Part I

Tour of Java

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 programing languages available. At the time of this writing, C++ is the language in widest use both academically and commercially. However, in 1996, Java exploded onto the scene as a viable contender.

Java's primary appeal is that it is a safe, portable language that supports modern object-oriented constructs. Many C++ constructs that are confusing to novices are not found in Java. Compared to C++, many common programming errors are caught by Java either at compile time or at run time. Java has an exception mechanism that requires the programmer to explicitly deal with errors and a relatively simple model that distinguishes between primitive types (such as `int`) and user-defined types. Java does not have an explicit pointer type.

Java is portable: for example, an integer has the same range of values in every Java implementation, regardless of the underlying computer architecture. Java also provides a graphical user interface (GUI) toolkit that allows input and output

to be performed using forms. Although we do not discuss this toolkit in the text, it is relatively easy to use. Most important, it is also portable to every Java implementation. Java's philosophy is "write once, run everywhere."

In the first five 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 1970's 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.

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*.

javac compiles `.java` files and generates `.class` files containing *bytecode*. *java* invokes the Java interpreter (which 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 executable program — *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, 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 up to the end of the line. This is shown on lines 1 and 2 in Figure 1.1.

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

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.

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 // First program
 2 // MW, 9/1/01
 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

1.2.2 `main`

When the program is run, the special function `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.

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

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

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

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. Hexadecimals are used in only one place (Section 12.1), and we will revisit them at that point.

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.

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

Escape sequences are used to represent certain character constants.

1.3.3 Declaration and Initialization of Primitive Types

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).

A variable is named by using an *identifier*.

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`.

Java is case-sensitive.

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 `readLine` 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 `String`s. 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 must associate a `BufferedReader` object with `System.in`. Then a `String` is read and can be parsed. 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`.

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

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 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 `-`.

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`.

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.

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 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

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

`boolean` and the second is illegal because `<` is not defined for `boolean`s. 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. 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. Inputs and outputs for the logical operators are `boolean`. Figure 1.4 shows the result of applying the logical operators for all possible inputs.

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

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

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.

```
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]

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

Figure 1.4 Result of logical operators

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
```

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.

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 at each of the `if` and `else` clauses contain 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
```

A semicolon by itself is the *null statement*.

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 *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

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

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

```
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: `for` statement and `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.

1.5.6 The do Statement

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

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

```
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 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:

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

```
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.

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.

1.5.9 The Conditional Operator

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

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

```
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 opera-

tor 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;
```

```
 1 switch( someCharacter )
 2 {
 3   case '(':
 4   case '[':
 5   case '{':
 6     // Code to process opening symbols
 7     break;
 8
 9   case ')':
10   case ']':
11   case '}':
12     // Code to process closing symbols
13     break;
14
15   case '\n':
16     // Code to handle newline character
17     break;
18
19   default:
20     // Code to handle other cases
21     break;
22 }
```

Figure 1.5 Layout of a `switch` statement

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.

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

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

Variables are passed using call-by-value.

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 C-like functions, such as `main`, 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.

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 3.5.

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 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.

```
 1 public class MinTest
 2 {
 3     public static void main( String [ ] args )
 4     {
 5         int a = 3;
 6         int b = 7;
 7
 8         System.out.println( min( a, b ) );
 9     }
10
11     // Method declaration
12     public static int min( int x, int y )
13     {
14         return x < y ? x : y;
15     }
16 }
```

Figure 1.6 Illustration of method declaration and calls

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.

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

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

```
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

```
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 3.4.6.

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.

Objects of the Game

assignment operators In Java, used to alter the value of a variable. These operators include `=`, `+=`, `-=`, `*=`, and `/=`. (13)

autoincrement (++) and autodecrement (--) operators Operators that add and subtract 1, respectively. There are two forms of incrementing and decrementing, prefix and postfix. (15)

binary arithmetic operators Used to perform basic arithmetic. Java provides several, including `+`, `-`, `*`, `/`, and `%`. (14)

block A sequence of statements within braces. (20)

`break` statement A statement that exits the innermost loop or `switch` statement. (25)

bytecode Portable intermediate code generated by the Java compiler. (5)

call-by-value The Java parameter-passing mechanism whereby the actual argument is copied into the formal parameter. (28)

comments Make code easier for humans to read but have no semantic meaning. Java has three forms of comments. (6)

conditional operator (`?:`) An operator that is used in an expression as a shorthand for simple `if-else` statements. (26)

`continue` statement A statement that goes to the next iteration of the innermost loop. (25)

`do` statement A looping construct that guarantees the loop is executed at least once. (23)

equality operators In Java, `==` and `!=` are used to compare two values; they return either `true` or `false` (as appropriate). (17)

escape sequence Used to represent certain character constants. (11)

