

Chapter 2

Methods

OBJECTIVES

- Learn how to call methods based on their specifications.
- Learn how to design and write methods.
- Learn how to execute a method call, using a “model of memory”.
- Study a methodology for writing method bodies called *stepwise refinement* or *top-down programming*.
- Learn how to conditionally execute a statement.
- Learn how to repeatedly execute a statement.

INTRODUCTION

Methods —functions, procedures, and constructors— were introduced in Sec. 1.3.3. In this chapter, we describe methods in depth, showing not only how to write them and to call, or invoke, them but also how they are executed by the computer. We will discuss two kinds of Java methods: *instance* methods and *static* methods.

2.1 Java methods are recipes

Activity
2-1.1

Cookbooks are filled with recipes. A recipe is a set of instructions for a cook to carry out. Java classes are like cookbooks, and Java methods are like recipes. Each method is a sequence of instructions for a computer to carry out.

Invoking a recipe or method

A recipe may require the use of another recipe. For example, a chocolate cake recipe may contain the instruction

Use White icing, page 250.

When a cook reaches that instruction when making chocolate cake, they pause,

make the white icing, and then continue with the chocolate cake recipe. Similarly, Java methods can contain instructions to carry out other methods. In Java, an instruction that pauses the current method and executes another method is called a *method call* or *invocation*.

In Java, we say that a method is being *executed* while the computer (or you) is carrying it out.

Parameters

Rombauer and Becker's *Joy of Cooking* has a recipe for Chocolate Apricot Cake. This recipe merely says to make another recipe, for chocolate prune cake, but to substitute apricots for the prunes. Thus, the call of the prune cake recipe asks for a substitution of one ingredient for another:

Chocolate Apricot Cake

Follow the recipe for: Chocolate Prune Cake, but

Substitute for the prunes: 1 cup cooked pureed apricots.

Omit the spices

...

This substitution of ingredients in recipes, with the substitution being indicated at the call on the recipe, is an important concept —much more so in programming than in cooking! We illustrate how the concept works in terms of recipes.

First, let's write a recipe for Chocolate X Cake, where X is a fruit to be named later. X is called a *parameter* of the recipe.

Chocolate (X) Cake

Sift: 1 1/2 cups cake flour

Resift with: 1/2 t baking soda, ...

Add: 1 cup X // Note the use of parameter X here

...

To use this recipe to make a chocolate prune cake, we use the instruction

Make Chocolate (prunes) Cake.

Executing this instruction results in following the Chocolate (X) Cake recipe with prunes substituted for X. We can use the same recipe to make an apricot cake as well:

Make Chocolate (apricots) Cake.

The Chocolate (X) Cake recipe is a *parameterized* recipe, with X being the parameter. In the same way, methods in Java are parameterized sequences of instructions.

We recommend that you reread this section (Sec. 2.1) again in a week, especially if you are confused about parameters and method calls in Java.

2.2 The black-box view of a method

Activity 2-1.2

You know that a *method* is the programming equivalent of a recipe. We now see how methods are used in Java.

2.2.1 The anatomy of a method header

The definition of a method has three parts: specification, header, and body. Here, we describe the first two parts, which are used to understand a call on a method. When you call a method, we say that you are the *client* or *customer* of the method. Below is a method definition, with the contents of the body not shown:

```
/** Draw a line in graphics window from
    pixel (x1, y1) to (x2, y2) */
public void drawLine(int x1, int y1, int x2, int y2) {
    ...
}
```

The first part of a method declaration is a comment that describes what the method does. It is called a *specification*. As you can see from the specification given above, method `drawLine` draws a line in the graphics window.

The second part of a method declaration is the *method header*, which contains (in order) these items:

Style Note 13.1.2: method names

- **Modifiers.** In this header, the one modifier, **public**, indicates that every class can use this method. (You will see **private** methods later on.)
- **The return type.** Keyword **void** indicates that this method is a *procedure*, which is a method that does not return a value. For a *function*, the type of value the function returns replaces **void**.
- **The name of the method**, in this case, `drawLine`.
- **Declarations of the parameters** of the method, enclosed in parentheses and separated by commas. Each parameter declaration consists of a type and the name of the parameter (which is an identifier). In this case, all four parameters — `x1`, `y1`, `x2`, and `y2` — are of type **int**.

The *signature* of a method consists of the name of the method and the number and types of its parameters. We write the signature of `drawLine` as follows:

```
drawLine(int, int, int, int)
```

Style Note 13.1.1: parameter names

The idea of a parameter was discussed in Sec. 2.1 when relating methods to recipes. There, a parameter `x` was associated with a value — a bunch of prunes or apricots. Similarly, in Java, a value gets associated with the parameter when the method is called. The idea of a parameter may still seem foreign to you, so we recommend that you memorize the following definition:

Parameter: A parameter is a variable that is declared within the parentheses of a method header.

Java syntax: Typical procedure and function declarations

```
/** Comment describing what the procedure does */
public void method-name ( parameter-declarations ) { ... }
```

Each *parameter-declaration* has the form *type identifier* (just like all variable declarations). Adjacent *parameter-declarations* are separated by commas. There may be 0 parameter declarations. In a function declaration, keyword **void** is replaced by the type of value that the function calculates.

Style Note

13.3.1:
method specs

Above, we made a point of including the specification of a method as part of its definition. The specification is not needed for the program to compile and run; it is just a comment. But clients of your code need the specification to understand how to use the method. Get in the habit of always writing the specification of a method before you write the method body.

Here is some motivation: during your programming career you will come to rely heavily on the online Java API specifications. The online specifications were automatically extracted from the comments in the code and turned into HTML web pages using a tool called *Javadoc*; those comments were written by programmers. Without those comments the API classes would be useless. When you get a programming job you will have to write documentation: you will have no choice! Also, if you learn to comment now, it will serve you well: programmers who work with you will sing your praises.

Plus, you will get better grades.

In this chapter, we discuss mainly procedures and functions. A procedure call is a statement: it is *executed*. A function call is an expression, with a type, so it is *evaluated*. Throughout this chapter, we use the word *method* when discussing methods in general, and when we need to distinguish the two, we use *procedure* or *function*.

Here is an example of a specification and header of a function.

Ad hoc polymorphism. *Polymorphism*, from a Greek word meaning *multiform*, means “capable of having or occurring in several distinct forms”. When we write the calls `println(5)`, `println(b || c)`, and `println("xyz")`, it looks like *one* procedure `println` is able to handle arguments of many types, and if that were the case, this would be an instance of *parametric polymorphism*. Instead, in Java, one writes several different procedures with the same name but with different parameter types — the procedure name is *overloaded*. Java is able to distinguish which procedure to call based on the types of the arguments of the call. This is known as *ad hoc polymorphism*. We will see other types of polymorphism later on, for polymorphism is an important feature of OO languages.

Polymorphism in programming languages was first discussed by Christopher Strachey in 1967, but the first major language that included polymorphism in a big way was Robin Milner’s language ML, in 1976.

Java syntax: Procedure call (or procedure invocation)

procedure-name (*argument* , ... , *argument*) ;

Each *argument* is an expression whose type is the same as or narrower than the corresponding parameter of the procedure being called.

Example: `drawRect(5, 10, 20, 30);`

Purpose: We can view execution of a procedure call as doing what the specification of the procedure call says (with the parameters in the specification replaced by the arguments of the call).

```
/** = the larger of x and y */
public int larger(int x, int y) { ... }
```

You can tell that this is a function because of the type `int` after `public`, which indicates the type of value that the function produces. The specification indicates that a call to the function evaluates to the larger of the two parameters `x` and `y`.

2.2.2 The procedure call

Activity
2-1.4

We now explain how the method specification and method header are used in writing a *method call*, or *method invocation*, as it is sometimes called.

```
/** Draw a line from pixel (x1, y1) to pixel (x2, y2). */
public void drawLine(int x1, int y1, int x2, int y2) {
    ...
}
```

Suppose we want to use procedure `drawLine`, shown above, to draw a line in the graphics window from pixel (20, 20) to pixel (80, 40). Notice that if `x1`, `y1`, `x2`, and `y2` in procedure `drawLine` are replaced with 20, 20, 80, and 40, the `drawLine` spec says that the procedure will do exactly what we want:

Draw a line from pixel (20, 20) to pixel (80, 40)

To write this statement in Java, we use a form of statement called the *procedure call*. Here is an example:

```
drawLine(20, 20, 80, 40);
```

This procedure call consists of:

- The name of the procedure, `drawLine`.
- A list of four integers, separated by commas and enclosed in parentheses; these are the *arguments* of the call.
- A semicolon.

A method call has one argument for each parameter of the method. The first argument corresponds to the first parameter, the second argument to the second

parameter, and so on. The type of each argument must be the same as or narrower than the type of the corresponding parameter.

In our example, since each parameter of procedure `drawLine` is declared using keyword `int`, the corresponding arguments must be integer-valued.

Determining what execution of a procedure call does

Learn to rely entirely on the specification and header of the method to determine what a method call will do: simply copy the specification but replace each parameter in it by the value of the corresponding argument. The result is a statement that is equivalent to the call.

Here is an example. Consider this procedure:

```
/** Draw a rectangle with top-left corner at pixel (tx, ty) with
    height h and width w. */
public void drawRect(int tx, int ty, int w, int h) { ... }
```

To figure out what the call

```
drawRect(20, 30, 25, 40);
```

does, make a copy of the specification and replace each parameter in that copy by the value of the corresponding argument:

```
Draw a rectangle with top-left corner at pixel (20, 30) with
height 40 and width 25.
```

The general form of a procedure call

A procedure call consists of:

- An identifier: the procedure name.
- Zero or more arguments, separated by commas and enclosed in parentheses.
- A semicolon.

There are several rules for the number and types of arguments:

- A method call has one argument for each parameter of the method.
- An argument is an expression, and its type must be the same as or narrower than the type of the corresponding parameter.
- If a method call has no arguments, the parentheses are still necessary.

Thus far, the arguments of method calls have just been integers, but any expression (of a suitable type) could be used. For example, the argument 40 could have been written as:

```
20 + 2 * 10
```

Activity
2-2.6**Writing a procedure call**

Suppose you have to write a program segment to carry out some task, and you believe that a call to a certain method can be used for it. In such a situation, try to rewrite the task so that it is the same as the specification of the method, but with expressions instead of parameters. The call will then be easy to write.

For example, suppose we want to:

Draw a rectangle with top-left corner (20, 40) and
lower-right corner (40, 70).

Here is method `drawRect` again:

```
/** Draw a rectangle with top-left corner at pixel (tx, ty) with
    height h and width w. */
public void drawRect(int tx, int ty, int w, int h) { ... }
```

The specification is not written in the same form as the desired task. So, we rewrite the task. The specification of `drawRect` uses the width and height of the rectangle, so we figure out the formulas for them and rewrite the task as:

Draw rectangle with top-left corner (20, 40),
height $70 + 1 - 40$, and width $40 + 1 - 20$.

Because this rewritten task has the same form as the specification of `drawRect`, we can easily write it in Java as the call

```
drawRect(20, 40, 40 + 1 - 20, 70 + 1 - 40);
```

Some programmers would look at the original task and immediately write the call

```
drawRect(20, 40, 21, 31);
```

They have saved two steps. They didn't rewrite the specification, and they calculated the width and height of the rectangle in their heads. You may do the same thing as long as you don't make a mistake!

