chapter 1

primitive java

The primary focus of this book is problem-solving techniques that allow the construction of sophisticated, time-efficient programs. Nearly all of the material discussed is applicable in any programming language. Some would argue that a broad pseudocode description of these techniques could suffice to demonstrate concepts. However, we believe that working with live code is vitally important.

There is no shortage of programming languages available. This text uses Java, which is popular both academically and commercially. In the first four chapters, we discuss the features of Java that are used throughout the book. Unused features and technicalities are not covered. Those looking for deeper Java information will find it in the many Java books that are available.

We begin by discussing the part of the language that mirrors a 1970s programming language such as Pascal or C. This includes primitive types, basic operations, conditional and looping constructs, and the Java equivalent of functions.

In this chapter, we will see

- Some of the basics of Java, including simple lexical elements
- The Java primitive types, including some of the operations that primitive-typed variables can perform

- How conditional statements and loop constructs are implemented in Java
- An introduction to the *static method*—the Java equivalent of the function and procedure that is used in non-object-oriented languages

1.1 the general environment

How are Java application programs entered, compiled, and run? The answer, of course, depends on the particular platform that hosts the Java compiler.

javac compiles `.java` files and generates `.class` files containing bytecode. *java* invokes the Java interpreter (which is also known as the *Virtual Machine*).

Java source code resides in files whose names end with the `.java` suffix. The local compiler, *javac*, compiles the program and generates `.class` files, which contain bytecode. Java *bytecodes* represent the portable intermediate language that is interpreted by running the Java interpreter, *java*. The interpreter is also known as the *Virtual Machine*.

For Java programs, input can come from one of many places:

- The terminal, whose input is denoted as *standard input*
- Additional parameters in the invocation of the Virtual Machine—*command-line arguments*
- A GUI component
- A file

Command-line arguments are particularly important for specifying program options. They are discussed in Section 2.4.5. Java provides mechanisms to read and write files. This is discussed briefly in Section 2.6.3 and in more detail in Section 4.5.3 as an example of the *decorator pattern*. Many operating systems provide an alternative known as *file redirection*, in which the operating system arranges to take input from (or send output to) a file in a manner that is transparent to the running program. On Unix (and also from an MS/DOS window), for instance, the command

```
java Program < inputfile > outputfile
```

automatically arranges things so that any terminal reads are redirected to come from `inputfile` and terminal writes are redirected to go to `outputfile`.

1.2 the first program

Let us begin by examining the simple Java program shown in Figure 1.1. This program prints a short phrase to the terminal. Note the line numbers shown on the left of the code *are not part of the program.* They are supplied for easy reference.

Place the program in the source file `FirstProgram.java` and then compile and run it. Note that the name of the source file must match the name of the class (shown on line 4), including case conventions. If you are using the JDK, the commands are[1]

```
javac FirstProgram.java
java FirstProgram
```

1.2.1 comments

Java has three forms of comments. The first form, which is inherited from C, begins with the token /* and ends with */. Here is an example:

```
/* This is a
   two-line comment */
```

Comments do not nest.

Comments make code easier for humans to read. Java has three forms of comments.

The second form, which is inherited from C++, begins with the token //. There is no ending token. Rather, the comment extends to the end of the line. This is shown on lines 1 and 2 in Figure 1.1.

The third form begins with /** instead of /*. This form can be used to provide information to the *javadoc* utility, which will generate documentation from comments. This form is discussed in Section 3.3.

```
 1 // First program
 2 // MW, 5/1/10
 3
 4 public class FirstProgram
 5 {
 6     public static void main( String [ ] args )
 7     {
 8         System.out.println( "Is there anybody out there?" );
 9     }
10 }
```

figure 1.1

A simple first program

1. If you are using Sun's JDK, *javac* and *java* are used directly. Otherwise, in a typical interactive development environment (IDE), such as Netbeans or Eclipse these commands are executed behind the scenes on your behalf.

Comments exist to make code easier for humans to read. These humans include other programmers who may have to modify or use your code, as well as yourself. A well-commented program is a sign of a good programmer.

1.2.2 `main`

When the program is run, the special method `main` is invoked.