`for` statement A looping construct used primarily for simple iteration. (22)

identifier Used to name a variable or method. (11)

`if` statement The fundamental decision maker. (19)

integral types `byte`, `char`, `short`, `int`, and `long`. (9)

java The java interpreter, which processes bytecodes. (5)

javac The java compiler; generates bytecodes. (5)

labeled `break` statement A `break` statement used to exit from nested loops. (25)

logical operators &&, `||`, and `!`, used to simulate the Boolean algebra concepts of AND, OR, and NOT. (18)

`main` The special function that is invoked when the program is run. (7)

method The Java equivalent of a function. (28)

method declaration Consists of the method header and body. (28)

method header Consists of the name, return type, and parameter list. (28)

null statement A statement that consists of a semicolon by itself. (20)

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`. (10)

overloading of a method name The action of allowing several methods to have the same name as long as their parameter list types differ. (30)

primitive types In Java, integer, floating-point, Boolean, and character. (9)

relational operators In Java, `<`, `<=`, `>`, and `>=` are used to decide which of two values is smaller or larger; they return `true` or `false`. (17)

`return` statement A statement used to return information to the caller. (29)

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. (18)

signature The combination of the method name and the parameter list types. The return type is not part of the signature. (30)

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

`static final` entity A global constant. (30)

`static` method Occasionally used to mimic C-style functions; discussed more fully in Section 3.5. (29)

string constant A constant that consists of a sequence of characters enclosed by double quotes. (11)

`switch` statement A statement used to select among several small integral values. (26)

type conversion operator An operator used to generate an unnamed temporary variable of a new type. (16)

unary operators Require one operand. Several unary operators are defined, including unary minus (`-`) and the autoincrement and autodecrement operators (`++` and `--`). (15)

Unicode International character set that contains over 30,000 distinct characters covering the principle written languages. (9)

`while` statement The most basic form of looping. (21)

virtual machine The bytecode interpreter. (5)

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`.
4. Use `&&` and `||` for logical operations; `&` and `|` do not short circuit.
5. The `else` clause matches the closest dangling `if`. It is common to forget to include the braces needed to match the `else` to a distant dangling `if`.
6. When a `switch` statement is used, it is common to forget the `break` statement between logical cases. If it is forgotten, control passes through to the next case; generally, this is not the desired behavior.
7. Escape sequences begin with the backslash `\`, not the forward slash `/`.
8. Mismatched braces may give misleading answers. Use `Balance`, described in Section 11.1, to check if this is the cause of a compiler error message.
9. The name of the Java source file must match the name of the class being compiled.

On the Internet

Following are the available files for this chapter. Everything is self-contained, and nothing is used later in the text.

FirstProgram.java	The first program, as shown in Figure 1.1
MinTest.java	Illustration of methods, as shown in Figure 1.6.
OperatorTest.java	Demonstration of various operators, as shown in Figure 1.3.

Exercises

In Short

1.1. What extensions are used for Java source and compiled files?

1.2. Describe the three kinds of comments used in Java programs.

1.3. What are the eight primitive types in Java?

1.4. What is the difference between the `*` and `*=` operators?

1.5. Explain the difference between the prefix and postfix increment operators.

1.6. Describe the three types of loops in Java.

1.7. Describe all the uses of a `break` statement. What is a labeled `break` statement?

1.8. What does the `continue` statement do?

1.9. What is method overloading?

1.10. Describe call-by-value.

In Theory

1.11. Let `b` have the value of 5 and `c` have the value of 8. What is the value of `a`, `b`, and `c` after each line of the following program fragment:

```
a = b++ + c++;
a = b++ + ++c;
a = ++b + c++;
a = ++b + ++c;
```

1.12. What is the result of `true && false || true`?

1.13. For the following, give an example in which the `for` loop on the left is not equivalent to the `while` loop on the right:

```
for( init; test; update )
{
    statements

}
```

```
init;
while( test )
{
    statements
    update;
}
```

1.14. For the following program, what are the possible outputs:

```
public class WhatIsX
{
    public static void f( int x )
      { /* body unknown */ }

    public static void main( String [ ] args )
    {
        int x = 0;
        f( x );
        System.out.println( x );
    }
}
```

In Practice

1.15. Write a `while` statement that is equivalent to the following `for` fragment. Why would this be useful?

```
for( ; ; )
    statement
```

1.16. Write a program to generate the addition and multiplication tables for single-digit numbers (the table that elementary school students are accustomed to seeing).

1.17. Write two static methods. The first should return the maximum of three integers, and the second should return the maximum of four integers.

1.18. Write a static method that takes a year as a parameter and returns `true` if the year is a leap year, and `false` otherwise.

Programming Projects

1.19. Write a program to determine all pairs of positive integers, (a, b), such that $a < b < 1000$ and $(a^2 + b^2 + 1)/(ab)$ is an integer.