As you will soon see, one of the hardest parts of programming is to find and correct mistakes. This process is called *debugging*. To debug, you run the program with various test cases, detect errors in the output, find the programming errors, and fix them. Often, more time is spent debugging than writing the program in the first place. Obviously, if you don't put mistakes in a program, debugging is much easier. You can approach this goal by developing a program in small steps that you know are correct, even if this approach seems to take more time.

Above, we took the small step of rewriting the task so that it looked similar to the specification of the method we wanted to call. We left to the computer the task of calculating the width and height from their formulas. Both of these choices helped eliminate potential errors.

Java syntax: Function call (or function invocation)

function-name (*argument* , ..., *argument*)

Each *argument* is an expression; its type is the same as or narrower than the type of corresponding parameter of the function being called.

Example: `larger(x * x + y, y * y + x)`

Purpose: View evaluation of a function call as yielding the value given by the specification of the function (with the parameters replaced by the arguments of the call).

2.2.3 The function call

Activity
2-4.2

Writing a function call, or function invocation, is similar to writing a procedure call. The parameter declarations tell us the types of the arguments to use in the call, and the specification tells us what they are for. We give an example with a bit of an oddity: `x` and `y` are used as parameter names as well as names of the variables in the arguments. The process does not change: replace the parameter names with the values of the corresponding arguments.

Recall method `larger`, whose specification and header we gave earlier:

```
/** = the larger of x and y */
public int larger(int x, int y) { ... }
```

Suppose we want to find the larger of two expressions:

the larger of `x * x + y` and `y * y + x`.

The specification of function `larger` has the same form as our desired value; it just has parameters `x` and `y` in place of the expressions `x * x + y` and `y * y + x`. Therefore, the desired value will be calculated by the function call

```
larger(x * x + y, y * y + x)
```

Note that a function call does not terminate in a semicolon, the way a procedure call does. A procedure call is a statement to be executed; a function call is an expression to be evaluated. For example, we could use a function call within another expression:

```
45 + larger(x * x + y, y * y + x)
```

2.2.4 Self-review exercises for calls

Below are the specifications of a few methods:

```
/** Print x, x2, and x3 on a single line */
public static void print3(int x)
```



```

/** Print "true" if  $x + y > z$  */
public static void print4(int x, int y, int z)

/** = the value of the statement " $x + y$  is greater than  $z$ " */
public static boolean testLengths(int x, int y, int z)

/** = the value of the statement " $s$  contains the letter  $e$ " */
public static boolean containsE(String s)

/** = the larger of  $x^2$  and  $y^2$  */
public static int larger2(int x, int y)

```

For each of the following calls, state what it does, using the specifications of the methods being called. For a procedure, the specification is a command to do something; for a function, it is the value of the expression.

- SR1. `print3(3);`
- SR2. `print4(3, 4, 5);`
- SR3. `print4(a, a, b);`
- SR4. `testLengths(b, b, b)`
- SR5. `testLengths(b, c, c * c)`
- SR6. `larger2(2, -3)`
- SR7. `print3(larger2(4, -5));`
- SR8. `print4(larger2(4, -5), 7, 9);`

Answers to self-review exercises

- SR1. Print 3 , 3^2 , and 3^3 on a single line
- SR2. Print "true" if $3 + 4 > 5$
- SR3. Print "true" if $a + a > b$
- SR4. the value of the statement " $b + b$ is greater than b "
- SR5. the value of the statement " $b + c$ is greater than $c * c$ "
- SR6. the larger of 2^2 and $(-3)^2$
- SR7. Print z , z^2 , and z^3 on a single line, where z is the larger of 4^2 and $(-5)^2$.
- SR8. Print "true" if $t + 7 > 9$, where t is the larger of 4^2 and $(-5)^2$.

Java syntax: print procedure call

```
System.out.print( expression );
```

Example: `System.out.print(5);`

Execution: Place the value of the *expression* in the Java console (which is a window or pane that contains error messages and output from such `print` and `println` calls).

Java syntax: println procedure call

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

Example: `System.out.println(5);`

Execution: Place the value of the *expression* in the Java console and then start a new line (in the Java console).

2.3 Method bodies

You know about the caller's view of a method (as opposed to the writer's view), and you know how to understand a method call. In this section, we investigate the third part of a method definition, the method body, and discuss its execution.

2.3.1 The procedure body

Style Note
13.2, 13.2.4:
indentation
conventions

A procedure body is a sequence of statements enclosed in braces `{ }`. Here is an example:

```
/** Print b, c, and b + c on separate lines. */
public static void print3(int b, int c) {
    System.out.println(b);
    System.out.println(c);
    System.out.println(b+c);
}
```

This procedure body contains three statements. Notice the indentation:

- The opening brace `{` appears on the same line as the header;
- The sequence of statements is indented; and
- The closing brace `}` appears indented exactly under the header.

This convention is used by many Java programmers. We use it throughout the text.

Note: Class `java.lang.System` has in it a `static` variable `out`, which refers to a `PrintStream` object. `PrintStream` objects deal with output and have a method `println`, which prints its argument, followed by a new-line character.

Execute this statement in your IDE:

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

Compiling and calling static methods

Every method needs to be inside a class, so in order to test `print3` we must write a class in which to place it. Over the next few pages, we will write several related methods, including `print3`. We will create a single class, `PrintExample`,

to contain them. `PrintExample` has nothing to do with `JFrames` or `Dates` or any other API class that we have seen; rather, it exists only as an organizational tool. We do not want to customize an existing class, so we leave off the **extends** clause. We place the method inside the class as before:

```
public class PrintExample {
    /** Print b, c, and b + c on separate lines */
    public static void print3(int b, int c) {
        System.out.println(b);
        System.out.println(c);
        System.out.println(b+c);
    }
}
```

How do we call `print3` with, say, arguments `-3` and `4`? Much like we write `Math.max(-3, 4)` to call static method `max` in class `Math`. We write:

```
PrintExample.print3(-3, 4);
```

Before you continue, type class `PrintExample` into your IDE and have a call to `print3` executed.

Variable scope

The *scope* of a variable is the area of a program in which that variable can be used. The scope of a parameter is the method body. Thus, two different meth-

```
public class PrintExample {
    /** Print b, c, and b + c on separate lines */
    public static void print3(int b, int c) {
        System.out.println(b);
        System.out.println(c);
        System.out.println(b+c);
    }

    /** Print b + c */
    public static void printSum(int b, int c) {
        System.out.println(b + c);
    }

    /** Print b + c and b + c * c */
    public static void printSums(int b, int c) {
        printSum(b, c);
        printSum(b, c * c);
    }
}
```

Figure 2.1: Class `PrintExample`, with three procedures

ods can use the same name for a parameter. For example, we can define the following procedure inside class `PrintExample` (place it either before or after method `print3`), even though it uses the same parameter names as `print3`:

```
/** Print b + c */
public static void printSum(int b, int c) {
    System.out.println(b + c);
}
```

Within the body of method `print3`, `b` refers to the parameter declared in the definition of `print3`; within the body of method `printSum`, `b` refers to the parameter declared in the definition of `printSum`. The variables are not related.

2.3.2 Executing a procedure call

We now discuss how a procedure call is executed. This is necessarily a high-level explanation. Later, we will give a lot more detail, giving a model that explains precisely how Java method calls work.

Suppose we have two variables, `x` and `y`, with values 20 and 5. We draw these variables as boxes, with the names to the left and the values inside:

`x` 20 `y` 5

In this situation, execution of the procedure call

```
PrintExample.print3(x, 2 * y);
```

proceeds as follows:

1. Draw the parameters of the method, as variables.
2. Evaluate the arguments of the call and store their values in the corresponding parameters of the procedure.
3. Execute the statements of the body of the procedure.
4. Erase the parameters of the method.

In this case, argument `x` corresponds to parameter `b` and argument `2 * y` corresponds to parameter `c`. Therefore, the first and second steps result in this state:

`b` 20 `c` 10

In performing step three, the statements are executed one by one, beginning with the first. If a parameter name is used, the value of the parameter is used in its place. In this case, execution of the method body results in three values being printed on three separate lines: 20, 10, and 30. Finally, the parameters are erased.

It is important that you can execute a procedure call yourself, as just shown.

As you already know, a method body can contain calls on other methods. In fact, it is typical for one method to contain several calls on others. As an exam-

ple, we write a method `printSums`, which contains two calls to method `printSum`. Class `PrintExample`, with all three methods discussed in this section, appears in Fig. 2.1.

Method `printSums` go inside class `PrintExample`. Because `printSum` and `printSums` are defined in the same class, procedure `printSums` can call `printSum` without having to write `PrintExample.printSum(...)`.

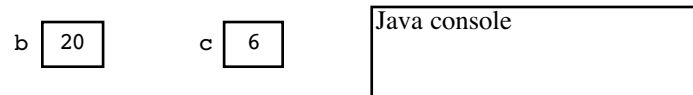
At times, you may want to execute parts of a program by hand. In doing so, there are two different ways that we might execute a call: by *stepping over* it and by *stepping into* it. Most IDE debuggers have these two possibilities, and it is essential that you understand the difference.

Stepping over a call

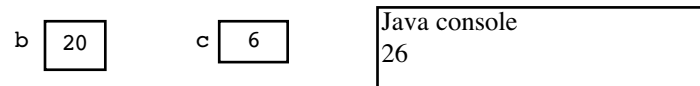
We describe what it means to *step over* a call. Suppose we are executing this call to procedure `printSums`:

```
PrintExample.printSums(20, 6);
```

We have assigned the arguments to the parameters, so that we have this situation:



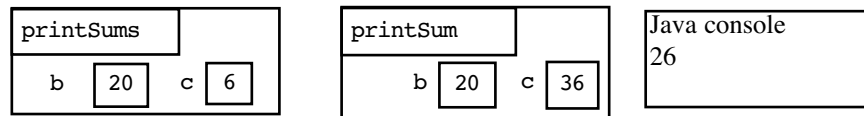
We are ready to execute the statements in the method body. The first statement is a function call `printSum(b, c)`. We execute it by *stepping over* the call. To do this, we do what the specification of the procedure says to do: print the larger of `b` and `c`. So, we place 26 in the Java console, yielding this state:



Thus, *stepping over a call* means simply to execute it as an indivisible action, doing what the specification says to do.

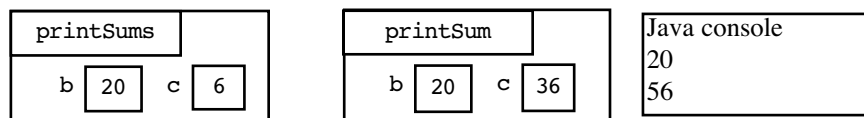
Stepping into a call

Now, the second statement is to be executed: `printSum(b, c * c)`. We execute this statement using the second method, *stepping into the call*. To do this, we go through the detailed steps mentioned earlier: assign the arguments of the call to the parameters of the method and then execute the method body. In this case, the parameter names are `b` and `c`. To avoid mixing up the parameters `b` and `c` for this new method call with those that already exist, we place them in boxes as shown below. Each box has in its upper left a subbox that contains the name of the method called.



We have carried out the first step of assigning the arguments 20 and 36 to the parameters. Notice that there are 2 b's and 2 c's in the picture. These variables are independent of each other.