A Java program consists of a collection of interacting classes, which contain methods. The Java equivalent of the function or procedure is the *static method*, which is described in Section 1.6. When any program is run, the special static method `main` is invoked. Line 6 of Figure 1.1 shows that the static method `main` is invoked, possibly with command-line arguments. The parameter types of `main` and the `void` return type shown are required.

1.2.3 terminal output

`println` is used to perform output.

The program in Figure 1.1 consists of a single statement, shown on line 8. `println` is the primary output mechanism in Java. Here, a constant string is placed on the standard output stream `System.out` by applying a `println` method. Input and output is discussed in more detail in Section 2.6. For now we mention only that the same syntax is used to perform output for any entity, whether that entity is an integer, floating point, string, or some other type.

1.3 primitive types

Java defines eight *primitive types*. It also allows the programmer great flexibility to define new types of objects, called *classes*. However, primitive types and user-defined types have important differences in Java. In this section, we examine the primitive types and the basic operations that can be performed on them.

1.3.1 the primitive types

Java's primitive types are integer, floating-point, Boolean, and character.

The Unicode standard contains over 30,000 distinct coded characters covering the principal written languages.

Java has eight primitive types, shown in Figure 1.2. The most common is the integer, which is specified by the keyword `int`. Unlike with many other languages, the range of integers is not machine-dependent. Rather, it is the same in any Java implementation, regardless of the underlying computer architecture. Java also allows entities of types `byte`, `short`, and `long`. These are known as *integral types*. Floating-point numbers are represented by the types `float` and `double`. `double` has more significant digits, so use of it is recommended over use of `float`. The `char` type is used to represent single characters. A `char` occupies 16 bits to represent the Unicode standard. The Unicode standard contains over 30,000 distinct coded characters covering the principal written

Primitive Type	What It Stores	Range
`byte`	8-bit integer	–128 to 127
`short`	16-bit integer	–32,768 to 32,767
`int`	32-bit integer	–2,147,483,648 to 2,147,483,647
`long`	64-bit integer	-2^{63} to $2^{63} - 1$
`float`	32-bit floating-point	6 significant digits (10^{-46}, 10^{38})
`double`	64-bit floating-point	15 significant digits (10^{-324}, 10^{308})
`char`	Unicode character	
`boolean`	Boolean variable	`false` and `true`

figure 1.2
The eight primitive types in Java

languages. The low end of Unicode is identical to ASCII. The final primitive type is `boolean`, which is either `true` or `false`.

1.3.2 constants

Integer constants can be represented in either *decimal, octal,* or *hexadecimal* notation.

Integer constants can be represented in either decimal, octal, or hexadecimal notation. Octal notation is indicated by a leading `0`; hexadecimal is indicated by a leading `0x` or `0X`. The following are all equivalent ways of representing the integer 37: `37`, `045`, `0x25`. Octal integers are not used in this text. However, we must be aware of them so that we use leading `0`s only when we intend to. We use hexadecimals in only one place (Section 12.1), and we will revisit them at that point.

A *string constant* consists of a sequence of characters enclosed by double quotes.

A *character constant* is enclosed with a pair of single quotation marks, as in `'a'`. Internally, this character sequence is interpreted as a small number. The output routines later interpret that small number as the corresponding character. A *string constant* consists of a sequence of characters enclosed within double quotation marks, as in `"Hello"`. There are some special sequences, known as *escape sequences*, that are used (for instance, how does one represent a single quotation mark?). In this text we use `'\n'`, `'\\'`, `'\''`, and `'\"'`, which mean, respectively, the newline character, backslash character, single quotation mark, and double quotation mark.

Escape sequences are used to represent certain character constants.

1.3.3 declaration and initialization of primitive types

A variable is named by using an *identifier*.

Any variable, including those of a primitive type, is declared by providing its name, its type, and optionally, its initial value. The name must be an *identifier*. An identifier may consist of any combination of letters, digits, and the underscore character; it may not start with a digit, however. Reserved words, such as int, are not allowed. Although it is legal to do so, you should not reuse identifier names that are already visibly used (for example, do not use main as the name of an entity).

Java is case-sensitive.