1.20. Write a method that prints the representation of its integer parameter as a Roman numeral. Thus, if the parameter is `1998`, the output is `MCMXCVIII`.

1.21. Suppose you want to print out numbers in brackets, formatted as follows: `[1][2][3]`, and so on. Write a method that takes two parameters: `howMany` and `lineLength`. The method should print out line numbers from 1 to `howMany` in the previous format, but it should not output more than `lineLength` characters on any one line. It should not start a `[` unless it can fit the corresponding `]`.

1.22. In the following decimal arithmetic puzzle, each of the ten different letters is assigned a digit. Write a program that finds all possible solutions (one of which is shown).

```
   MARK     A=1 W=2 N=3 R=4 E=5       9147
+ ALLEN     L=6 K=7 I=8 M=9 S=0    + 16653
  -----                              -----
  WEISS                             25800
```

References

Some of the C-style material in this chapter is taken from [5]. The complete Java language specification may be found in [3]. Introductory Java books include [1], [2] and [4].

1. G. Cornell and C. S. Horstmann, *Core Java 2 Volumes 1 and 2*, 4th ed., Prentice-Hall, Englewood Cliffs, N.J., 2000.

2. J. Lewis and W. Loftus, *Java Software Solutions: Foundations of Program Design*, Addison-Wesley, Reading, Mass., 1997.

3. J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, 2nd ed, Addison-Wesley, Reading, Mass., 2000.

4. W. Savitch, *An Introduction to Computer Science & Programming*, 2nd ed, Prentice-Hall, Englewood Cliffs, N.J., 2001.

5. M. A. Weiss, *Efficient C Programming: A Practical Approach*, Prentice-Hall, Englewood Cliffs, N.J., 1995.

CHAPTER

2 Reference Types

CHAPTER 1 examined the Java primitive types. All types that are not one of the eight primitive types are *reference types*, including important entities such as strings, arrays, and file streams.

In this chapter, we will see:

- What a reference type and value is
- How reference types differ from primitive types
- Examples of reference types, including strings, arrays, and streams
- How exceptions are used to signal erroneous behavior

2.1 What Is a Reference?

Chapter 1 described the eight primitive types, along with some of the operations that these types can perform. All other types in Java are reference types, including strings, arrays, and file streams. So what is a reference? A *reference variable* (often abbreviated as simply *reference*) in Java is a variable that somehow stores the memory address where an object resides.

As an example, in Figure 2.1 are two objects of type `Point`. It happens, by chance, that these objects are stored in memory locations 1000 and 1024, respec-

tively. For these two objects, there are three references: `point1`, `point2`, and `point3`. `point1` and `point3` both reference the object stored at memory location 1000; `point2` references the object stored at memory location 1024. Both `point1` and `point3` store the value 1000, while `point2` stores the value 1024. Note that the actual locations, such as 1000 and 1024, are assigned by the compiler at its discretion (when it finds available memory). Thus these values are not useful externally as numbers. However, the fact that `point1` and `point3` store identical values is useful: It means they are referencing the same object.

A reference will always store the memory address where some object is residing, unless it is not currently referencing any object. In this case, it will store the *null reference*, `null`. Java does not allow references to primitive variables.

There are two broad categories of operations that can be applied to reference variables. One allows us to examine or manipulate the reference value. For instance, if we change the stored value of `point1` (which is 1000), we could have it reference another object. We can also compare `point1` and `point3` and determine if they are referencing the same object. The other category of operations applies to the object being referenced; perhaps we could examine or change the internal state of one of the `Point` objects. For instance, we could examine some of `Point`'s *x* and *y* coordinates.

Before we describe what can be done with references, let us see what is not allowed. Consider the expression `point1*point2`. Since the stored values of `point1` and `point2` are 1000 and 1024, respectively, their product would be 1024000. However, this is a meaningless calculation that could not have any pos-

sible use. Reference variables store addresses, and there is no logical meaning that can be associated with multiplying two addresses.

Similarly, `point1++` has no Java meaning; it suggests that `point1` — 1000 — should be increased to 1001, but in that case it might not be referencing a valid `Point` object. Many languages define the *pointer*, which behaves like a reference variable. However, pointers in C++ are much more dangerous because arithmetic on stored addresses is allowed. Thus, in C++, `point1++` has a meaning. Because C++ allows pointers to primitive types, one must be careful to distinguish between arithmetic on addresses and arithmetic on the objects being referenced. This is done by explicitly *dereferencing* the pointer. In practice, C++'s unsafe pointers tend to cause numerous programming errors.

Some operations are performed on references themselves, while other operations are performed on the objects being referenced. In Java, the only operators that are allowed for reference types (with one exception made for `String`s) are assignment via `=` and equality comparison via `==` or `!=`.