We now execute the statement in the body of procedure `printSum`, printing 56 in the Java console:



Execution of the call to `printSum` is finished, so we erase the box. We are now in this state:



We summarize:

- **To step over a call**, execute it as an individual action, doing what the specification of the method does. The method being called is a black box into which we cannot look, and we rely only on its specification.
- **To step into a call**, (1) draw the parameters of the method, (2) assign the arguments of the call to the method, (3) execute the method body, and (4) erase the parameters of the method.

Suppose you are executing a program yourself, or you are using a debugger, presumably to find an error in the program. Which of these two ways you use to execute a call will depend on the situation. If you are 100% sure that a method is correct, you can step over a call to that method. However, if you believe an error may be in a particular method, step into calls to it.

2.3.3 Conditional statements and blocks

The body of a method is a sequence of statements, which are executed in the order in which they appear. So far, we have seen assignments and procedure calls as statements that can appear in a method body. We now introduce three more kinds of statement: the if-statement, the if-else-statement, and the block. The if-statement and if-else-statement are examples of *conditional statements*.

Java syntax: if-statement

```
if ( boolean-expression )
    then-part
```

Example: `if (x < 0) {`
 `x = -x;`
 `}`

Then-part: any statement

Execution: Evaluate the *boolean-expression*. If it is **true**, execute the *then-part*.

Java syntax: if-else statement

```
if ( boolean-expression )
    then-part
else
    else-part
```

then-part and else-part: each is a statement

Example: `if (x < y) {`
 `y = y - x;`
 `} else {`
 `x = x - y;`
 `}`

Execution: Evaluate the *boolean-expression*. If **true**, execute the *then-part*; otherwise, execute the *else-part*.

The if-statement

Style Note
 13.2, 13.2.1:
 indenting if-
 statements

There are situations in which you would do something depending on whether some condition is true. For example, if it is cold, you would put your coat on, but not if it is warm.

In a Java program, a *conditional statement* is used for this purpose. As an example, the statement below tests whether `x < 0`, and if so, it executes the assignment `x = -x`; . But if `x ≥ 0`, the assignment is not executed.

```
if (x < 0) x = -x;
```

An if-statement has this form:

```
if ( condition ) then-part
```

where the condition is a **boolean** expression and the *then-part* is a statement.

To execute an if-statement, evaluate the condition; if it is **true**, execute the *then-part*. If the boolean expression is **false**, execution of the if-statement is finished.

The block

Suppose we want a statement that adds 2 to both `x` and `y` if `x` is larger than `y`. To do this, we write the *then-part* of the if-statement as a *block*: a sequence of statements enclosed in braces `{` and `}`. Here is the Java code for it:

```
// Add 2 to both x and y if x > y
if (x > y) {
    x = x + 2;
    y = y + 2;
}
```

The braces `{` and `}` are used to aggregate the sequence of statements into a

single statement. In the if-statement shown above, the then-part consists of this block:

```

        {
            x= x + 2;
            y= y + 2;
        }

```

Notice the indentation: statements inside a block are indented. The opening brace is put on the same line as the condition. The closing brace appears on its own line, indented the same amount as the **if**. (A method body is a block, so it follows these conventions.)

We (and Sun Microsystems) strongly advocate using a block for the then-part even if there is only a single statement within the braces, as in this example:

```

if (x < y) {
    x= x + 2;
}

```

Why? Because often, after writing some Java code, we have to change it. Here is an example of a fairly common occurrence, even among professional programmers. We have written this statement, without braces:

```

if (x < y)
    x= x + 2;

```

and we want to change it so that the then-part adds 2 to y as well as to x. We are quite likely to simply append the new assignment, yielding:

```

if (x < y)
    x= x + 2;
    y= y + 2;

```

But this is not correct because the braces are missing, and it is actually equivalent to:

```

if (x < y)
    x= x + 2;
y= y + 2;

```

Thus, if we don't make the then-part a block right from the beginning, we are liable to make a mistake if we have to change the then-part later on.

Hereafter, we always include the braces.

The if-else statement

At times, we want to execute one thing if a condition is **true** and another if it is **false**. For this we use an if-else statement. Here is an example:

Style Note
13.2, 13.2.1:
indenting if-
statements

```
// Set z to the minimum of x and y
    if (x < y) {
        z = x;
    } else {
        z = y;
    }
```

If the condition $x < y$ is **true**, the then-part $\{z = x;\}$ is executed; otherwise, the else-part $\{z = y;\}$ is executed.

Here is the general form of an if-else statement:

```
if ( condition ) then-part
else else-part
```

where the *else-part* is a statement. To give a complete example, we write a procedure that prints the smaller of its two parameters.

```
/** Print the smaller of b and c */
public static void printSmaller(int b, int c) {
    if (b < c) {
        System.out.println(b);
    } else {
        System.out.println(c);
    }
}
```

2.3.4 Self-review exercises for ifs

In the exercises that ask you to write code, it is best that you actually type them into your Java IDE and test them to make sure that they work. We recommend starting a class `Chapter2Exercises` that will contain all the static methods you write for these self-review exercises.

SR1. Create a class called `Chapter2Exercises` that does not extend anything. Write a **public static void** method called `sr1` that has an **int** parameter `x`. Inside method `sr1`, write a conditional statement that (1) adds 1 to `x` if `x` is negative and (2) prints the value of `x`. To test this method, try these calls:

```
Chapter2Exercises.sr1(-3);
Chapter2Exercises.sr1(3);
Chapter2Exercises.sr1(0);
```

SR2. What is a *block*?

SR3. Write a conditional statement that, if `x` is negative, sets `x` to 0 and adds 1 to `y`. The then-part will have to be a block. (If you write this conditional statement in a method, the method should have two parameters.)

SR4. Write a conditional that stores x in z and y in x if z is greater than 0.

SR5. Write a conditional statement to set z to the minimum of $x + y$ and $x - y$.

SR6. In the following conditional statement, the else-part is written in English:

```
// Set d to the minimum of a, b, and c
if (a <= b && a <= c) {
    d= a;
} else { // the minimum is b or c
    Set d to the minimum of b and c
}
```

Replace the else-part by a Java statement to accomplish that task. You will end up with a *nested conditional statement*: a conditional statement that appears within another conditional statement.

SR7. In the following conditional statement, the then-part is written in English:

```
// Set d to the maximum of a, b, and c
if (a < b || a < c) {
    Set d to the maximum of b and c
} else { // the maximum is a
    d= a;
}
```

Replace the then-part by a Java statement to accomplish that task. You will end up with a *nested conditional statement*: a conditional statement that appears within another conditional statement.

SR8. Variables b , c , and d contain integers. Write a program segment that sets boolean variable t to the value of “ b , c , and d are the lengths of the sides of a triangle”. Three integers are the lengths of the sides of some triangle if and only if the sum of any two sides is at least the third side.

Answers to self-review exercises

For the answer to SR1, we provide class Chapter2Exercises and method `sr1`. For the rest, we provide only the conditional code.

```
SR1. public class Chapter2Exercises {
    public static void sr1(int x) {
        if (x < 0) {
            x= x + 1;
        }
    }
}
```

SR2. A block is a sequence of statements delimited by (enclosed in) braces `{` and `}`. It *aggregates* the sequence of statements into a single statement.

```

SR3.  if (z > 0) {
        x= 0;
        y= y + 1;
    }

SR4.  if (z > 0) {
        z= x;
        x= y;
    }

SR5.  if (x + y <= x - y) { // you could write the condition as y <= 0
        z= x + y;
    } else {
        z= x - y;
    }

SR6.  // Set d to the minimum of a, b, and c
    if (a <= b && a <= c) {
        d= a;
    } else if (b <= c) {
        d= b;
    } else {
        d= c;
    }

SR7.  // Set d to the maximum of a, b, and c
    if (a < b || a < c) {
        if (b >= c) {
            d= b;
        } else {
            d= c;
        }
    } else { // This else belongs with the first if, not the second.
        d= a;
    }

```

SR8. There are several ways to write this program segment. The following one illustrates nested if-statements:

```

if (b + c < d) { t= false; }
else if (c + d < b) { t= false; }
else if (d + b < c) { t= false; }
else { t= true; }

```

Here is a neater solution:

```

t= (b + c >= d) && (c + d >= b) && (d + b >= c);

```

Java syntax: procedure return statement
return ;

Example: return ;

Purpose: Terminate execution of a procedure call.

Java syntax: function return statement
return expression ;

Example: return b + c;

Purpose: Terminate execution of a function call and use the value of the *expression* as the value of the function call.

2.3.5 The return statement

Activity
2-3.4

When a procedure is called, the statements in the procedure body are executed one at a time, in the order in which they appear. However, it is occasionally advantageous to terminate execution of the body before the last statement has been executed. We use the return statement for this purpose. It has this form:

return;

Execution of a return statement terminates execution of the procedure body and, hence, of the procedure call. Once a return statement is executed, no more statements in the procedure body are executed.

Procedure `printSmallest` in Fig. 2.2 contains two return statements. Suppose the call `printSmallest(5, 6, 6);` is to be executed. Then, parameter `b` will be 5, `c` will be 6, and `d` will be 6. Therefore, the condition of the first if-statement will be **true**, `b` will be printed, and execution of the return statement will terminate execution of the procedure body and, hence, of the procedure call. The second if-statement and the last print statement are not executed.

Consider a call `printSmallest(6, 2, 5);`. In this case, parameter `b` will be 6, `c` will be 2, and `d` will be 5. Therefore, the condition of the first if-statement is

```

/** Print the smallest of b, c, and d */
public static void printSmallest(int b, int c, int d) {
    if (b <= c && b <= d) {
        System.out.println(b);
        return;
    }

    // { the smallest is c or d }
    if (c <= d) {
        System.out.println(c);
        return;
    }

    // { the smallest is d }
    System.out.println(d);
}

```

Figure 2.2: A procedure with **return** statements

false, so execution of the first if-statement is finished. The condition of the second if-statement is **true**, so the value of `c`, or 2, will be printed, and execution of the return statement will terminate execution of the procedure body and, thus, of the procedure call.

Using an assertion to help the reader

Style Note
13.2, 13.2.2:
assertions

The reader of a program can benefit from the insertion of comments at judiciously chosen places in the program to alert them to what is true about the variables at those places. For example, after the first if-statement in this body, it may help to indicate that `b` is not the smallest parameter. Such a description of the variables is called an *assertion* because we are asserting that it is true at a point of execution of the program. By convention, we enclose assertions in curly braces (note that the curly braces are not part of the assertion). The braces alert the reader to the fact that this comment is an assertion about the values of the variables, not a specification or command to do something.

2.3.6 The function body

Activity
2-4.1

The procedure in Fig. 2.2 prints the smallest of its three parameters. In many programs, it may be useful to use the smallest of three values in a later calculation, rather than print it, and in these applications, procedure `printSmallest` is useless. Instead, we need a function that calculates the smallest of three values and returns it for later use. This function is given in Fig. 2.3.

A function must return a value. Therefore, execution of a function must terminate by executing a return statement of the form

return *expression* ;

where the type of the *expression* is the same as (or narrower than) the type of the result of the function. Execution of such a return statement terminates the func-

```

/** = smallest of b, c, and d. */
public static int smallest(int b, int c, int d) {
    if (b <= c && b <= d) {
        return b;
    }

    // {the smallest is c or d}
    if (c < d) {
        return c;
    }

    // {the smallest is d}
    return d;
}

```

Figure 2.3: A function that returns the smallest of its parameters

tion body, and, thus, the function call, and yields the value of the *expression* as the result of the call.