Java is *case-sensitive*, meaning that Age and age are different identifiers. This text uses the following convention for naming variables: All variables start with a lowercase letter and new words start with an uppercase letter. An example is the identifier minimumWage.

Here are some examples of declarations:

```
int num3;                      // Default initialization
double minimumWage = 4.50;     // Standard initialization
int x = 0, num1 = 0;           // Two entities are declared
int num2 = num1;
```

A variable should be declared near its first use. As will be shown, the placement of a declaration determines its scope and meaning.

1.3.4 terminal input and output

Basic formatted terminal I/O is accomplished by nextLine and println. The standard input stream is System.in, and the standard output stream is System.out.

The basic mechanism for formatted I/O uses the String type, which is discussed in Section 2.3. For output, + combines two Strings. If the second argument is not a String, a temporary String is created for it if it is a primitive type. These conversions to String can also be defined for objects (Section 3.4.3). For input, we associate a Scanner object with System.in. Then a String or a primitive type can be read. A more detailed discussion of I/O, including a treatment of formatted files, is in Section 2.6.

1.4 basic operators

This section describes some of the operators available in Java. These operators are used to form *expressions*. A constant or entity by itself is an expression, as are combinations of constants and variables with operators. An expression followed by a semicolon is a simple statement. In Section 1.5, we examine other types of statements, which introduce additional operators.

1.4.1 assignment operators

A simple Java program that illustrates a few operators is shown in Figure 1.3. The basic *assignment operator* is the equals sign. For example, on line 16 the variable a is assigned the value of the variable c (which at that point is 6). Subsequent changes to the value of c do not affect a. Assignment operators can be chained, as in z=y=x=0.

Another assignment operator is the +=, whose use is illustrated on line 18 of the figure. The += operator adds the value on the right-hand side (of the += operator) to the variable on the left-hand side. Thus, in the figure, c is incremented from its value of 6 before line 18, to a value of 14.

Java provides a host of *assignment operators,* including =, +=, -=, *=, and /=.

Java provides various other assignment operators, such as -=, *=, and /=, which alter the variable on the left-hand side of the operator via subtraction, multiplication, and division, respectively.

```
 1 public class OperatorTest
 2 {
 3     // Program to illustrate basic operators
 4     // The output is as follows:
 5     // 12 8 6
 6     // 6 8 6
 7     // 6 8 14
 8     // 22 8 14
 9     // 24 10 33
10
11     public static void main( String [ ] args )
12     {
13         int a = 12, b = 8, c = 6;
14
15         System.out.println( a + " " + b + " " + c );
16         a = c;
17         System.out.println( a + " " + b + " " + c );
18         c += b;
19         System.out.println( a + " " + b + " " + c );
20         a = b + c;
21         System.out.println( a + " " + b + " " + c );
22         a++;
23         ++b;
24         c = a++ + ++b;
25         System.out.println( a + " " + b + " " + c );
26     }
27 }
```

figure 1.3

Program that illustrates operators

1.4.2 binary arithmetic operators

Java provides several *binary arithmetic operators*, including +, -, *, /, and %.

Line 20 in Figure 1.3 illustrates one of the *binary arithmetic operators* that are typical of all programming languages: the addition operator (+). The + operator causes the values of b and c to be added together; b and c remain unchanged. The resulting value is assigned to a. Other arithmetic operators typically used in Java are -, *, /, and %, which are used, respectively, for subtraction, multiplication, division, and remainder. Integer division returns only the integral part and discards any remainder.

As is typical, addition and subtraction have the same precedence, and this precedence is lower than the precedence of the group consisting of the multiplication, division, and mod operators; thus 1+2*3 evaluates to 7. All of these operators associate from left to right (so 3-2-2 evaluates to –1). All operators have precedence and associativity. The complete table of operators is in Appendix A.

1.4.3 unary operators

Several *unary operators* are defined, including -.

Autoincrement and autodecrement add 1 and subtract 1, respectively. The operators for doing this are ++ and --. There are two forms of incrementing and decrementing: prefix and postfix.

In addition to binary arithmetic operators, which require two operands, Java provides *unary operators*, which require only one operand. The most familiar of these is the unary minus, which evaluates to the negative of its operand. Thus -x returns the negative of x.