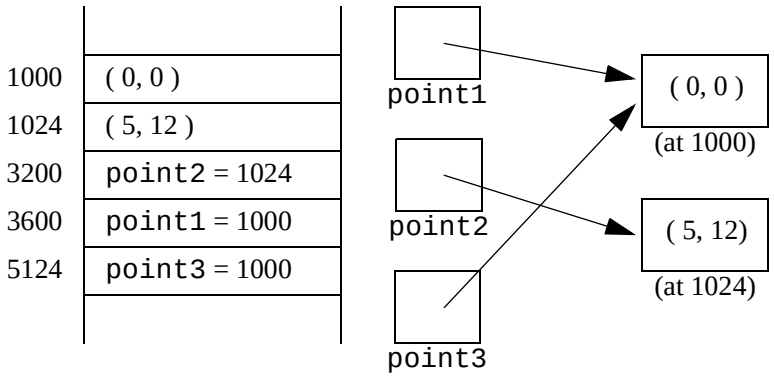

Figure 2.1 An illustration of a reference: The `Point` object stored at memory location 1000 is referenced by both `point1` and `point3`. The `Point` object stored at memory location 1024 is referenced by `point2`. The memory locations where the variables are stored are arbitrary

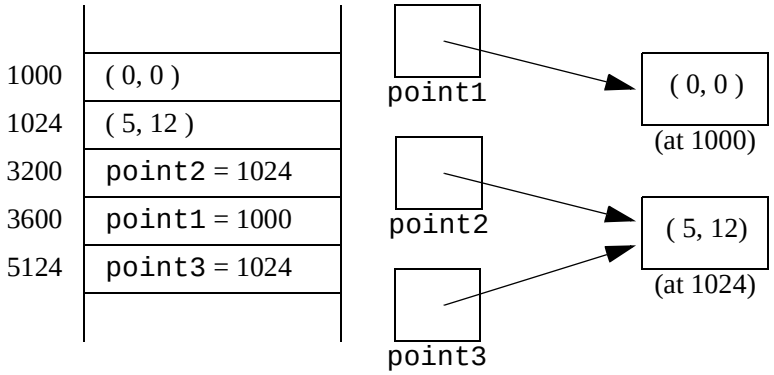

Figure 2.2 The result of `point3=point2`: `point3` now references the same object as `point2`

Figure 2.2 illustrates the assignment operator for reference variables. By assigning `point3` the stored value of `point2`, we have `point3` reference the same object that `point2` was referencing. Now, `point2==point3` is `true` because `point2` and `point3` both store 1024 and thus reference the same object. `point1!=point2` is also `true` because `point1` and `point2` reference different objects.

The other category of operations deals with the object that is being referenced. There are only three basic actions that can be done:

1. Apply a type conversion (Section 1.4.4).
2. Access an internal field or call a method via the dot operator (`.`) (Section 2.2.1).
3. Use the `instanceof` operator to verify that the stored object is of a certain type (Section 3.5.3).

The next section illustrates in more detail the common reference operations.

2.2 Basics of Objects and References

In Java, an *object* is an instance of any of the nonprimitive types.

In Java, an *object* is an instance of any of the nonprimitive types. Objects are treated differently from primitive types. Primitive types, as already shown, are handled by *value*, meaning that the values assumed by the primitive variables are stored in those variables and copied from primitive variable to primitive variable during assignments. As shown in Section 2.1, reference variables store references to objects. The actual object is stored somewhere in memory, and the reference variable stores the object's memory address. Thus a reference variable simply represents a name for that part of memory. This means that primitive variables

and reference variables behave differently. This section examines these differences in more detail and illustrates the operations that are allowed for reference variables.

2.2.1 The Dot Operator (.)

The dot operator (`.`) is used to select a method that is applied to an object. For instance, suppose we have an object of type `Circle` that defines an `area` method. If `theCircle` references a `Circle`, then we can compute the area of the referenced `Circle` (and save it to a variable of type `double`) by doing this:

```
double theArea = theCircle.area( );
```

It is possible that `theCircle` stores the `null` reference. In this case, applying the dot operator will generate a `NullPointerException` when the program runs. Generally, this will cause abnormal termination.

The dot operator can also be used to access individual components of an object, provided arrangements have been made to allow internal components to be viewable. Chapter 3 discusses how these arrangements are made. It also explains why it is generally preferable to not allow direct access of individual components.

2.2.2 Declaration of Objects

We have already seen the syntax for declaring primitive variables. For objects, there is an important difference. When we declare a reference variable, we are

simply providing a name that can be used to reference an object that is stored in memory. However, the declaration by itself does not provide that object. For example, suppose there is an object of type `Button` that we want to add into an existing `Panel p`, using the method `add` (all this is provided in the Java library). Consider the statements

```
Button b;            // b may reference a Button object
b.setLabel( "No" );  // Label the button b refers to "No"
p.add( b );          // and add to Panel p
```

When a reference type is declared, no object is allocated. At that point, the reference is to `null`. To create the object, use *new*.