Since execution of a function body must terminate with execution of a return statement, a return statement is usually the last statement in the function body. However, return statements may appear in other places as well, see e.g. Fig. 2.3.

Suppose the call `smallest(2, 6, 6)` is to be evaluated. Then, parameter `b` is 2, `c` is 6, and `d` is 6. Therefore, the condition of the first if-statement is true, and the then-part of that if-statement, `return b;`, is executed. This terminates execution of the function body and yields 2 as the value of the function call.

Executing a function call

Earlier, we gave a list of four steps for executing a procedure call. The only difference in executing a function call is that the value of the expression in the return statement whose execution terminates the call has to be returned as the value of the function. For purposes of completeness, we summarize here the steps in executing a function call:

1. Draw the parameters of the function, as variables.
2. Evaluate the arguments of the call and store their values in the corresponding parameters of the function.
3. Execute the statements of the body of the function.
4. To execute `return e;`, evaluate expression `e`, erase the parameters of the method, and use the value of `e` as the value of the function call.

2.3.7 Local variables

Activity
2-3.1

A *local variable* is a variable that is declared within a method body. The *scope* of a variable is the area of a program where it can be used. The scope of a local variable is the sequence of statements that follows its declaration, up until the end of the block in which it is declared. The declaration of a local variable has this form:

type variable-name ;

Its initial value is unknown, and the variable cannot be referenced until a value has been stored in it. An initializing declaration of a local variable has this form:

```
/** Using g, draw a triangle that fits in the rectangle drawn by drawRect(x, y, w, h).
    One side is the base of the rectangle; the other two sides meet at pixel (x + w / 2, y). */
public void drawTriangle(Graphics g, int x, int y, int w, int h) {
    g.drawLine(x, y + h, x + w, y + h);
    g.drawLine(x, y + h, x + w / 2, y);
    g.drawLine(x + w, y + h, x + w / 2, y);
}
```

Figure 2.4: Drawing a triangle

Java syntax: local variable declaration

type variable ;

Example: `int temperature;`

Purpose: Introduce a variable that can be used in the sequence of statements that follows the declaration.

Java syntax: initializing declaration

type variable-name= expression ;

Example: `int temperature= 50;`

Purpose: Introduce a variable that can be used in the sequence of statements that follows the declaration and give it an initial value.

Style Note

13.1.1:
local-variable
names

type variable-name= expression ;

Method `drawTriangle` of Fig. 2.4 draws a triangle in a graphics window `g`. Its body is not as easy to understand as it could be because of the many expressions, some of which are duplicated. For example, the expression `y + h` appears four times. Not only does this complicate the body, it is inefficient. We can make the body clearer and more efficient by using local variables.

Style Note

13.4:
describing
variables

Figure 2.5 shows the same procedure as in Fig. 2.4, but with three local variables. While their declarations and initializations make the procedure look longer, the statements that do the work (the three calls to procedure `drawLine`) are easier to understand.

Note the use of a comment to describe what a local variable is being used for. The comment mentions not only the local variable but other variables as well. Variables are related to each other, and one often describes them together.

Quite often, a declaration of a variable is followed by an assignment that provides the variable with its initial value. It is possible —and usually advantageous— to combine the two into a single *initializing declaration*. For example, the three local variables in Fig. 2.5 were declared and initialized using these three initializing declarations:

```
int y1= y + h;           // (x, y1) is the left lower vertex
int x1= x + w;           // (x1, y1) is the right lower vertex
int x2= x + w / 2;       // (x2, y) is the top vertex
```

A variable can be declared only once, but it can be assigned many times. In

```
/** Using g, draw a triangle that fits in the rectangle drawn by drawRect(x, y, w, h).
    One side is the base of the rectangle; the other two sides meet at pixel (x + w / 2, y). */
public void drawTriangle(Graphics g, int x, int y, int w, int h) {
    int y1= y + h;           // (x, y1) is the left lower vertex
    int x1= x + w;           // (x1, y1) is the right lower vertex
    int x2= x + w / 2;       // (x2, y) is the top vertex
    g.drawLine(x, y1, x1, y1);
    g.drawLine(x, y1, x2, y);
    g.drawLine(x1, y1, x2, y);
}
```

Figure 2.5: Drawing a triangle using local variables

Java syntax: for-loop for processing a range of integers $b..c$

```
for (int i= b; i <= c; i= i + 1) {
    Process i;
}
```

Purpose: To perform the sequence of statements “Process b ; Process $b+1$; ...; Process c ;”. Here, “Process i ” can be any statement sequence that refers to variable i . Also, b and c can be any integer expressions such that $b \leq c + 1$. If $b = c + 1$, then no integers are processed.

the following sequence, the second declaration is illegal, since variable b is declared in the first line. But the third line, the assignment to b , is legal.

```
int b= 45;    // A legal initializing declaration
int b= 61;    // An illegal declaration, since b is already declared
b= b + 2;     // A legal assignment
```

2.3.8 Processing a range of integers

This section need not be read at this time. It is presented here so that instructors who want to introduce loops early can do so. If you are not interested in studying loops at this point, skip this section.

At times, we want to write a program segment to process a range of integers. Here are some examples of tasks that we might want to perform:

- Add the squares of the integers in the range $1..100$ (i.e. the integers 1, 2, ..., 100).
- Determine whether some integer in the range $2..n$ divides an integer k .
- Find the first integer in the range $100..$ (i.e. the first integer ≥ 100) that is a power of 2 (i.e. can be written in the form 2^k for some k).
- Find the number of times the letter 'e' appears in `String s` (this requires processing the possible indices $1..(s.length-1)$ of s).

Program segments to process a range of integers are usually written using a *loop*, and often with what is called a *for-loop*. Chapter 7 covers loops in detail; here, we provide just enough information to allow you to write simple loops to process a range of integers.

Suppose you want to implement the following sequence of statements, which stores in `int` variable x the sum of the integers in the range $2..200$. Here is one way to do it:


```
(1) x= 0;
    x= x + 2 * 2;
    x= x + 3 * 3;
    x= x + 4 * 4;
    ...
    x= x + 200 * 200;
```

200 lines is a lot to type. It would be nice to be able to paraphrase it like this:

For each number *i* in the range 2..200, add *i***i* to *x*.

Here is a for-loop (preceded by an initializing declaration of *x*) that does just that:

```
(2) int x= 0;
    for (int i= 2; i <= 200; i= i + 1) {
        x= x + i * i;
    }
```

Variable *i* is called the *loop counter*.

The constituents of this loop are:

- The part of the loop within the parentheses:
 - The initializing declaration of **int** variable *i* (initialized to 2);
 - A semicolon;
 - The loop-condition *i* <= 200. It can be any boolean expression;
 - A semicolon;
 - An assignment that adds 1 to loop counter *i*.
- The block after the parentheses (the opening brace { followed by the sequence of statements followed by the closing brace }) is called the *repetend* of the loop. *Repetend* means “the thing to be repeated”.

Program segment (2) performs exactly the same task as program segment (1) above. It is just a shorthand version. Thus, sequence (2) executes *x*= 0; and then executes the statement

```
x= x + i * i;
```

with *i* containing 2, then with *i* containing 3, and so on up to *i* containing 200.

You can (and should) put that code in a method and step through it in your debugger.

As a second example, we write a loop that performs the following assignment:

```
x= 1 * 1 - 2 * 2 + 3 * 3 - 4 * 4 + ... + 21 * 21 - 22 * 22;
```

Here, the squares of odd integers are added and the squares of even integers are subtracted, so this assignment is equivalent to:

```

x= 0;
x= x + 1 * 1;
x= x - 2 * 2;
...
x= x + 20 * 20;
x= x - 21 * 21;
x= x + 22 * 22;

```

Thus, we write the following loop (with initialization):

```

x= 0;
for (int k= 1; k <= 22; k= k + 1) {
    if (k % 2 == 0) {
        x= x - k * k;
    } else {
        x= x + k * k;
    }
}

```

The general for-loop

We have shown the use of the for-loop to process a range of the integers. That should be enough for now, and the following may be skipped. The general for-loop is discussed in Chap. 7. However, for those who want to know a bit more at this point, we discuss the for-loop further here.

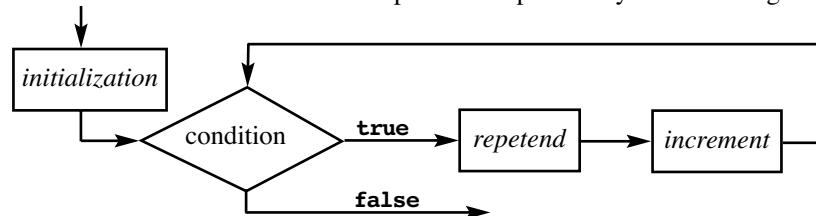
The general form of the for-loop is

```
for ( initialization ; condition ; increment ) repetend
```

where

- The *initialization* is an assignment to the *control variable*, including, optionally, its declaration.
- The *condition* is a boolean expression.
- The *increment* is generally an assignment (without a semicolon) to the control variable.
- The *repetend* is any statement —usually a block.

Execution of the for-loop can be explained by the following flow chart



As an example, we write a loop (with initialization) that sums the even positive integers 2, 4, ... until the sum gets over 500.

```

/* Set x to the sum of the first even integers 2, 4, 6, ... such the sum
   > 500 but the sum of one less even integer is ≤ 500. */
x= 0;
for (int k= 2; x <= 500; k= k + 2) {
    x= x + k;
}

```

Here is another example. Assume n is at least 2. We write a loop segment that sets m to the largest divisor of n that is smaller than n . We do not declare n and m because we assume they are declared elsewhere. This loop is strange because its repetend does nothing. Everything is done in the test of the condition and the decrementing of m . Execute this loop by hand, using for n the value 5, so that you see how it works.

```

/* Precondition: n ≥ 2. Store in m the largest integer that is
   less than n and that divides n. */
for (m= n - 1; n % m != 0; m= m - 1) {
}

```

2.3.9 Self-review exercises for for-loops

After writing an exercise, test it on the computer. That is the best way to determine that your answer is correct. This usually requires you to calculate some of the answers by hand for small values of the variables in question. Most, but not all, of these loops require a statement that initializes a variable or two.

SR1. Write a for-loop to print the values in the range 4..24 on the Java console (use statement `System.out.println(...);`).

SR2. Assume $n \geq 2$. Write a loop to store this value in x :

$$1 * (n - 1) + 2 * (n - 2) + 3 * (n - 3) + \dots + (n - 1) * (n - (n - 1))$$

SR3. Given $n \geq 1$, write a loop that stores in **double** variable v the sum:

$$1 / 1 + 1 / 2 + 1 / 3 + \dots + 1 / n$$

What happens to the sum as n gets large? What would happen if $n < 1$? (Try it!)

SR4. Given $n \geq 1$, write a loop that stores in **double** variable v the sum:

$$1 / (1 * 1) + 1 / (2 * 2) + 1 / (3 * 3) + \dots + 1 / (n * n)$$

Once you have tested your loop to make sure it is right, try it for increasingly large values of n . What value does the sum “converge” to as n gets larger?

SR5. Given $n \geq 2$, write a loop to find the smallest integer that is greater than 1 and that divides n .

SR6. Given $n \geq 2$, write a loop that sets boolean value *b* to the value of the sentence “no integer in the range $2..(n - 1)$ divides *n*”. Make sure you test your answer for various values of *n*.

