

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.

Exercise 4.1 Draw a stack diagram that shows the state of the program after `main` invokes `nLines` with the parameter `n=4`, just before the last instance of `nLines` returns.

4.10 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?

4.11 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 ever 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.

4.12 Exercises

Exercise 4.2 If you are given three sticks, you may or may not be able to arrange them in a triangle. For example, if one of the sticks is 12 inches long and the other two are one inch long, it is clear that you will not be able to get the short sticks to meet in the middle. For any three lengths, there is a simple test to see if it is possible to form a triangle:

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

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

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

Exercise 4.3 This exercise reviews the flow of execution through a program with multiple methods. Read the following code and answer the questions below.

```
public class Buzz {

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

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

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

- Write the number 1 next to the first *statement* of this program that will be executed. Be careful to distinguish things that are statements from things that are not.
- Write the number 2 next to the second statement, and so on until the end of the program. If a statement is executed more than once, it might end up with more than one number next to it.

- c. What is the value of the parameter `blimp` when `baffle` gets invoked?
- d. What is the output of this program?

Exercise 4.4 The first verse of the song “99 Bottles of Beer” is:

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

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

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

And then the song (finally) ends.

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

As you are developing your code, you will probably want to test it with a small number of verses, like “3 Bottles of Beer.”

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

Exercise 4.5 What is the output of the following program?

```
public class Narf {

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

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

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

    public static void ping (String strangStrung) {
```

```
        System.out.println ("any " + strangStrung + "more ");
    }
}
```

Exercise 4.6 Fermat’s Last Theorem says that there are no integers a , b , and c such that

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

except in the case when $n = 2$.

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

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

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

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

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.

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

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

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

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

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,

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

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

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:

$$\begin{aligned}0! &= 1 \\ n! &= n \cdot (n - 1)!\end{aligned}$$

(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 $n - 1$...

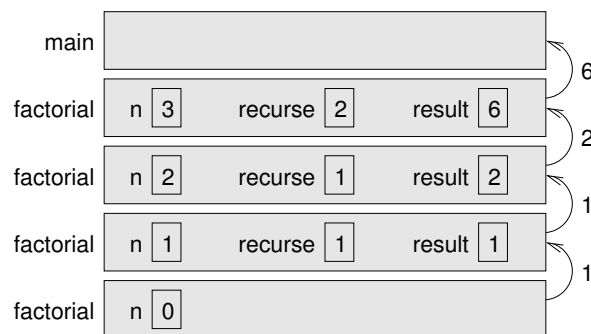
Since 0 *is* zero, we take the first branch and return the value 1 immediately without making any more recursive calls.

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 labyrinthine. 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 5.7 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:

$$fibonacci(0) = 1$$

$$\begin{aligned} \text{fibonacci}(1) &= 1 \\ \text{fibonacci}(n) &= \text{fibonacci}(n-1) + \text{fibonacci}(n-2); \end{aligned}$$

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.

5.12 Exercises

Exercise 5.1 Write a class method named `isDivisible` that takes two integers, `n` and `m` and that returns `true` if `n` is divisible by `m` and `false` otherwise.

Exercise 5.2 What is the output of the following program? The purpose of this exercise is to make sure you understand logical operators and the flow of execution through fruitful methods.

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

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

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

Exercise 5.3 The distance between two points (x_1, y_1) and (x_2, y_2) is

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

Please write a method named `distance` that takes four doubles as parameters—`x1`, `y1`, `x2` and `y2`—and that prints the distance between the points.

You should assume that there is a method named `sumSquares` that calculates and returns the sum of the squares of its arguments. For example:

```
double x = sumSquares (3.0, 4.0);
```

would assign the value 25.0 to `x`.

The point of this exercise is to write a new method that uses an existing one. You should write only one method: `distance`. You should not write `sumSquares` or `main` and you should not invoke `distance`.

Exercise 5.4 The point of this exercise is to practice the syntax of fruitful methods.

- Dig out your solution to Exercise 3.5 and make sure you can still compile and run it.
- Transform `multadd` into a fruitful method, so that instead of printing a result, it returns it.
- Everywhere in the program that `multadd` gets invoked, change the invocation so that it stores the result in a variable and/or prints the result.
- Transform `yikes` in the same way.

Exercise 5.5 The point of this exercise is to use a stack diagram to understand the execution of a recursive program.

```
public class Prod {

    public static void main (String[] args) {
        System.out.println (prod (1, 4));
    }

    public static int prod (int m, int n) {
        if (m == n) {
            return n;
        } else {
            int recurse = prod (m, n-1);
            int result = n * recurse;
            return result;
        }
    }
}
```

- Draw a stack diagram showing the state of the program just before the last instance of `prod` completes. What is the output of this program?
- Explain in a few words what `prod` does.
- Rewrite `prod` without using the temporary variables `recurse` and `result`.

Exercise 5.6 The purpose of this exercise is to translate a recursive definition into a Java method. The Ackerman function is defined for non-negative integers as follows:

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

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

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

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

Exercise 5.7

- a. Create a program called `Recurse.java` and type in the following methods:

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

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

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

- b. Write some code in `main` that tests each of these methods. Make sure they work, and make sure you understand what they do.
- c. Write a method called `printString` that takes a `String` as a parameter and that prints the letters of the `String`, one on each line. It should be a `void` method.
- d. Write a method called `printBackward` that does the same thing as `printString` but that prints the `String` backwards (one character per line).
- e. Write a method called `reverseString` that takes a `String` as a parameter and that returns a new `String` as a return value. The new `String` should contain the same letters as the parameter, but in reverse order. For example, the output of the following code

```
String backwards = reverseString ("Allen Downey");
System.out.println (backwards);
should be
yenwoD nella
```

Exercise 5.8

- a. Create a new program called `Sum.java`, and type in the following two methods.

```

public static int methOne (int m, int n) {
    if (m == n) {
        return n;
    } else {
        return m + methOne (m+1, n);
    }
}

public static int methTwo (int m, int n) {
    if (m == n) {
        return n;
    } else {
        return n * methTwo (m, n-1);
    }
}

```

- b. Write a few lines in `main` to test these methods. Invoke them a couple of times, with a few different values, and see what you get. By some combination of testing and examination of the code, figure out what these methods do, and give them more meaningful names. Add comments that describe their function abstractly.
- c. Add a `println` statement to the beginning of both methods so that they print their arguments each time they are invoked. This is a useful technique for debugging recursive programs, since it demonstrates the flow of execution.

Exercise 5.9 Write a recursive method called `power` that takes a double `x` and an integer `n` and that returns x^n . Hint: a recursive definition of this operation is `power (x, n) = x * power (x, n-1)`. Also, remember that anything raised to the zeroeth power is 1.

Exercise 5.10 (This exercise is based on page 44 of Ableson and Sussman's *Structure and Interpretation of Computer Programs*.)

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

The algorithm is based on the observation that, if r is the remainder when a is divided by b , then the common divisors of a and b are the same as the common divisors of b and r . Thus we can use the equation

$$\text{gcd}(a, b) = \text{gcd}(b, r)$$

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

$$\text{gcd}(36, 20) = \text{gcd}(20, 16) = \text{gcd}(16, 4) = \text{gcd}(4, 0) = 4$$

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

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

Chapter 6

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:

<code>int fred = 5;</code>	<code>fred</code> 5
<code>fred = 7;</code>	<code>fred</code> 5 7

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 alternative 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:

```
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:

$$\log_2 x = \log_e x / \log_e 2 \quad (6.1)$$

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 2.4, 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 4.3 and `isSingleDigit` in Section 5.7.

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