All seems well with these statements until we remember that `b` is the name of some `Button` object but no `Button` has been created yet. As a result, after the declaration of `b` the value stored by the reference variable `b` is `null`, meaning `b` is not yet referring to a valid `Button` object. Consequently, the second line is illegal because we are attempting to alter an object that does not exist. In this scenario, the compiler will probably detect the error, stating that "b is uninitialized." In other cases, the compiler will not notice, and a run-time error will result in the cryptic `NullPointerException` error message.

The `new` keyword is used to *construct* an object.

The (only common) way to allocate an object is to use the `new` keyword. `new` is used to construct an object. One way to do this is as follows:

```
Button b;            // b may reference a Button object
b = new Button( );   // Now b refers to an allocated object
b.setLabel( "No" );  // Label the Button b refers to "No"
p.add( b );          // and add it to Panel p
```

Parentheses are required when `new` is used.

Note, parentheses are required after the object name.

It is also possible to combine the declaration and object construction, as in

```
Button b = new Button( );
b.setLabel( "No" );  // Label the Button b refers to "No"
p.add( b );          // and add it to Panel p
```

The construction can specify an initial state of the object.

Many objects can also be constructed with initial values. For instance, it happens that the `Button` can be constructed with a `String` that specifies the label:

```
Button b = new Button( "No" );
p.add( b );          // add it to Panel p
```

2.2.3 Garbage Collection

Java uses *garbage collection*. With garbage collection, unreferenced memory is automatically reclaimed.

Since all objects must be constructed, we might expect that when they are no longer needed, we must explicitly destroy them. In Java, when a constructed object is no longer referenced by any object variable, the memory it consumes will automatically be reclaimed and therefore be made available. This technique is known as *garbage collection*.

The run-time system (i.e. the Java Virtual Machine) guarantees that as long as it is possible to access an object by a reference, or a chain of references, the object will never be reclaimed. Once the object is unreachable by a chain of references, it can be reclaimed at the discretion of the runtime system if memory is low. It is possible that if memory does not run low, the virtual machine will not attempt to reclaim these objects.

2.2.4 The Meaning of =

lhs and rhs stand for *left-hand side* and *right-hand side*, respectively.

Suppose we have two primitive variables `lhs` and `rhs` where *`lhs`* and *`rhs`* stand for *left-hand side* and *right-hand side*, respectively. Then the assignment statement

```
lhs = rhs;
```

has a simple meaning: The value stored in `rhs` is copied to the primitive variable `lhs`. Subsequent changes to either `lhs` or `rhs` do not affect the other.

For objects, = is a reference assignment, rather than an object copy.

For objects, the meaning of = is the same: stored values are copied. If `lhs` and `rhs` are references (of compatible types), then after the assignment statement, `lhs` will refer to the same object that `rhs` does. Here, what is being copied is an address. The object that `lhs` used to refer to is no longer referred to by `lhs`. If `lhs` was the only reference to that object, then that object is now unreferenced and subject to garbage collection. Note that the objects are not copied.

Here are some examples. First, suppose we want two `Button` objects. Then suppose we try to obtain them first by creating `noButton`. Then we attempt to create `yesButton` by modifying `noButton` as follows:

```
Button noButton = new Button( "No" );
Button yesButton = noButton;
yesButton.setLabel( "Yes" );
p.add( noButton );
p.add( yesButton );
```

This does not work because only one `Button` object has been constructed. Thus the second statement simply states that `yesButton` is now another name for the constructed `Button` at line one. That constructed `Button` is now known by two

names. On line three, the constructed `Button` has its label changed to `Yes`, but this means that the single `Button` object, known by two names, is now labelled `Yes`. The last two lines add that `Button` object to the `Panel p` twice.

The fact that `yesButton` never referred to its own object is immaterial in this example. The problem is the assignment. Consider

```
Button noButton = new Button( "No" );
Button yesButton = new Button( );
yesButton = noButton;
yesButton.setLabel( "Yes" );
p.add( noButton );
p.add( yesButton );
```

The consequences are the same. Here, there are two `Button` objects that have been constructed. At the end of the sequence, the first object is being referenced by both `noButton` and `yesButton`, while the second object is unreferenced.

At first glance, the fact that objects cannot be copied seems like a severe limitation. Actually, it is not, although this does take a little getting used to. (Some objects do need to be copied. For those, if a `clone` method is available, it should be used. However, `clone` is not used in this text.)