Java also provides the autoincrement operator to add 1 to a variable—denoted by ++ —and the autodecrement operator to subtract 1 from a variable—denoted by --. The most benign use of this feature is shown on lines 22 and 23 of Figure 1.3. In both lines, the *autoincrement operator* ++ adds 1 to the value of the variable. In Java, however, an operator applied to an expression yields an expression that has a value. Although it is guaranteed that the variable will be incremented before the execution of the next statement, the question arises: What is the value of the autoincrement expression if it is used in a larger expression?

In this case, the placement of the ++ is crucial. The semantics of ++x is that the value of the expression is the new value of x. This is called the *prefix increment*. In contrast, x++ means the value of the expression is the original value of x. This is called the *postfix increment*. This feature is shown in line 24 of Figure 1.3. a and b are both incremented by 1, and c is obtained by adding the *original* value of a to the *incremented* value of b.

1.4.4 type conversions

The *type conversion operator* is used to generate a temporary entity of a new type.

The *type conversion operator* is used to generate a temporary entity of a new type. Consider, for instance,

```
double quotient;
int x = 6;
int y = 10;
quotient = x / y;    // Probably wrong!
```

The first operation is the division, and since x and y are both integers, the result is integer division, and we obtain 0. Integer 0 is then implicitly converted to a `double` so that it can be assigned to `quotient`. But we had intended `quotient` to be assigned 0.6. The solution is to generate a temporary variable for either x or y so that the division is performed using the rules for `double`. This would be done as follows:

```
quotient = ( double ) x / y;
```

Note that neither x nor y are changed. An unnamed temporary is created, and its value is used for the division. The type conversion operator has higher precedence than division does, so x is type-converted and then the division is performed (rather than the conversion coming after the division of two `int`s being performed).

1.5 conditional statements

This section examines statements that affect the flow of control: conditional statements and loops. As a consequence, new operators are introduced.

1.5.1 relational and equality operators

The basic test that we can perform on primitive types is the comparison. This is done using the equality and inequality operators, as well as the relational operators (less than, greater than, and so on).

In Java, the *equality operators* are == and !=. For example,

In Java, the *equality operators* are == and !=.

```
leftExpr==rightExpr
```

evaluates to `true` if `leftExpr` and `rightExpr` are equal; otherwise, it evaluates to `false`. Similarly,

```
leftExpr!=rightExpr
```

evaluates to `true` if `leftExpr` and `rightExpr` are not equal and to `false` otherwise.

The *relational operators* are <, <=, >, and >=. These have natural meanings for the built-in types. The relational operators have higher precedence than the equality operators. Both have lower precedence than the arithmetic operators

The *relational operators* are <, <=, >, and >=.

but higher precedence than the assignment operators, so the use of parentheses is frequently unnecessary. All of these operators associate from left to right, but this fact is useless: In the expression `a<b<6`, for example, the first `<` generates a `boolean` and the second is illegal because `<` is not defined for `booleans`. The next section describes the correct way to perform this test.

1.5.2 logical operators

Java provides logical operators that are used to simulate the Boolean algebra concepts of AND, OR, and NOT. The corresponding operators are &&, ||, and !.

Java provides *logical operators* that are used to simulate the Boolean algebra concepts of AND, OR, and NOT. These are sometimes known as *conjunction*, *disjunction*, and *negation*, respectively, whose corresponding operators are `&&`, `||`, and `!`. The test in the previous section is properly implemented as `a<b && b<6`. The precedence of conjunction and disjunction is sufficiently low that parentheses are not needed. `&&` has higher precedence than `||`, while `!` is grouped with other unary operators (and is thus highest of the three). The operands and results for the logical operators are `boolean`. Figure 1.4 shows the result of applying the logical operators for all possible inputs.

Short-circuit evaluation means that if the result of a logical operator can be determined by examining the first expression, then the second expression is not evaluated.

