defined in the class for that object.

The Time Class

One common reason to define a new class is to encapsulate related data in an object that can be treated as a single unit. That way, we can use objects as parameters and return values, rather than passing and returning multiple values. This design principle is called **data encapsulation**.

Here is an example constructor for the `Time` class:

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

This constructor does not take any arguments. Each line initializes an instance variable to zero (which in this example means midnight).

The name `this` is a keyword that refers to the object we are creating. You can use `this` the same way you use the name of any other object. For example, you can read and write the instance variables of `this`, and you can pass `this` as an argument to other methods. But you do not declare `this`, and you can't make an assignment to it.

A common error when writing constructors is to put a `return` statement at the end. Like void methods, constructors do not return values.

To create a `Time` object, you must use the `new` operator:

```
Time time = new Time();
```

When you invoke `new`, Java creates the object and calls your constructor to initialize the instance variables. When the constructor is done, `new` returns a reference to the new object. In this example, the reference gets assigned to the variable `time`, which has type `Time`. Figure 11-1 shows the result.

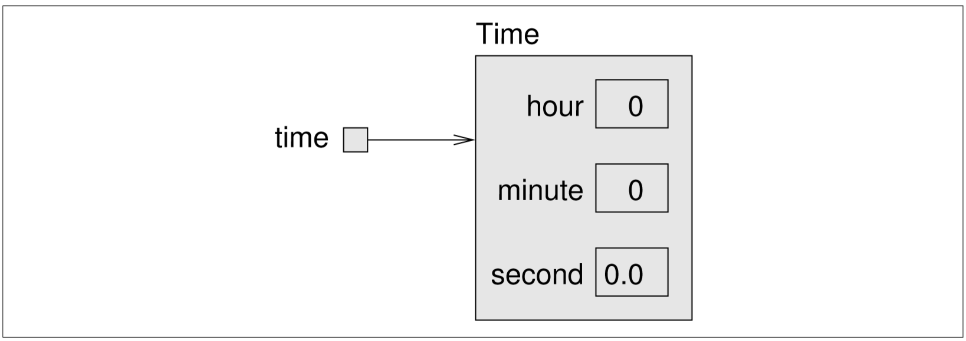

Figure 11-1. State diagram of a `Time` object.

By default it simply displays the type of the object and its address, but you can **override** this behavior by providing your own `toString` method. For example, here is a `toString` method for `Time`:

```
public String toString() {
    return String.format("%02d:%02d:%04.1f\n",
        this.hour, this.minute, this.second);
}
```

The definition does not have the keyword `static`, because it is not a static method. It is an **instance method**, so called because when you invoke it, you invoke it on an instance of the class (`Time` in this case). Instance methods are sometimes called "non-static"; you might see this term in an error message.

The body of the method is similar to `printTime` in the previous section, with two changes:

- Inside the method, we use `this` to refer to the current instance; that is, the object the method is invoked on.
- Instead of `printf`, it uses `String.format`, which returns a formatted `String` rather than displaying it.

Now you can call `toString` directly:

```
Time time = new Time(11, 59, 59.9);
String s = time.toString();
```

Or you can invoke it indirectly through `println`:

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

In this example, `this` in `toString` refers to the same object as `time`. The output is `11:59:59.9`.

The equals Method

We have seen two ways to check whether values are equal: the == operator and the `equals` method. With objects you can use either one, but they are not the same.

- The == operator checks whether objects are **identical**; that is, whether they are the same object.

  The `equals` method checks whether they are **equivalent**; that is, whether they have the same value.

The definition of identity is always the same, so the == operator always does the same thing. But the definition of equivalence is different for different objects, so objects can define their own `equals` methods.

CHAPTER 13
Objects of Arrays

In the previous chapter, we defined a class to represent cards and used an array of `Card` objects to represent a deck.

In this chapter, we take another step toward object-oriented programming by defining a class to represent a deck of cards. And we present algorithms for shuffling and sorting arrays.

The code for this chapter is in `Card.java` and `Deck.java`, which are in the directory `ch13` in the repository for this book. Instructions for downloading this code are in "Using the Code Examples" on page xi.

The Deck Class

The main idea of this chapter is to create a `Deck` class that encapsulates an array of `Card`s. The initial class definition looks like this:

```
public class Deck {
    private Card[] cards;

    public Deck(int n) {
        this.cards = new Card[n];
    }
}
```

The constructor initializes the instance variable with an array of `n` cards, but it doesn't create any card objects. Figure 13-1 shows what a `Deck` looks like with no cards.

Before you start the exercises, we recommend that you compile and run the examples.

Exercise 13-1.

You can learn more about the sorting algorithms in this chapter, and others, at *http://www.sorting-algorithms.com/*. This site includes explanations of the algorithms, animations that show how they work, and analysis of their efficiency.

Exercise 13-2.

The goal of this exercise is to implement the shuffling algorithm from this chapter.