SR7. Given $n \geq 1$, write a loop that stores in *x* the sum of the first *n* values of this sequence: 1, 2, -3, 4, 5, -6, 7, 8, -9,

2.4 Static versus non-static methods

The purpose of this section is to make clear, once more, the difference between static and non-static components of a class. This material is placed here for completeness, so that everything about methods is in this one Chap. 2. We assume that you know about class definitions and how to draw an instance (manila folder) of a class.

Below is a class that contains a static method called `staticMethod` and a static variable `staticVar`, as well as a non-static method called `nonStaticMethod` and variable `nonStaticVar`:

```
public class C {
    static int staticVar;
    int nonStaticVar;
    static void staticMethod(int x) {...}
    void nonStaticMethod(int y) {...}
}
```

The distinction between static and non-static components is simple: static components go directly into the file drawer for the class, while non-static components appear in each and every instance of the class. Figure 2.6 illustrates this, showing a filing cabinet with a drawer named *C* and, to its right, the contents of the drawer. Static components `staticVar` and `staticMethod` are in the drawer. The drawer also contains two instances of class *C*. (We would create them using `new`-expressions.) Note that both instances contain a field `nonStaticVar` and a method `nonStaticMethod` because these are defined to be non-static.

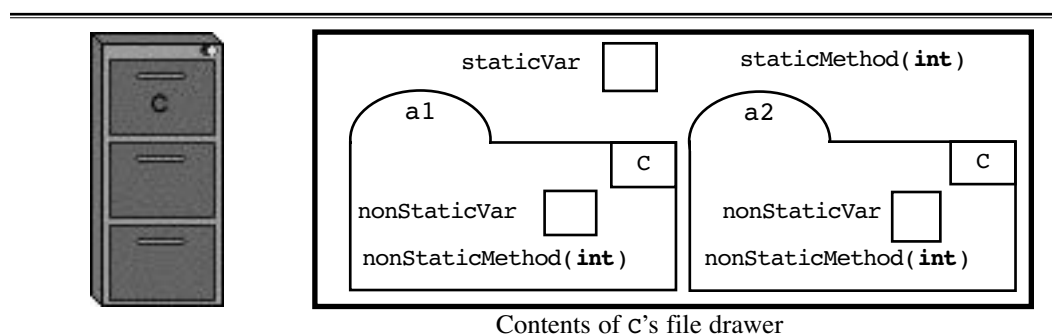


Figure 2.6: The file drawer for class *C*

Any static variable or static method can be referenced using the name of the class: `C.staticVar` and `C.staticMethod(5)`.

The non-static components cannot be referenced without having a variable that contains the name of an instance. Suppose a variable `cVar` contains the name `a1`. Then `a1`'s two components can be referenced using `cVar.nonStaticVar` and, say, `cVar.nonStaticMethod(5)`.

The inside-out rule

We consider the question of what, exactly, can be referenced from the body of method `nonStaticMethod` in instance `a1`. Almost all programming languages have a general inside-out rule that can be used to answer this question.

Inside-out rule: Code in a construct can reference any of the names that are declared or defined in that construct, together with any names that appear in the enclosing construct(s) — unless the name is declared twice, in which case the closer one prevails.

Naturally, there may be restrictions on this general inside-out rule. For example, a local variable cannot be used in a statement that precedes its declaration, and the use of some of Java's access modifiers (**public**, **protected**, **private**, and the default) may give further restrictions. But this general inside-out rule is a good first step in understanding scope issues.

Here, we use the inside-out rule to determine what can be referenced from `a1.nonStaticMethod`'s method body, based on Fig. 2.5:

1. The parameter and any local variables of `a1.nonStaticMethod`.
2. Field `nonStaticVar` and method `nonStaticMethod` itself because they appear in an enclosing construct.
3. Static members `staticVar` and `staticMethod` because they appear in an enclosing construct.

By the inside-out rule, method `staticMethod` can reference only its parameters and local variables, static variable `staticVar`, and `staticMethod` itself. The inside-out rule does not let it reference components of instances `a1` and `a2`.

We will apply the inside-out rule in other situations later on.

2.5 Stepwise refinement

See lesson 2-5 for a discussion of stepwise refinement.

You now have an understanding of the following kinds of statements: assignment statements, conditional statements (if- and if-else-statements), method calls, statements to read and write on the Java console, statements to read and display values in fields of a `JLiveWindow` GUI, and statements to draw in a graphics window. With this knowledge, we discuss the task of developing sequences of Java statements to solve some task. In an ideal setting, we use what is called *top-down programming*, or *stepwise refinement*.

2.5.1 Stepwise refinement: making coffee

This discussion is also presented in activity 2-5.1, using synchronized animation.

We introduce the notion of stepwise refinement with this problem:

Get coffee in the morning.

This statement says *what* to do; we want to replace it by a sequence of instructions, or statements, that say *how* to do it.

So how do we get coffee? If no one in the house has made coffee yet, we have to make it. After that, we can pour coffee into a cup:

```
// Get coffee in the morning.
if (coffee not made) {
    Make coffee.
}
```

Pour coffee into cup.

Note that we made the original task into a *statement-comment*, and we have written its implementation underneath it. A *statement-comment* states *what* the corresponding code does; it is often a big help when reading the code.

Note also that we use Java notation where appropriate. Some people prefer to continue to write everything in stylized English —they call it *pseudocode*. We prefer using programming notation whenever it is reasonable, so that the final result is as close to being a program as possible.

We now work on refining the statement *Make coffee*. There are two choices: real or instant, so we insert an if-else statement with those two choices:

```
// Get coffee in the morning.
if (coffee not made) {
    // Make coffee.
    if (real coffee desired) {
        Brew coffee.
    } else {
        Make instant coffee.
    }
}
```

Pour coffee into cup.

We now have two statements that could be refined further: *Brew coffee* and *Make instantcoffee*. We can refine them in any order; they are independent. We also have a choice of notation. If we continue to make each statement into a statement-comment, the program becomes harder and harder to read. It exhibits too much nested structure. Another choice is to introduce a method for a task and to make an English statement into a procedure call. This allows the program to stay short and simple. So let us create two methods:

```

// Get coffee in the morning
if (coffee not made) {

    // Make coffee
    if (real coffee desired) {
        brewCoffee();
    } else {
        makeInstant();
    }
}

Pour coffee into cup.

/** Brew coffee. */
public void brewCoffee() { }

/** Make instant coffee. */
public void makeInstant() { }

```

Notice that we write the specification and header of the new procedures, but we leave their bodies empty. If one of them were a function, we would write a return statement that produced some value of the right type, simply so that we could compile the program, if we were writing and testing the program incrementally.

We can now work on either procedure; we decide to implement the body of procedure `brewCoffee`. To brew coffee, we grind coffee, put the coffee in the filter, put the filter in the coffee maker, add water, and turn the coffee maker on:

```

/** Brew coffee */
public void brewCoffee() {
    Grind coffee.
    Put coffee in filter.
    Put filter in coffee maker.
    Add water.
    Turn coffee maker on.
}

```

We stop the stepwise refinement of this program now, for the idea should be clear. (Much more detail could be added — e.g. how do we boil water?)

2.5.2 A summary of stepwise refinement

We summarize the ideas of stepwise refinement, or top-down programming. The development of the program consists of a series of steps.

1. Each step consists of replacing a statement that says *what* to do by a sequence of one or more statements that describe *how* to do it. The replacement is called the *implementation*, or refinement, of the statement.

There are many choices for what the implementation can be:

2. The implementation can be a single statement (e.g. an assignment statement or a conditional statement), a sequence of statements (some in English and some in the programming language), and so forth.

There is a choice of notation to use:

3. We prefer to use a mixture of English and Java, moving closer and closer to Java at each step, so that the final result *is* a Java program.

To this end:

4. We make heavy use of statement-comments.
5. We introduce methods and method calls in appropriate places, to keep the program appearance simple.

It is important to realize that:

6. At each step, the program that has been written so far is correct. However, some of the methods are not completely in Java. They contain English statements.

One point that we have not discussed yet is the introduction of variables:

7. If a step of a top-down design introduces variables, this may cause changes in several statements that have to refer to those variables. When variables are introduced, it is important to write program comments that describe the meanings of the variables.

Top-down design in other fields

Activity 2-5.2 discusses Poe's amazing essay. You *have* to see it. You can get the essay and the poem in footnotes.

We often think that what we are doing is new and exciting and all ours. This is true also of top-down design when it was first discussed in programming in the very early 1970s. However, top-down design is used in almost all fields, since it is about the most logical way one can think of to develop anything. In fact, even people in poetry have used it. Edgar Allen Poe, for example, wrote an essay on how he developed the poem *The Raven*. He did not use the term top-down design, but he sure used the concept! We urge you to put the *ProgramLive* CD in your computer and watch a discussion of his essay.

2.5.3 Top-down development of a Java task

Activity 2-5.3 does this better, using time and synched animation. Watch it!

We look at stepwise refinement in a real programming situation. In this development, we emphasize that we are not only doing stepwise refinement, we are also using incremental programming and incremental testing. At every step, we compile the program to make sure that it is syntactically correct and then do whatever testing we can. This is far better than writing the whole program and

then trying to compile it and debug it. Waiting to compile until the program is done may mean seeing dozens (literally) of syntactic error messages, which can be overwhelming. Also, if we test only when we believe the program is finished, we have no idea how to go about it. Finished programs are possibly *thousands* of lines long. There will be too much to test all at once, too many places that might have errors. Programming, commenting, compiling, and testing incrementally lends a great deal of control — and will your preserve sanity at 3am the day your assignment is due.

Anglicizing integers

We write a function that anglicizes integers, producing their English equivalents. For example, for the integer 2001, our function produces the String "two thousand one". We start with a specification and a function heading:

```
/** = English equivalent of n, for 0 < n < 1,000,000 */
public static String anglicize(int n)
```

For small numbers like 2 and 5, the English equivalent is easy to get; the larger the number, the more work it takes to get its English equivalent. Because of this we investigate large numbers, just to see what the problems are. The value 1000 is a point of differentiation: if $n \geq 1000$ the result has the word "thousand" in it; if $n < 1000$, it does not. Thus we can make an educated guess that the first step of the body of anglicize is to make this differentiation:

```
/** = English equivalent of n, for 0 < n < 1,000,000 */
public static String anglicize(int n) {
    if (n >= 1000) {
        return anglicized n (for 1000 <= n < 1000000);
    } else {
        return anglicized n (for n <= 1000);
    }
}
```

Refining the expression "anglicized n (for n >= 1000)"

Anglicizing an integer n in the range $1000 \leq n < 1000000$ requires breaking it into the parts $n / 1000$ and $n \% 1000$, anglicizing the two parts, and placing " thousand " between them. For example, the English equivalent of 2121 is the word for $2121 / 1000$ followed by " thousand " followed by the anglicization of $2121 \% 1000$. We can therefore write the then-part of the if-statement as

```
return (anglicized (n / 1000))
      + " thousand " + (anglicized (n % 1000))
```

This return statement calls for anglicizing two integers that are less than 1000, and the else-part also calls for anglicizing an integer that is less than 1000. Therefore, in three places we have to do essentially the same task. It makes sense

to write a function, say `anglicizeHundreds`, to perform this service. We show this method, together with function `anglicize` but modified to call the new function, in Fig. 2.7.