One important rule is that `&&` and `||` are short-circuit evaluation operations. *Short-circuit evaluation* means that if the result can be determined by examining the first expression, then the second expression is not evaluated. For instance, in

```
x != 0 && 1/x != 3
```

if `x` is 0, then the first half is `false`. Automatically the result of the AND must be `false`, so the second half is not evaluated. This is a good thing because division-by-zero would give erroneous behavior. Short-circuit evaluation allows us to not have to worry about dividing by zero.[2]

figure 1.4

Result of logical operators

x	y	x && y	x || y	!x
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

2. There are (extremely) rare cases in which it is preferable to not short-circuit. In such cases, the `&` and `|` operators with `boolean` arguments guarantee that both arguments are evaluated, even if the result of the operation can be determined from the first argument.

1.5.3 the `if` statement

The `if` statement is the fundamental decision maker. Its basic form is

The `if` statement is the fundamental decision maker.

```
if( expression )
    statement
next statement
```

If `expression` evaluates to `true`, then `statement` is executed; otherwise, it is not. When the `if` statement is completed (without an unhandled error), control passes to the next statement.

Optionally, we can use an `if-else` statement, as follows:

```
if( expression )
    statement1
else
    statement2
next statement
```

In this case, if `expression` evaluates to `true`, then `statement1` is executed; otherwise, `statement2` is executed. In either case, control then passes to the next statement, as in

```
System.out.print( "1/x is " );
if( x != 0 )
    System.out.print( 1 / x );
else
    System.out.print( "Undefined" );
System.out.println( );
```

Remember that each of the `if` and `else` clauses contains at most one statement, no matter how you indent. Here are two mistakes:

```
if( x == 0 );    // ; is null statement (and counts)
    System.out.println( "x is zero " );
else
    System.out.print( "x is " );
    System.out.println( x ); // Two statements
```

The first mistake is the inclusion of the ; at the end of the first `if`. This semicolon by itself counts as the *null statement*; consequently, this fragment won't compile (the `else` is no longer associated with an `if`). Once that mistake is fixed, we have a logic error: that is, the last line is not part of the `else`, even though the indentation suggests it is. To fix this problem, we have to use a *block*, in which we enclose a sequence of statements by a pair of braces:

A semicolon by itself is the *null statement*.

A *block* is a sequence of statements within braces.

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

The `if` statement can itself be the target of an `if` or `else` clause, as can other control statements discussed later in this section. In the case of nested `if-else` statements, an `else` matches the innermost dangling `if`. It may be necessary to add braces if that is not the intended meaning.

1.5.4 the `while` statement

The `while` statement is one of three basic forms of looping.

Java provides three basic forms of looping: the `while` statement, `for` statement, and `do` statement. The syntax for the `while` statement is

```
while( expression )
    statement
next statement
```

Note that like the `if` statement, there is no semicolon in the syntax. If one is present, it will be taken as the null statement.

While `expression` is `true`, `statement` is executed; then `expression` is reevaluated. If `expression` is initially `false`, then `statement` will never be executed. Generally, `statement` does something that can potentially alter the value of `expression`; otherwise, the loop could be infinite. When the `while` loop terminates (normally), control resumes at the next statement.

1.5.5 the `for` statement

The `for` statement is a looping construct that is used primarily for simple iteration.

The `while` statement is sufficient to express all repetition. Even so, Java provides two other forms of looping: the `for` statement and the `do` statement. The `for` statement is used primarily for iteration. Its syntax is

```
for( initialization; test; update )
    statement
next statement
```

Here, `initialization`, `test`, and `update` are all expressions, and all three are optional. If `test` is not provided, it defaults to `true`. There is no semicolon after the closing parenthesis.

The `for` statement is executed by first performing the `initialization`. Then, while `test` is `true`, the following two actions occur: `statement` is performed, and then update is performed. If initialization and update are omitted, then the for statement behaves exactly like a while statement. The advantage of a for statement is clarity in that for variables that count (or iterate), the for statement makes it much easier to see what the range of the counter is. The following fragment prints the first 100 positive integers:

```
for( int i = 1; i <= 100; i++ )
    System.out.println( i );
```

This fragment illustrates the common technique of declaring a counter in the initialization portion of the loop. This counter's scope extends only inside the loop.