2.2.5 Parameter Passing

Call-by-value means that for reference types, the formal parameter references the same object as does the actual argument.

Because of call-by-value, the actual arguments are sent into the formal parameters using normal assignment. If the parameter is a reference type, then we know that normal assignment means that the formal parameter now references the same object as does the actual argument. Any method applied to the formal parameter is thus also being applied to the actual argument. In other languages, this is known as *call-by-reference parameter passing*. Using this terminology for Java would be somewhat misleading because it implies that the parameter passing is different. In reality, the parameter passing has not changed; rather, it is the parameters that have changed, from nonreference types to reference types.

As an example, suppose we pass `yesButton` as a parameter to the `clearButton` routine that is defined as follows:

```
public static void clearButton( Button b )
{
    b.setLabel( "No" );
    b = null;
}
```

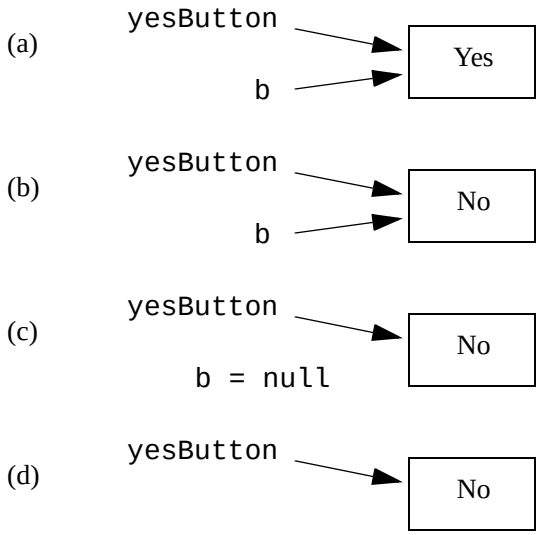

Figure 2.3 The result of call-by-value. (a) `b` is a copy of `yesButton`; (b) after `b.setLabel( "No")`: changes to the state of the object referenced by `b` are reflected in object referenced by `yesButton` because these are same object; (c) after `b=null`: change to value of `b` does not affect value of `yesButton`; (d) after method returns, `b` is destroyed.

Then, as Figure 2.3 shows, `b` references the same object as `yesButton`, and changes made to the state of this object by methods invoked through `b` will be seen when `clearButton` returns. Changes to the value of `b` (i.e. which object it references) will not have any affect on `yesButton`.

2.2.6 The Meaning of ==

For primitive types, == is true if the stored values are identical. For reference types, its meaning is different but is perfectly consistent with the previous discussion.

Two reference types are equal via == if they refer to the same stored object (or they are both `null`). Consider, for example, the following:

For reference types, == is true only if the two references reference the *same object*.

```
Button a = new Button( "Yes" );
Button b = new Button( "Yes" );
Button c = b;
```

Here, we have two objects. The first is known by the name `a`, and the second is known by two names: `b` and `c`. `b==c` is `true`. However, even though `a` and `b` are referencing objects that seem to have the same value, `a==b` is `false`, since they reference different objects. Similar rules apply for `!=`.

Sometimes it is important to know if the states of the objects being referenced are identical. All objects can be compared by using `equals`, but for many objects (including `Button`) `equals` returns `false` unless the two references are referencing the same object (in other words, for some objects `equals` is no more than the == test). We will see an example of where `equals` is useful when the `String` object is discussed in Section 2.3.

The `equals` method can be used to test whether two references reference objects that have identical states.

2.2.7 No Operator Overloading for Objects

Except for the single exception described in the next section, new operators, such as `+`, `-`, `*`, and `/` cannot be defined to work for objects. Thus there is no `<` operator available for any object. Instead, a named method, such as `lessThan`, must be defined for this task.

2.3 Strings

The `String` behaves like a reference type.

Strings in Java are handled with the `String` reference type. The language does make it appear that the `String` type is a primitive type because it provides the `+` and `+=` operator for concatenation. However, this is the only reference type for which any operator overloading is allowed. Otherwise, the `String` behaves like any other object.

2.3.1 Basics of String Manipulation

Strings are *immutable*; that is, the `String` object will not be changed.

There are two fundamental rules about a `String` object. First, with the exception of the concatenation operators, it behaves like an object. Second, the `String` is *immutable*. This means that once a `String` object is constructed, its contents may not be changed.

Because a `String` is immutable, it is always safe to use the `=` operator with it. Thus a `String` may be declared as follows:

```
String empty   = "";
String message = "Hello"
String repeat  = message;
```

After these declarations, there are two `String` objects. The first is the empty string, which is referenced by `empty`. The second is the `String` `"Hello"` which is referenced by both `message` and `repeat`. For most objects being referenced by both `message` and `repeat` could be a problem. However, because `String`s are immutable, the sharing of `String` objects is safe, as well as efficient. The only way to change the value that the string `repeat` refers to is to

construct a new `String` and have `repeat` reference it. This has no effect on the `String` that `message` references.