In Fig. 2.7, function `anglicize` is finished, and we should test it before proceeding. But how do we do that? Look at function `anglicizeHundreds` in Fig. 2.7. We have “stubbed it in”, by which we mean that we have fixed it to return something that allows us to test function `anglicize`. It does not produce the English words for `n`, but it does produce `n`. With this function, for the call `anglicize(2024)` we expect to get the value “2 thousand 24”. Writing `anglicizeHundreds` in this fashion allows us to test function `anglicize` thoroughly before proceeding.

In fact, if you test `anglicize` thoroughly, looking at “extreme” cases like `n = 999`, `n = 1000`, `n = 1001`, `n = 999999`, you will find an error. Calling `anglicize(1000)` yields the value “1 thousand 0”, and the 0 does not belong at the end. The problem is that we wrote `anglicizeHundreds` to handle an integer `n` in the range $0 < n < 1000$, but it is called at least once with `n = 0`. We should change the specification on `anglicizeHundreds` so that it accepts integers in the range `0..1000` and also state expressly (in the specification) that the English equivalent of 0 is the empty String “”.

In general, when programming and testing incrementally, compile often: every few lines. Test just as often: as soon as you have written the specification and header of a method, you can write a test for it. To facilitate testing, after writing the specification and header of a new method, write a body that is simple and short but that allows you to test the parts of the program that call it.

The development of function `anglicizeHundreds` proceeds along similar lines, so we do not show it. As you proceed, you will find the need for other functions, e.g. a function `teenName(n)` that produces the English equivalent of `n` in the range `10..19` and a function `tensName(n)` for `n` in the range `0..9`.

A procedural approach

The above development focused on introducing more and more functions,

```

/** = English equivalent of n, for 0 < n < 1,000,000 */
public static String anglicize(int n) {
    if (n >= 1000)
        { return anglicizeHundreds(n / 1000); +
          " thousand " + anglicizeHundreds(n % 1000); }
    else { return anglicizeHundreds(n % 1000); }
}

/** = English equivalent of n, for 0 < n < 1,000 */
public static String anglicizeHundreds(int n)
    { return "" + n; }

```

Figure 2.7: Function `anglicize` with stubbed-in function `anglicizeHundreds`

Activity 2-5.4.
The activity
gives the complete develop-
ment, in a way
that is impossi-
ble on paper.

all of which are short and almost trivial. It could be called a *functional approach*, since there are no assignments and all the work is done using function calls (and the if- and return statements). It is an effective manner of programming, and entire programming languages (e.g. LISP and Scheme) have been built on it.

It is possible to use a *procedural approach* to solve this problem, ending up with a program that has fewer functions and whose main function contains some local variables that are being manipulated. We outline this approach here.

To set the stage, consider the following. When summing a list of values, we would initialize a variable *x* (say) to 0 and then, one by one, add values to *x*.

In the problem of anglicizing *n*, we can imagine initializing a *String* variable *s* to be the empty string "" and then, little by little, appending pieces of the English equivalent to *s*. In doing this, we let an integer variable *k* tell us what remains to be anglicized and appended to *s*.

Thus, we start out with this function:

```
/** = English equivalent of n, for 0 < n < 1,000,000 */
public static String anglicize(int n) {
    // anglicize(n) = s + (the English equivalent of k)
    int k= n;      // the part of n left to translate
    String s= "";  // the translation so far

    Reduce k to 0, keeping the definition of s and k true;
    return s;
}
```

We cannot overemphasize the importance of the definition of *s* and *k* as:

anglicize(n) = s + (the English equivalent of k)

In developing the method body, this definition will be the key. Note, for example, how the statement before the return statement refers to this definition. Assuming that the English equivalent of 0 is "", this is a correct description of the task. We can refine the statement to reduce *k* to 0 as follows. The first step, as it was in the functional approach, is to deal with the case when $k \geq 1000$. We refine *anglicize* to deal with this case:

```
/** = English equivalent of n, for 0 < n < 1,000,000 */
public static String anglicize(int n) {
    // anglicize(n) = s + (the English equivalent of k)
    int k= n;      // the part of n left to translate
    String s= "";  // the translation so far

    // Handle the part that is ≥ 1000
    if (k >= 1000) {
        s= s + (English equivalent of k / 1000) + " thousand ";
        k= k % 1000;
    }
}
```

```

// { k < 1000 }
Reduce k to 0, keeping the definition of s and k true;
return s;
}

```

In making this refinement, we have made progress. The statement to reduce k can now assume that $k < 1000$. And, because of the way the statement to reduce k is written, the program is still correct.

To test what we have done so far, we can replace the expression “English equivalent of $k / 1000$ ” by $(k / 1000)$. We should call the function several times, perhaps with several boundary-case arguments — 999999, 2000, 1001, 1000, 999, and 1 — and make sure we get the expected answers back. We can then proceed to refine “English equivalent of $k / 1000$ ”.

We stop the development of this algorithm here because the main ideas have been illustrated. Activity 2-5.4 on the CD explains the development far better than can be done on paper.

Summary of the functional and procedural approaches

The functional approach emphasizes the use of function calls and de-emphasizes the use of local variables and assignment statements. The procedural approach makes heavy use of variables and assignments and creates far fewer functions. Which method you prefer is a matter of taste — and perhaps your previous programming experiences. If you prefer one over the other, make a conscious effort to practice the other so that you become adept at both approaches. Then, you can use whichever is more preferable in any given situation.

2.6 Assertions in programs

Style Note
13.2, 13.2.2
assertions

A program usually contains comments to help the reader understand it. Some comments explain what a program segment does. Other comments describe relationships between variables of the program. Here are two examples of the latter type of comment:

```

// { x < y }
// { n is the number of values read in so far }

```

In this section, we study such comments. We begin by studying the notion of a *relation*.

2.6.1 Relations about variables and values

Activity
1-6.1

A *relation* is simply a true-false statement about some variables. For example, the relation $2 < 3$ is a true statement, which happens not to mention any variables at all, while the relation $2 = 3$ is a false statement. The relation $2 < x$ concerns the single variable x ; we cannot tell whether it is true or false until we know what

value x is associated with. If x contains 7, the relation is true; if x contains 2, the relation is false. Here is a more complex relation concerning variables x , y , and z : $x = y + z$.

The relations given so far were in mathematical notation. Java boolean expressions are also relations, and relations can also be written in a natural language. For example, here are some relations, using `int` variables x , y , z and `String` variable s :

Variable s contains the character 'g'
 The number of characters in s is $2*y$
 The temperature in Ithaca got below x on 1 January 1999
 x is the number of values read in so far

The first relation concerns variable s . The second is a relation between variables s and y . The third is a relation about variable x . The fourth could be about a variable x in a particular Java program.

Note that the mathematical relation $b = c$ is written in Java as $b == c$. When discussing a program, we rely on mathematical notation; when we have to write such a relation in Java, we have no recourse but to use the bad notation.

Simplifying relations

Some relations take this form:

(0) If Bill has black hair, `blackHair` is true.

What is often meant is this:

(1) If Bill has black hair, `blackHair` is true; else, `blackHair` is false.

But these two relations mean different things. The first does not say what value `blackHair` has if Bill has red hair, while the second says that if Bill has red hair, `blackHair` is false. Some might say that it is implicit in (0) that `blackHair` is false if Bill has red hair, but mathematical convention disagrees.

There is a much simpler alternative for the second relation. It is shorter and doesn't have any case analysis:

(2) `blackHair = "Bill has black hair"`

We have placed quotes around the sub-relation to make clear that it is a unit. The quoted phrase is itself a relation; the equality says that `blackHair` is equal to the value of that relation.

Let us show why relations (1) and (2) mean the same thing. In the case that Bill does have black hair, (1) reduces to "`blackHair` is true", while (2) reduces to "`blackHair = true`". These two are equivalent, so in the case that Bill has black hair, (1) and (2) mean the same thing.

In the case that Bill does not have black hair, relation (1) reduces to "`blackHair` is **false**", while relation (2) reduces to "`blackHair = false`". Again, these two are equivalent, so in the case that Bill does not have black hair, (1) and

(2) mean the same thing.

Since (1) and (2) mean the same thing whether or not Bill has black hair, they mean the same thing.

It takes time to get used to writing relations using form (2) instead of (1). Do the self-help exercises at the end of this section to make the transition easier.

Examples

Here are more examples in which using the value of a relation provides a better alternative. This statement stores the value of relation $x < y$ in variable v :

```
if (x < y) { b= true; }
else { b= false; }
```

Instead of this if-statement, you can use this assignment:

```
b= x < y;
```

Now consider this specification of a function:

Return **true** if x is less than y and **false** otherwise.

Instead, write more simply:

Return $x < y$.

A footnote on lesson page 1-6 has a hilarious example of ambiguity.

Here, the value of the relation $x < y$, which is written as a Java boolean expression, is to be returned.

One goal of the earlier part of this discussion is to make clear to you that English is ambiguous. Computer programs should never be ambiguous: when writing a specification of part of a program, be alert to any possible ambiguities and do your best to remove them. Your teammates will love you for it.

2.6.2 Assertions

Activity
1-6.3

Below, the assignment is preceded and followed by a comment:

```
// {x > 0}
x= x + 1;
// {x > 1}
```

Each comment is a relation enclosed in braces $\{$ and $\}$. We call such a relation an *assertion* because we are asserting something about the state of execution whenever the place where the comment appears is reached. Here is how to read this code:

The code says nothing about what the statement does if $x \leq 0$! It deals only with $x > 0$. And, if x is greater than 0 initially, when the statement terminates, x will be greater than 1.

Style Note
13.2, 13.2.2
indenting
assertions

An assertion that precedes a statement is called a *precondition* of the statement. An assertion that follows a statement is called a *postcondition*. Such a precondition-statement-postcondition triple has the following meaning:

Execution of the statement begun with the precondition true is guaranteed to terminate, and when it terminates, the postcondition will be true.

When you see a relation enclosed in braces within a comment, you should assume that the program author is asserting that that relation is true at that point. The earlier code ensures that the relation is true; the later code relies on it being true. In later sections, you will see hints from time to time about when and where to use such assertions.

2.7 A model of execution

We now show you precisely how a method call is executed. Learning this material, and being able to execute method calls yourself using our model of execution, will make writing programs much easier. You will *know* what is going on inside the computer. Further, from time to time you will want to execute a method call by hand in order to pinpoint a difficult-to-find error in your program.

This section requires knowledge of classes and our view of a class as a file drawer of manila folders.

2.7.1 Frames for method calls

Activity 4-3.3 gives a pictorial description of this complete subsection.

Whenever a method is called, some memory is set aside to contain information related to the call: parameter values, the current statement being executed, and so on. This memory is called the *frame for the call*. Figure 2.8 shows the format of the frame that we use throughout this text. We discuss its components:

- **Method-name.** The method name appears in a box in the upper-left of the frame.