Both initialization and update may use a comma to allow multiple expressions. The following fragment illustrates this idiom:

```
for( i = 0, sum = 0; i <= n; i++, sum += n )
    System.out.println( i + "\t" + sum );
```

Loops nest in the same way as if statements. For instance, we can find all pairs of small numbers whose sum equals their product (such as 2 and 2, whose sum and product are both 4):

```
for( int i = 1; i <= 10; i++ )
    for( int j = 1; j <= 10; j++ )
        if( i + j == i * j )
            System.out.println( i + ", " + j );
```

As we will see, however, when we nest loops we can easily create programs whose running times grow quickly.

Java 5 adds an "enhanced" for loop. We discuss this addition in Section 2.4 and Chapter 6.

1.5.6 the do statement

The do statement is a looping construct that guarantees the loop is executed at least once.

The while statement repeatedly performs a test. If the test is true, it then executes an embedded statement. However, if the initial test is false, the embedded statement is never executed. In some cases, however, we would like to guarantee that the embedded statement is executed at least once. This is done using the do statement. The do statement is identical to the while statement, except that the test is performed after the embedded statement. The syntax is

```
do
    statement
while( expression );
next statement
```

Notice that the do statement includes a semicolon. A typical use of the do statement is shown in the following pseudocode fragment:

```
do
{
    Prompt user;
    Read value;
} while( value is no good );
```

The do statement is by far the least frequently used of the three looping constructs. However, when we have to do something at least once, and for some reason a for loop is inappropriate, then the do statement is the method of choice.

1.5.7 break and continue

The for and while statements provide for termination before the start of a repeated statement. The do statement allows termination after execution of a repeated statement. Occasionally, we would like to terminate execution in the middle of a repeated (compound) statement. The break statement, which is the keyword break followed by a semicolon, can be used to achieve this. Typically, an if statement would precede the break, as in

```
while( ... )
{
    ...
    if( something )
        break;
    ...
}
```

The break statement exits the innermost loop or switch statement. The labeled break statement exits from a nested loop.

The break statement exits the innermost loop only (it is also used in conjunction with the switch statement, described in the next section). If there are several loops that need exiting, the break will not work, and most likely you have poorly designed code. Even so, Java provides a labeled break statement. In the labeled break statement, a loop is labeled, and then a break statement can be applied to the loop, regardless of how many other loops are nested. Here is an example:

```
  outer:
    while( ... )
    {
        while( ... )
            if( disaster )
                break outer; // Go to after outer
    }
    // Control passes here after outer loop is exited
```

Occasionally, we want to give up on the current iteration of a loop and go on to the next iteration. This can be handled by using a `continue` statement. Like the `break` statement, the `continue` statement includes a semicolon and applies to the innermost loop only. The following fragment prints the first 100 integers, with the exception of those divisible by 10:

The `continue` statement goes to the next iteration of the innermost loop.

```
for( int i = 1; i <= 100; i++ )
{
    if( i % 10 == 0 )
        continue;
    System.out.println( i );
}
```

Of course, in this example, there are alternatives to the `continue` statement. However, `continue` is commonly used to avoid complicated `if-else` patterns inside loops.

1.5.8 the `switch` statement

The `switch` statement is used to select among several small integer (or character) values. It consists of an expression and a block. The block contains a sequence of statements and a collection of labels, which represent possible values of the expression. All the labels must be distinct compile-time constants. An optional default label, if present, matches any unrepresented label. If there is no applicable case for the `switch` expression, the `switch` statement is over; otherwise, control passes to the appropriate label and all statements from that point on are executed. A `break` statement may be used to force early termination of the `switch` and is almost always used to separate logically distinct cases. An example of the typical structure is shown in Figure 1.5.

The `switch` statement is used to select among several small integer (or character) values.