1. In the repository for this book, you should find a file called `Deck.java` that contains the code in this chapter. Check that you can compile it in your environment.
2. Add a `Deck` method called `randomInt` that takes two integers, `low` and `high`, and returns a random integer between `low` and `high`, including both. You can use the `nextInt` provided by `java.util.Random`, which we saw in "Random Numbers" on page 108. But you should avoid creating a `Random` object every time `randomInt` is invoked.
3. Write a method called `swapCards` that takes two indexes and swaps the cards at the given locations.
4. Write a method called `shuffle` that uses the algorithm in "Shuffling Decks" on page 176.

Exercise 13-3.

The goal of this exercise is to implement the sorting algorithms from this chapter. Use the `Deck.java` file from the previous exercise (or create a new one from scratch).

1. Write a method called `indexLowest` that uses the `compareCard` method to find the lowest card in a given range of the deck (from `lowIndex` to `highIndex`, including both).
2. Write a method called `selectionSort` that implements the selection sort algorithm in "Selection Sort" on page 177.
3. Using the pseudocode in "Merge Sort" on page 178, write the method called `merge`. The best way to test it is to build and shuffle a deck. Then use `subdeck` to form two small subdecks, and use selection sort to sort them. Then you can pass the two halves to `merge` to see if it works.

CHAPTER 14
Objects of Objects

Now that we have classes that represent cards and decks, let's use them to make a game! *Crazy Eights* is a classic card game for two or more players. The main objective is to be the first player to get rid of all your cards. Here's how to play:

- Deal five or more cards to each player, and then deal one card face up to create the "discard pile". Place the remaining cards face down to create the "draw pile".
- Each player takes turns placing a single card on the discard pile. The card must match the rank or suit of the previously played card, or be an eight, which is a "wild card".
- When players don't have a matching card or an eight, they must draw new cards until they get one.
- If the draw pile ever runs out, the discard pile is shuffled (except the top card) and becomes the new draw pile.
- As soon as a player has no cards, the game ends and all other players score penalty points for their remaining cards. Eights are worth 20, face cards are worth 10, and all others are worth their rank.

You can read *https://en.wikipedia.org/wiki/Crazy_Eights* for more details, but we have enough to get started.

The code for this chapter is in the directory `ch14` in the repository for this book. Instructions for downloading this code are in "Using the Code Examples" on page xi.

`ArrayList` provides additional methods we aren't using here. You can read about them in the documentation, which you can find by doing a web search for "Java ArrayList".

Inheritance

At this point we have a class that represents a collection of cards. Next we'll use it to define `Deck` and `Hand`. Here is the complete definition of `Deck`:

```
public class Deck extends CardCollection {

    public Deck(String label) {
        super(label);

        for (int suit = 0; suit <= 3; suit++) {
            for (int rank = 1; rank <= 13; rank++) {
                cards.add(new Card(rank, suit));
            }
        }
    }
}
```


The first line uses the keyword `extends` to indicate that `Deck` extends the class `CardCollection`. That means a `Deck` object has the same instance variables and methods as a `CardCollection`. Another way to say the same thing is that `Deck` "inherits from" `CardCollection`. We could also say that `CardCollection` is a **superclass**, and `Deck` is one of its subclasses.

In Java, classes may only extend one superclass. Classes that do not specify a superclass with `extends` automatically inherit from `java.lang.Object`. So in this example, `Deck` extends `CardCollection`, which in turn extends `Object`. The `Object` class provides the default `equals` and `toString` methods, among other things.

Constructors are not inherited, but all other `public` attributes and methods are. The only additional method in `Deck`, at least for now, is a constructor. So you can create a `Deck` object like this:

```
Deck deck = new Deck("Deck");
```

The first line of the constructor uses something new, `super`, which is a keyword that refers to the superclass of the current class. When `super` is used like a method, as in this example, it invokes the constructor of the superclass.

So in this case, `super` invokes the `CardCollection` constructor, which initializes the attributes `label` and `cards`. When it returns, the `Deck` constructor resumes and populates the (empty) `ArrayList` with `Card` objects.

Exercise B-3.

In this exercise, you will draw “Moiré patterns” that seem to shift around as you move. For an explanation of what is going on, see *https://en.wikipedia.org/wiki/Moire_pattern*.

1. In the directory `app02` in the repository for this book, you’ll find a file named `Moire.java`. Open it and read the `paint` method. Draw a sketch of what you expect it to do. Now run it. Did you get what you expected?
2. Modify the program so that the space between the circles is larger or smaller. See what happens to the image.
3. Modify the program so that the circles are drawn in the center of the screen and concentric, as in Figure B-5 (left). The distance between the circles should be small enough that the Moiré interference is apparent.
4. Write a method named `radial` that draws a radial set of line segments as shown in Figure B-5 (right), but they should be close enough together to create a Moiré pattern.
5. Just about any kind of graphical pattern can generate Moiré-like interference patterns. Play around and see what you can create.