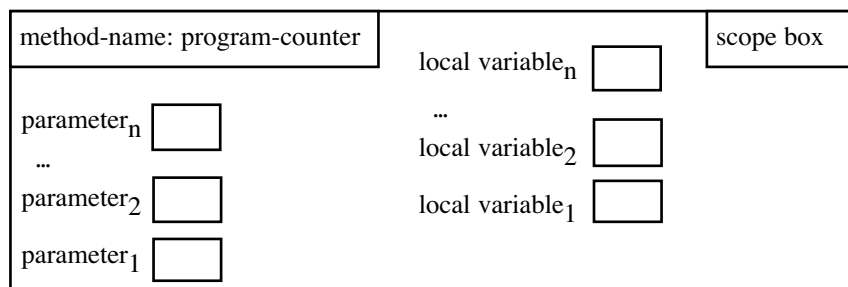


Figure 2.8: Format of the frame for a method call

- **Program counter.** The program counter is the number of the next statement of the method body to execute. Initially, it is 1, and it is incremented each time a statement is executed.
- **Scope box.** The scope box is used to find a variable or method that is referenced in the method body. Its value depends on what kind of method this is:
 - for a static method:** The name of the class in which the method is defined.
 - for an instance method:** The name of the object in which the method resides.
 - for a constructor:** The name of the object that was just created.
- **Parameters.** Each parameter of the method appears as a variable. Parameters are drawn in the lower left of the frame for reasons that will become clear later.
- **Local variables.** Each local variable of the method appears in the frame.

As an example, consider this class:

```
public class C {
    public void meth(int p) {
        double d;
        ...
    }
}
```

Suppose we execute these statements:

```
C c= new C();
c.meth(5);
```

Execution of the first statement creates a new folder, stores its name in variable `c`, and calls procedure `c.meth`. Figure 2.9 shows variable `c`, the folder, and the frame for the method call `c.meth` just after the frame is created and the argument is stored in the parameter.

The call stack: the stack of frames for uncompleted method calls

A frame for a method call lasts as long as the method call is being executed. When the call is finished, the frame is erased. If the method is called again later, a new frame is created for it.

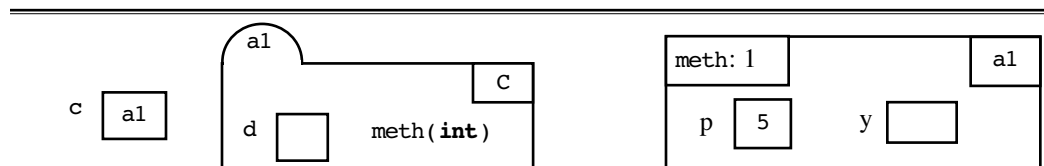


Figure 2.9: The frame just after the argument has been assigned to parameter `p`

This fact explains why local variables do not retain their values from one call of a method to the next call of the same method: all the information about the first call is in a frame, and the frame is erased when the call is completed.

Suppose a call of method `m1` is being executed so that a frame for the call exists. Suppose also that `m1` calls `m2`. A frame for the call is created so that there are now two frames. If `m2` now calls a method `m3`, there will be three frames.

Of the three frames, the frame for `m3` will be erased first, then the frame for `m2`, and finally the frame for `m1`. This is because the call to `m3` is the first to complete. The last frame to be created is the first to be erased, and the first frame to be created is the last to be erased.

The creation and destruction of frames follows a last-in-first-out, or LIFO discipline, and the frames may be maintained on what is called a *stack*.

A stack is a list of items with two operations for changing the list: *pushing an item* onto the stack (inserting a new item on its top) and *popping an item* from the top (removing the topmost item).

As an example of a stack, consider the stack of trays in a cafeteria. An employee will load a large number of trays onto the top of the stack; then, people take them off the top one at a time. The last one added by the employee is the first one removed by a customer.

We show how a stack is used to maintain the frames for a series of calls. Consider class `X` of Fig. 2.10, which contains two static methods. Assume that a call `printLarger2(20, 6)` is about to be executed and that the call appears in a method `m`. At this point, the stack of frames is as shown in the first (leftmost) diagram in Fig. 2.11. The second diagram in Fig. 2.11 shows the situation just after the frame for the call to `printLarger` has been created and the arguments have been assigned to the parameters.

The first statement in the body of `printLarger2` is a call to procedure `printLarger`, so a frame for this call is created and pushed onto the stack of frames. The third diagram in Fig. 2.11 shows the situation after this frame has

```
public class X {
    /** Print the larger of b and c */
    public static void printLarger(int b, int c) {
        if (b >= c) { System.out.println(b); }
        else { System.out.println(c); }
    }

    /** Print larger of b and c and larger of b and c * c */
    public static void printLarger2(int b, int c) {
        printLarger(b, c);
        printLarger(b, c * c);
    }
}
```

Figure 2.10: Class `X` with two **static** procedures

been created and the arguments have been stored in the parameters. When this call is completed, the frame is erased, and the situation is as shown in the third diagram again, but with the program counter changed to 2 to indicate that the statement 2 of the method body is to be executed next.

The *active frame*, the frame for the call, whose body is being executed, is always at the top of the stack. The frames below the top one are *inactive*.

Placing argument values on the call stack

Above, we simply said that the argument values are assigned to the parameters. But how is this done? The arguments are evaluated before the frame for the call is created, and the calling side, the method body where the call is made, knows nothing about how big the frame for the call will be.

Here is how it works. The argument values are pushed onto the call stack, simply as values. When the frame is created, the locations containing these values become the parameters. Thus, the call stack itself acts as the communication device for argument values.

Figure 2.12 illustrates this “parameter passing” mechanism. It shows the call stack just before the first call to procedure `printLarger2`, after the arguments have been placed on the call stack, and then again after the frame has been created. Notice that in the middle picture, the values are not yet named.

The return value of a function

The call stack is also used for communicating the value of a function to the caller. When a statement `return e;` is executed, the active frame is popped from the call stack and the value of `e` is pushed onto the call stack. At the place where the function was called, the value is popped from the call stack and used as the value of the function call.

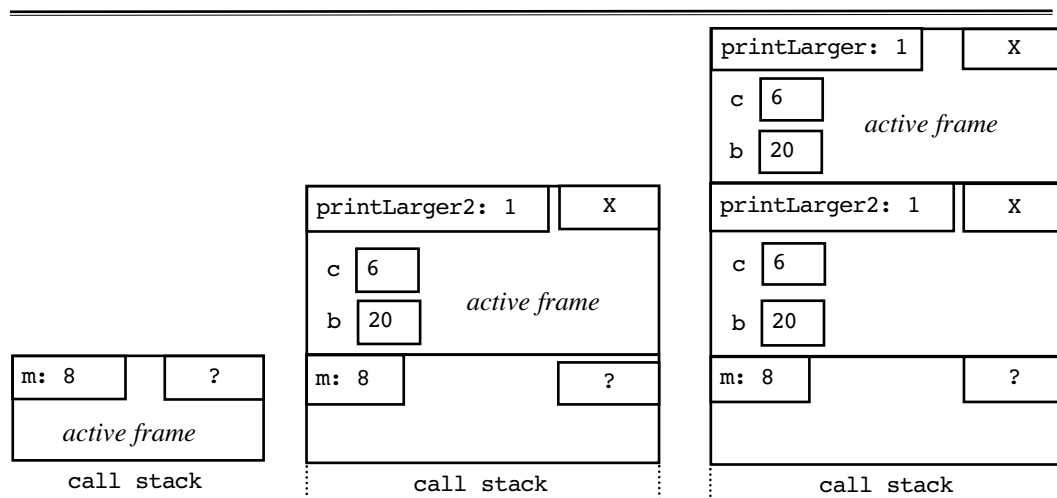


Figure 2.11 The call stack: the stack of frames for uncompleted calls

2.7.2 The steps in executing a method call

Now that we have shown the format of a frame for a method call, discussed the call stack of frames, and discussed the communication of argument and result values, we give the sequence of steps for executing a method call. It is wise to memorize them and practice doing them on 5-10 small examples..

1. Evaluate the arguments of the call and push them onto the call stack.
2. Draw a frame for the call at the top of the call stack; the frame includes the argument values at the top of the stack.
 - 2(a) Fill in the name of the method and set the program counter to 1.
 - 2(b) Fill in the scope box with the name of the entity in which the method appears: the name of a folder for a non-static method or constructor, and the name of the class for a static method.
 - 2(c) Draw all local variables of the method body in the frame.
 - 2(d) Label the argument values pushed onto the call stack in the first step with the names of the corresponding parameters.
3. Execute the method body. Whenever a name is referenced, look in the frame for it. If it is not there, look in the item given by the scope box of the frame.
4. Erase the frame —pop it from the stack. If the method is a function and the call is terminated by execution of a return statement **return e**; , push the value of **e** onto the call stack.

2.8 Key concepts

- **Method.** A *method* is a recipe for getting something done or producing a result. There are three kinds of methods in Java: *procedure*, *function*, and *constructor*. Constructors have to do with initializing objects and are covered fully in Chaps. 3 and 4.

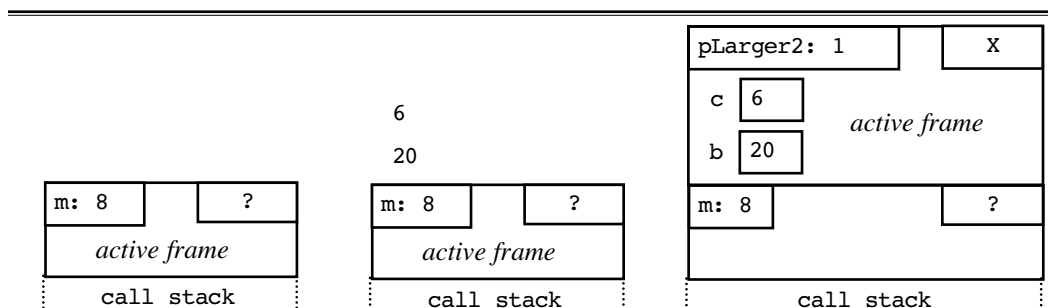


Figure 2.12 The call stack as a communication device for argument values

- **Parameter.** A method may have *parameters*, which are variables that are declared in the header of the method (within the parentheses and separated by commas).
- **Method body.** A method body consists of a sequence of statements. It may also contain declarations of *local variables* to simplify the body and make it more efficient.
- **Assertion.** An assertion is a relation that we place as a comment before or after a statement in a method body to assert that the relation is true at that point. Assertions help programmers understand method bodies.
- **Return statement in a function.** Execution of a function body must terminate by executing a statement `return expression;`; the value of the *expression* is the result of the function call.
- **Method call (or invocation).** A method does nothing until it is called, or invoked, just as a cooking recipe just sits there until someone looks at it and follows the instructions for cooking something. A *procedure call* is a statement; it is executed. A *function call* is an expression; it is evaluated. A *constructor call* is executed; it can appear only in a new-expression.
- **Argument.** Each method call can have *arguments*, which are expressions that appear within the parentheses of a call (and separated by commas).
- **Execution or evaluation of a method call.** Executing or evaluating a call consists of assigning the values of the arguments to the corresponding parameters and then executing the body of the called method. The process of executing the method body stops when there are no more statements to execute or until a return statement is executed.
- **Method specification.** The *specification* of a method gives constraints (pre-conditions) on the parameters of the method and explains precisely *what* it does (or what value it produces). The specification should generally be written before writing the method body. Someone wanting to write a call to the method should be able to do so using only the specification and header of the method.
- **Static versus non-static.** Method definitions (except constructor definitions) may have the modifier **static**. A *static method* is placed in the file drawer for the class in which the method definition appears. A *non-static method* is called an *instance method* because a copy of it is placed in each folder (or instance, or object) of the class in which the method definition appears.
- **Stepwise refinement or top-down programming.** Stepwise refinement is an idealized approach used to develop a method. The process starts with a specification and refines it, step by step, into the final program.
- **An execution model.** Our model of execution of a method call is in terms of the call stack of frames for calls that have been started but have not yet been-

completed. Understanding this model is important for overall understanding, and you should practice executing method calls yourself, using this model. If you cannot do it, then it is likely that you do not understand some important details of how a program is executed!