1.5.9 the conditional operator

The *conditional operator* `?:` is used as a shorthand for simple `if-else` statements. The general form is

The *conditional operator* `?:` is used as a shorthand for simple `if-else` statements.

```
testExpr ? yesExpr : noExpr
```

`testExpr` is evaluated first, followed by either `yesExpr` or `noExpr`, producing the result of the entire expression. `yesExpr` is evaluated if `testExpr` is `true`; otherwise, `noExpr` is evaluated. The precedence of the conditional operator is just above that of the assignment operators. This allows us to avoid using parentheses when assigning the result of the conditional operator to a variable. As an example, the minimum of `x` and `y` is assigned to `minVal` as follows:

```
minVal = x <= y ? x : y;
```

figure 1.5

Layout of a switch statement

```
switch( someCharacter )
{
  case '(':
  case '[':
  case '{':
    // Code to process opening symbols
    break;

  case ')':
  case ']':
  case '}':
    // Code to process closing symbols
    break;

  case '\n':
    // Code to handle newline character
    break;

  default:
    // Code to handle other cases
    break;
}
```

1.6 methods

A *method* is similar to a function in other languages. The *method header* consists of the name, return type, and parameter list. The *method declaration* includes the body.

What is known as a function or procedure in other languages is called a *method* in Java. A more complete treatment of methods is provided in Chapter 3. This section presents some of the basics for writing functions, such as `main`, in a non-object-oriented manner (as would be encountered in a language such as C) so that we can write some simple programs.

A *method header* consists of a name, a (possibly empty) list of parameters, and a return type. The actual code to implement the method, sometimes called the *method body*, is formally a *block*. A *method declaration* consists of a header plus the body. An example of a method declaration and a `main` routine that uses it is shown in Figure 1.6.

A `public static` method is the equivalent of a "C-style" global function.

By prefacing each method with the words `public static`, we can mimic the C-style global function. Although declaring a method as `static` is a useful technique in some instances, it should not be overused, since in general we do not want to use Java to write "C-style" code. We will discuss the more typical use of `static` in Section 3.6.

In *call-by-value*, the actual arguments are copied into the formal parameters. Variables are passed using call-by-value.

The method name is an identifier. The parameter list consists of zero or more *formal parameters*, each with a specified type. When a method is called, the *actual arguments* are sent into the formal parameters using normal

```java
public class MinTest
{
    public static void main( String [ ] args )
    {
        int a = 3;
        int b = 7;

        System.out.println( min( a, b ) );
    }

    // Method declaration
    public static int min( int x, int y )
    {
        return x < y ? x : y;
    }
}
```

figure 1.6

Illustration of method declaration and calls

assignment. This means primitive types are passed using *call-by-value* parameter passing only. The actual arguments cannot be altered by the function. As with most modern programming languages, method declarations may be arranged in any order.

The return statement is used to return a value to the caller.

The `return` statement is used to return a value to the caller. If the return type is `void`, then no value is returned, and `return;` should be used.

1.6.1 overloading of method names

Suppose we need to write a routine that returns the maximum of three `int`s. A reasonable method header would be

```java
int max( int a, int b, int c )
```

In some languages, this may be unacceptable if `max` is already declared. For instance, we may also have

```java
int max( int a, int b )
```

Overloading of a method name means that several methods may have the same name as long as their parameter list types differ.

Java allows the *overloading* of method names. This means that several methods may have the same name and be declared in the same class scope as long as their *signatures* (that is, their parameter list types) differ. When a call to `max` is made, the compiler can deduce which of the intended meanings should be applied based on the actual argument types. Two signatures may have the same number of parameters, as long as at least one of the parameter list types differs.

Note that the return type is not included in the signature. This means it is illegal to have two methods in the same class scope whose only difference is the return type. Methods in different class scopes may have the same names, signatures, and even return types; this is discussed in Chapter 3.

1.6.2 storage classes

Entities that are declared inside the body of a method are local variables and can be accessed by name only within the method body. These entities are created when the method body is executed and disappear when the method body terminates.

static final variables are constants.

A variable declared outside the body of a method is global to the class. It is similar to global variables in other languages if the word `static` is used (which is likely to be required so as to make the entity accessible by static methods). If both `static` and `final` are used, they are global symbolic constants. As an example,

```
static final double PI = 3.1415926535897932;
```

Note the use of the common convention of naming symbolic constants entirely in uppercase. If several words form the identifier name, they are separated by the underscore character, as in `MAX_INT_VALUE`.

If the word `static` is omitted, then the variable (or constant) has a different meaning, which is discussed in Section 3.6.5.

summary

This chapter discussed the primitive features of Java, such as primitive types, operators, conditional and looping statements, and methods that are found in almost any language.