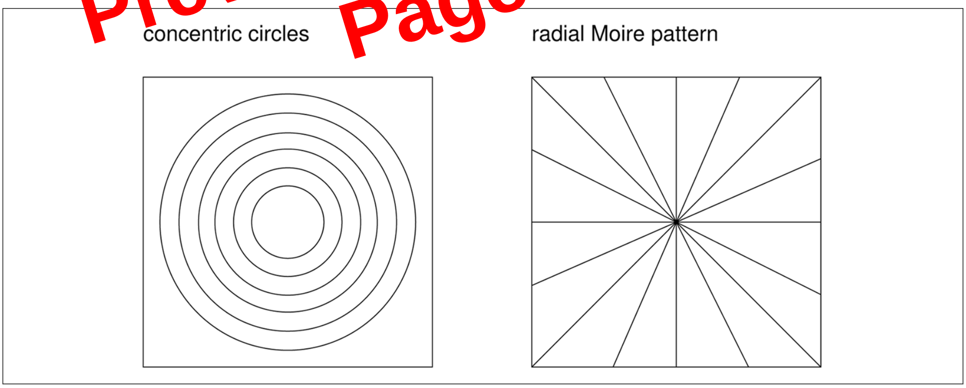

Figure B-5. Graphical patterns that can exhibit Moiré interference.

APPENDIX C

Debugging

Although there are debugging suggestions throughout the book, we thought it would be useful to organize them in an appendix. If you are having a hard time debugging, you might want to review this appendix from time to time.

The best debugging strategy depends on what kind of error you have:

- **Compile-time errors** indicate that there is something wrong with the syntax of the program. Example: omitting the semicolon at the end of a statement.
- **Run-time errors** are produced if something goes wrong while the program is running. Example: infinite recursion eventually causes a `StackOverflowError`.
- **Logic errors** cause the program to do the wrong thing. Example: an expression may not be evaluated in the order you expect.

The following sections are organized by error type; some techniques are useful for more than one type.

Compile-Time Errors

The best kind of debugging is the kind you don't have to do because you avoid making errors in the first place. Incremental development, which we presented in "Writing Methods" on page 73, can help. The key is to start with a working program and add small amounts of code at a time. When there is an error, you will have a pretty good idea where it is.

Nevertheless, you might find yourself in one of the following situations. For each situation, we have some suggestions about how to proceed.

The compiler is spewing error messages.

If the compiler reports 100 error messages, that doesn't mean there are 100 errors in your program. When the compiler encounters an error, it often gets thrown off track for a while. It tries to recover and pick up again after the first error, but sometimes it reports spurious errors.

Only the first error message is truly reliable. We suggest that you only fix one error at a time, and then recompile the program. You may find that one semicolon or brace "fixes" 100 errors.

I'm getting a weird compiler message, and it won't go away.

First of all, read the error message carefully. It may be written in terse jargon, but often there is a carefully hidden kernel of information.

If nothing else, the message will tell you where in the program the problem occurred. Actually, it tells you where the compiler was when it noticed a problem, which is not necessarily where the error is. Use the information the compiler gives you as a guideline, but if you don't see an error where the compiler is pointing, broaden the search.

Generally the error will be prior to the location of the error message, but there are cases where it will be somewhere else entirely. For example, if you get an error message at a method invocation, the actual error may be in the method definition itself.

If you don't find the error quickly, take a breath and look more broadly at the entire program. Make sure the program is indented properly; that makes it easier to spot syntax errors.

Now, start looking for common syntax errors:

1. Check that all parentheses and brackets are balanced and properly nested. All method definitions should be nested within a class definition. All program statements should be within a method definition.
2. Remember that uppercase letters are not the same as lowercase letters.
3. Check for semicolons at the end of statements (and no semicolons after squiggly braces).
4. Make sure that any strings in the code have matching quotation marks. Make sure that you use double quotes for strings and single quotes for characters.
5. For each assignment statement, make sure that the type on the left is the same as the type on the right. Make sure that the expression on the left is a variable name or something else that you can assign a value to (like an element of an array).

method foo", where `foo` is the name of the method. Now when you run the program, it displays a trace of each method as it is invoked.

You can also display the arguments each method receives. When you run the program, check whether the values are reasonable, and check for one of the most common errors—providing arguments in the wrong order.

When I run the program I get an exception.

When an exception occurs, Java displays a message that includes the name of the exception, the line of the program where the exception occurred, and a "stack trace". The stack trace includes the method that was running, the method that invoked it, the method that invoked that one, and so on.

The first step is to examine the place in the program where the error occurred and see if you can figure out what happened.

NullPointerException:

You tried to access an instance variable or invoke a method on an object that is currently `null`. You should figure out which variable is `null` and then figure out how it got to be that way.

Remember that when you declare a variable with an array type, its elements are initially `null` until you assign a value to them. For example, this code causes a `NullPointerException`:

```
int[] array = new Point[5];
System.out.println(array[0].x);
```

ArrayIndexOutOfBoundsException:

The index you are using to access an array is either negative or greater than `array.length - 1`. If you can find the site where the problem is, add a print statement immediately before it to display the value of the index and the length of the array. Is the array the right size? Is the index the right value?

Now work your way backwards through the program and see where the array and the index come from. Find the nearest assignment statement and see if it is doing the right thing. If either one is a parameter, go to the place where the method is invoked and see where the values are coming from.

StackOverflowError:

See "Infinite recursion" on page 221.

FileNotFoundException:

This means Java didn't find the file it was looking for. If you are using a project-based development environment like Eclipse, you might have to import the file