2.3.2 String Concatenation

Java does not allow operator overloading for reference types. However, a special language exemption is granted for string concatenation.

String concatenation is performed with + (and +=).

The operator `+`, when at least one operand is a `String`, performs concatenation. The result is a reference to a newly constructed `String` object. For example,

```
"this" + " that"   // Generates "this that"
"abc" + 5          // Generates "abc5"
5 + "abc"          // Generates "5abc"
"a" + "b" + "c"    // Generates "abc"
```

Single-character strings should not be replaced with character constants; Exercise 2.6 asks you to show why. Note that operator `+` is left-associative, and thus

```
"a" + 1 + 2        // Generates "a12"
1 + 2 + "a"        // Generates "3a"
1 + ( 2 + "a" )    // Generates "12a"
```

Also, operator `+=` is provided for the `String`. The effect of `str+=exp` is the same as `str=str+exp`. Specifically, this means that `str` will reference the newly constructed `String` generated by `str+exp`.

2.3.3 Comparing Strings

Use *equals* and *compareTo* to perform string comparison.

Since the basic assignment operator works for `String`s, it is tempting to assume that the relational and equality operators also work. This is not true.

In accordance with the ban on operator overloading, relational operators (`<`, `>`, `<=`, and `>=`) are not defined for the `String` object. Further, `==` and `!=` have the typical meaning for reference variables. For two `String` objects `lhs` and `rhs`, for example `lhs==rhs` is `true` only if `lhs` and `rhs` refer to the same `String` object. Thus, if they refer to different objects that have identical contents, `lhs==rhs` is `false`. Similar logic applies for `!=`.

To compare two `String` objects for equality, we use `equals`. `lhs.equals(rhs)` is `true` if `lhs` and `rhs` reference `String`s that store identical values.

A more general test can be performed with the method `compareTo`. `lhs.compareTo(rhs)` compares two `String` objects, `lhs` and `rhs`. It returns a negative number, zero, or a positive number, depending on whether `lhs` is lexicographically less than, equal to, or greater than `rhs`, respectively.

2.3.4 Other `String` Methods

Use *`length`*, *`charAt`*, and *`substring`* to compute string length, get a single character, and get a substring, respectively.

The length of a `String` object (an empty string has length zero) can be obtained with the method `length`. Since `length` is a method, parentheses are required.

Two methods are defined to access individual characters in a `String`. The method `charAt` gets a single character by specifying a position (the first position is position 0). The method `substring` returns a reference to a newly constructed `String`. The call is made by specifying the starting point and the first nonincluded position.

Here is an example of these three methods:

```
String greeting = "hello";
int len    = greeting.length( );          // len is 5
char ch    = greeting.charAt( 1 );        // ch  is 'e'
String sub = greeting.substring( 2, 4 );  // sub is "ll"
```

2.3.5 Converting Other Types of Strings

`toString` converts primitive types (and objects) to `String`s.

String concatenation provides a lazy way to convert any primitive to a `String`. For instance, `""+45.3` returns the newly constructed `String` `"45.3"`. There are also methods to do this directly.

The method `toString` can be used to convert any primitive type to a `String`. As an example, `Integer.toString(45)` returns a reference to the newly constructed `String` `"45"`. All reference types also provide an implementation of `toString` of varying quality. In fact, when operator `+` has only one `String` argument, the non`String` argument is converted to a `String` by auto-

matically applying an appropriate `toString`. For the integer types, an alternative form of `Integer.toString` allows the specification of a radix. Thus

```
System.out.println( "55 in base 2: " +
                    Integer.toString( 55, 2 ) );
```

prints out the binary representation of 55.

The `int` value that is represented by a `String` can be obtained by calling the method `Integer.parseInt`. This method generates an exception if the `String` does not represent an `int`. Exceptions are discussed in Section 2.5. Similar ideas work for a `double`s. Here are some examples:

```
int    x = Integer.parseInt( "75" );
double y = Double.parseDouble( "3.14" );
```

2.4 Arrays

An *array* stores a collection of identically typed entities.

An *aggregate* is a collection of entities stored in one unit. An *array* is the basic mechanism for storing a collection of identically typed entities. In Java the array is not a primitive type. Instead, it behaves very much like an object. Thus many of the rules for objects also apply to arrays.

The *array indexing operator* `[]` provides access to any object in the array.

Each entity in the array can be accessed via the *array indexing operator* `[]`. We say that the `[]` operator *indexes* the array, meaning that it specifies which object is to be accessed. Unlike C and C++, bounds checking is performed automatically.