2.9 Self-review exercises

SR1. What is the difference between a static method and a non-static method?

SR2. A function call is an _____; a procedure call is a _____.

SR3. Is the following a function call or a procedure call? `meth(4, 3);`

SR4. Is the following a function definition or a procedure definition?

```
public void method(int b) { ... }
```

SR5. Define *parameter*: _____.

SR6. Define *argument*: _____.

SR7. To figure out what a procedure call `meth(a1, a2)` does, we _____.

SR8. The scope of a parameter is: _____.

SR9. The scope of a local variable is: _____.

SR10. A local variable keeps its value from one call of the method to the next (true or false).

SR11. A function call must terminate with a statement whose syntax is: _____.

SR12. Draw a frame for the call `C.meth(45, 6 + 2);` on method `meth` of the class shown below. Show the state just after the arguments have been assigned to the parameters but before the method body is executed.

```
public class C {
    public static void meth(int b, double c) {
        String s;
        ...
    }
}
```

Answers to Self-review exercises

SR1. A static method is placed in the file drawer for the class in which it is defined. A nonstatic method appears in every instance of the class in which it is defined.

SR2. A function call is an expression, so it has a value. A procedure call is a statement, so it does not have a value.

SR3. It must be a procedure call because it ends in a semicolon.

SR4. The presence of keyword **void** tells you that it is a procedure.

SR5. A parameter is a variable that is declared within the parentheses of a method definition (adjacent parameter declarations are separated by commas).

SR6. An argument is an expression that appears within the parentheses of a method call (adjacent arguments are separated by commas).

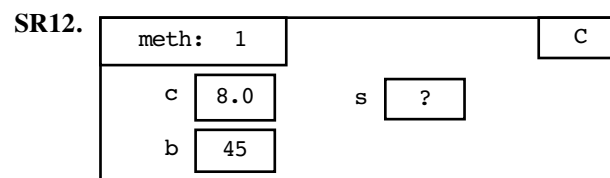
SR7. To figure out what a call `meth(a1, a2)` does, copy the specification of the method and replace all occurrences of the parameter names by the corresponding arguments.

SR8. The scope of a parameter —i.e. where it can be referenced— is the method body.

SR9. The scope of a local variable is the sequence of statements following its declaration (until the end of the block in which the local variable is declared).

SR10. False.

SR11. **return** *expression* ;



Exercises for Chapter 2

Each exercise below asks you to write a method and test it thoroughly. To do this, write a single class `Functions` and place all the functions and procedures in this class. Here is how you can test a method. Suppose you named the function of the first exercise `average`. Then, after writing the function and compiling class `Functions`, type this expression into DrJava's Interactions Pane:

```
Functions.average(3, 5, 7)
```

Make sure you specify your methods, with comments that precede the method definitions. If an exercise asks you to print values, label them suitably on the output. For example, for the above function call, print `average: 5.0`.

E1. Write a function that returns the average of its three **double** arguments.

E2. Write a procedure that prints the average of its three **double** arguments on

the Java console.

E3. Write a procedure that prints the sum, difference, and product of its two **double** parameters.

E4. Write a function that returns "Hooray" if one of its three arguments is less than 10 and returns "Booo" otherwise.

E5. Write a procedure that prints "Hooray" in the Java console if one of its three arguments is less than 10 and prints "Booo" otherwise.

E6. Write a function that computes the area of a circle, given its radius r . The formula for the area is πr^2 . You can get π using `Math.PI`.

E7. Write a procedure that prints the area of a circle, given its radius r . The formula for the area is πr^2 . You can get π using `Math.PI`.

E8. Write a function that, given the number of gallons of gas a tank can hold and the fuel efficiency of the car (miles per gallon), calculates how far the car can go on one tank of gas.

E9. Write a procedure that, given the number of gallons of gas a tank can hold and the fuel efficiency of the car (miles per gallon), prints how far the car can go on one tank of gas.

E10. Write a procedure with three parameters that prints, on separate lines of the Java console, the smallest parameter, then the middle one, and finally the largest.

E11. Write a function that converts its parameter from miles to kilometers. One mile equals 1.60935 kilometers. Use type **double**.

E12. Write a procedure that converts its parameter from kilometers to miles and prints the result. One mile equals 1.60935 kilometers. Use type **double**.

E13. Write a function that converts its parameter from pounds to kilograms. One pound equals 0.45359237 kilograms. Use type **double**.

E14. Write a procedure that converts its parameter from pounds to kilograms and prints the result. One pound equals 0.45359237 kilograms. Use type **double**.

E15. Write a function that is given as parameters the number of hours, minutes, and seconds of a time on a particular day and yields the total number of seconds that the time represents.

E16. Write a function that is given an integer in the range 0..999999 and returns it as a String, with a comma separating the last three digits from the first ones. Of course, include the comma only if the number is at least 1000. For example, for argument 1546, the returned value is "1,546", and for argument 34, the returned value is "34".

E17. Write a procedure that is given a dollar amount and prints out how many

quarters, dimes, nickels, and pennies it takes to make that amount. As many higher-valued coins as possible should be used. For example, for the amount \$4.20, the program should print on one line:

```
16 quarters, 2 dimes, 0 nickels, and 0 pennies
```

E18. Write a function that is given the number of seconds from midnight and returns a `String` that contains the number of hours, minutes, and seconds from midnight that it represents, suitably annotated. For example, called with argument 3675, it yields "1 hour 1 minute 15 seconds".

E19. Write a procedure that is given the number of seconds from midnight and prints the number of hours, minutes, and seconds from midnight that it represents, suitably annotated. For example, called with argument 3675, this should be printed: 1 hour 1 minute 15 seconds.

E20. Write a function that is given a time in terms of hours, minutes, and seconds and returns the time in seconds only (as an integer). This is, in a sense, the inverse of the previous exercise. For example, for 1 hour, 1 minute and 15 seconds, return the integer 3676.

E21. Write a function that is given two times in military format and prints the hours and minutes between the two times. You may assume that the second parameter is the bigger of the two. For example, for the arguments (0352, 1900)—that is 3:52AM and 7:00P—the result is: 15 hours 8 minutes. To make things easier for you, try doing it by calling the function and procedure of the previous two exercises.

E22. Write a function with an `int` parameter whose value is in the range 0..15 and that returns a `String` of length 4 that depicts its binary equivalent. For example, for the number 7, the answer is the `String` "0111". Here is a hint. The rightmost bit is `7%2`, and the first three bits are the binary representation of `7/2`.

E23. Write a procedure with an `int` parameter whose value is in the range 0..15 and that prints its binary equivalent. For example, for the number 7, this should be printed: 0111. Here is a hint. The rightmost bit is `7%2`, and the first three bits are the binary representation of `7/2`.

E24. In Sec. 0.1, near Fig. 0.2, we outlined the relationship between the decimal, hexadecimal, octal, and binary numbers systems. Write a procedure that prints the first 7 natural numbers (0, 1, 2, ..., 8) in all four systems. The first line should contain 0 in all four systems; the second, 1 in all four systems; and so on. For descriptions of static methods to help you do this exercise, turn to lesson page 5-1 of the CD and click on the footnote for static methods of class `Integer` (near activity 3).

E25. Do the same as for the previous exercise, but print the decimal values 21, 22, 23, 24, 25 in each of the four number systems.

E26. Write a function that has three **int** parameters *a*, *b*, and *c* and returns the value of the statement “*a*, *b*, *c* are the lengths of the sides of a triangle”. Three such values form the lengths of the sides of a triangle if and only if $a \leq b + c$, $b \leq c + a$, and $c \leq a + b$.

E27. Write a procedure that has three **int** parameters *a*, *b*, and *c* and returns one of these strings: “equilateral” (meaning the sides have the same length), “isosceles” (only two sides are equal), “triangle” (no sides equal, but they form a triangle; see the previous exercise), or “not a triangle”.

E28. Write a function that has as its parameters the lengths *a*, *b*, and *c* of a triangle and returns the area of the triangle. Let *s* be $1/2$ the perimeter of the triangle. Then the area of the triangle is the square root of the expression $s * (s - a) * (s - b) * (s - c)$.

E29. Write a procedure that has as its parameters the (two) lengths of the sides of a rectangle and prints: the area, the perimeter, and the length of the diagonal.

E30. Write a function with an **int** parameter *n*, with the precondition that $0 \leq n < 1000$, and returns the three digits of the number (with leading zeros if necessary) with a blank between adjacent digits. For example, for argument 43, the string “0 4 3” is returned.

E31. Write a function that is given the coordinates (x_1, y_1) , (x_2, y_2) of two points and returns the distance between them: the square root of $(x_1 - x_2)^2 + (y_1 - y_2)^2$.

E32. Write a function that is given a letter in A..Z and returns the corresponding digit on the telephone. Here is the translation: 2:ABC, 3:DEF, 4:GHI, 5:KHL, 6:MNO, 7:PRS, 8:TUV, and 9:WXY. For Z, return -2, and for Q, return 1.

E33. Turn to lesson page 2-1 of the CD *ProgramLive* and click the Project icon. Do Project Dates, which asks you to write 5-6 functions that manipulate dates in various ways.

E34. Write an if-statement that swaps (exchanges) the values of **int** variables *b* and *c*, if necessary, so that the larger of the two is in *c*.

E35. Is it possible to write the body of procedure `swap`, shown below, so that it is consistent with its specification? For example, if you write a call `swap(d, e);`, is the larger of *d* and *e* guaranteed to be in *e* after the call? If you do not fully understand, do not guess. Instead, write the procedure body and then execute the call to it by hand, using the steps given in Sec. 2.7.2, and see what happens. Explain your answer.

```

    /** Swap the larger of b and c, if necessary, to get the larger in c. */
    public static void swap(int b, int c) {
        ?
    }

```

E36. Consider class `C` shown below. Draw a frame for the call `meth(42, 43)`, showing the state just after the arguments have been assigned to the parameters:

```

public class C {
    public static void meth(int c, int d) { ... }
}

```

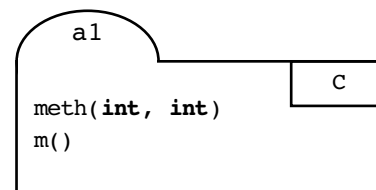
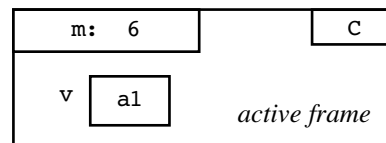
E37. Consider the class `C` shown below.

```

public class C {
    public static void meth(int c, int d) { ... }
    public static void m() { ... }
}

```

Let the active frame be as shown below, and assume that this call appears in method `m`: `v.meth(45, 46);`. Copy the active frame and folder `a1` onto a piece of paper and view the frame as the top frame in the call stack. Then execute the method call, using the steps provided in Sect. 2.7.2 (omit the steps of executing the method body and erasing the call.)





<http://www.springer.com/978-0-387-22681-1>

Multimedia Introduction to Programming Using Java

Gries, D.; Gries, P.

2005, XVIII, 536 p. 420 illus., Softcover

ISBN: 978-0-387-22681-1