Any nontrivial program will require the use of nonprimitive types, called *reference types*, which are discussed in the next chapter.

key concepts

assignment operators In Java, used to alter the value of a variable. These operators include =, +=, -=, *=, and /=. (9)

autoincrement (++) and **autodecrement (--) operators** Operators that add and subtract 1, respectively. There are two forms of incrementing and decrementing prefix and postfix. (10)

binary arithmetic operators Used to perform basic arithmetic. Java provides several, including +, -, *, /, and %. (10)

block A sequence of statements within braces. (13)

break statement A statement that exits the innermost loop or `switch` statement. (16)

bytecode Portable intermediate code generated by the Java compiler. (4)

call-by-value The Java parameter-passing mechanism whereby the actual argument is copied into the formal parameter. (18)

comments Make code easier for humans to read but have no semantic meaning. Java has three forms of comments. (5)

conditional operator (?:) An operator that is used in an expression as a shorthand for simple `if-else` statements. (17)

continue statement A statement that goes to the next iteration of the innermost loop. (17)

do statement A looping construct that guarantees the loop is executed at least once. (15)

equality operators In Java, `==` and `!=` are used to compare two values; they return either `true` or `false` (as appropriate). (11)

escape sequence Used to represent certain character constants. (7)

for statement A looping construct used primarily for simple iteration. (14)

identifier Used to name a variable or method. (7)

if statement The fundamental decision maker. (13)

integral types `byte`, `char`, `short`, `int`, and `long`. (6)

java The Java interpreter, which processes bytecodes. (4)

javac The Java compiler; generates bytecodes. (4)

labeled break statement A `break` statement used to exit from nested loops. (16)

logical operators &&, `||`, and `!`, used to simulate the Boolean algebra concepts of AND, OR, and NOT. (12)

main The special method that is invoked when the program is run. (6)

method The Java equivalent of a function. (18)

method declaration Consists of the method header and body. (18)

method header Consists of the name, return type, and parameter list. (18)

null statement A statement that consists of a semicolon by itself. (13)

octal and hexadecimal integer constants Integer constants can be represented in either decimal, octal, or hexadecimal notation. Octal notation is indicated by a leading `0`; hexadecimal is indicated by a leading `0x` or `0X`. (7)

overloading of a method name The action of allowing several methods to have the same name as long as their parameter list types differ. (19)

primitive types In Java, integer, floating-point, Boolean, and character. (6)

relational operators In Java, <, <=, >, and >= are used to decide which of two values is smaller or larger; they return `true` or `false`. (11)

`return` statement A statement used to return information to the caller. (19)

short-circuit evaluation The process whereby if the result of a logical operator can be determined by examining the first expression, then the second expression is not evaluated. (12)

signature The combination of the method name and the parameter list types. The return type is not part of the signature. (18)

standard input The terminal, unless redirected. There are also streams for standard output and standard error. (4)

`static final` entity A global constant. (20)

`static` method Occasionally used to mimic C-style functions; discussed more fully in Section 3.6. (18)

string constant A constant that consists of a sequence of characters enclosed by double quotes. (7)

`switch` statement A statement used to select among several small integral values. (17)

type conversion operator An operator used to generate an unnamed temporary variable of a new type. (10)

unary operators Require one operand. Several unary operators are defined, including unary minus (-) and the autoincrement and autodecrement operators (++ and --). (10)

Unicode International character set that contains over 30,000 distinct characters covering the principle written languages. (6)

`while` statement The most basic form of looping. (14)

Virtual Machine The bytecode interpreter. (4)

common errors

1. Adding unnecessary semicolons gives logical errors because the semicolon by itself is the null statement. This means that an unintended semicolon immediately following a `for`, `while`, or `if` statement is very likely to go undetected and will break your program.
2. At compile time, the Java compiler is required to detect all instances in which a method that is supposed to return a value fails to do so. Occasionally, it provides a false alarm, and you have to rearrange code.
3. A leading `0` makes an integer constant octal when seen as a token in source code. So `037` is equivalent to decimal `31`.