In Java, arrays are always indexed starting at zero. Thus an array `a` of three items stores `a[0]`, `a[1]`, and `a[2]`. The number of items that can be stored in an array `a` can always be obtained by `a.length`. Note that there are no parentheses. A typical array loop would use

Arrays are indexed starting at zero. The number of items stored in the array is obtained by the *length* field. No parentheses are used.

```
for( int i = 0; i < a.length; i++ )
```

2.4.1 Declaration, Assignment, and Methods

An array is an object, so when the array declaration

```
int [ ] array1;
```

is given, no memory is yet allocated to store the array. `array1` is simply a name (reference) for an array, and at this point is `null`. To have 100 `int`s, for example, we issue a `new` command:

To allocate an array, use new.

```
array1 = new int [ 100 ];
```

Now `array1` references an array of 100 `int`s.

There are other ways to declare arrays. For instance, in some contexts

```
int [ ] array2 = new int [ 100 ];
```

is acceptable. Also, initializer lists can be used, as in C or C++, to specify initial values. In the next example, an array of four `int`s is allocated and then referenced by `array3`.

```
int [ ] array3 = { 3, 4, 10, 6 };
```

The brackets can go either before or after the array name. Placing them before makes it easier to see that the name is an array type, so that is the style used here. Declaring an array of objects (rather than primitive types) uses the same syntax. Note, however, that when we allocate an array of objects, each object initially stores a `null` reference. Each also must be set to reference a constructed object. For instance, an array of five buttons is constructed as

```
Button [ ] arrayOfButtons;
arrayOfButtons = new Button [ 5 ];
for( int i = 0; i < arrayOfButtons.length; i++ )
    arrayOfButtons[ i ] = new Button( );
```

Figure 2.4 illustrates the use of arrays in Java. The program in Figure 2.4 repeatedly chooses numbers between 1 and 100, inclusive. The output is the number of times that each number has occurred. The import directive at line 1 will be discussed in Section 3.6.1.

Always be sure to declare the correct array size. Off-by-one errors are common.

Line 14 declares an array of integers that count the occurrences of each number. Because arrays are indexed starting at zero, the `+1` is crucial if we want to access the item in position `DIFFERENT_NUMBERS`. Without it we would have an array whose indexible range was 0 to 99, and thus any access to index 100 be out-of-bounds. The loop at lines 15 and 16 initializes the array entries to zero; this is actually unnecessary, since by default, array elements are initialized to zero for primitive and null for references.

The rest of the program is relatively straightforward. It uses the `Random` object defined in the `java.util` library (hence the import directive at line 1). The `nextInt` method repeatedly gives a (somewhat) random number in the

range the includes zero but stops at one less than the parameter to `nextInt`; thus by adding one, we get a number in the desired range. The results are output at lines 25 and 26.

```
 1 import java.util.Random;
 2
 3 public class RandomNumbers
 4 {
 5     // Generate random numbers (from 1-100)
 6     // Print number of occurrences of each number
 7
 8     public static final int DIFF_NUMBERS     =     100;
 9     public static final int TOTAL_NUMBERS    = 1000000;
10
11     public static void main( String [ ] args )
12     {
13         // Create array; initialize to 0s
14         int [ ] numbers = new int [ DIFF_NUMBERS + 1 ];
15         for( int i = 0; i < numbers.length; i++ )
16             numbers[ i ] = 0;
17
18         Random r = new Random( );
19
20         // Generate the numbers
21         for( int i = 0; i < TOTAL_NUMBERS; i++ )
22             numbers[ r.nextInt( DIFF_NUMBERS ) + 1 ]++;
23
24         // Output the summary
25         for( int i = 1; i <= DIFF_NUMBERS; i++ )
26             System.out.println( i + ": " + numbers[ i ] );
27     }
28 }
```

Figure 2.4 Simple demonstration of arrays

Since an array is a reference type, = does not copy arrays. Instead, if `lhs` and `rhs` are arrays the effect of

```
int [ ] lhs = new int [ 100 ];
int [ ] rhs = new int [ 100 ];
   ...
lhs = rhs
```

is that the array object that was referenced by `rhs` is now also referenced by `lhs`. Thus changing `rhs[0]` also changes `lhs[0]`. (To make `lhs` an independent copy of `rhs`, one could use the `clone` method, but often making complete copies is not really needed.)

Finally, an array can be used as a parameter to a method. The rules follow logically from our understanding that an array name is a reference. Suppose we have a method `methodCall` that accepts one array of `int` as its parameter. The caller/callee views are

```
methodCall( actualArray );        // method call
void methodCall( int [ ] formalArray ) // method declaration
```

The contents of an array are passed by reference.

In accordance with the parameter-passing conventions for Java reference types, `formalArray` references the same array object as `actualArray`. Thus `formalArray[i]` accesses `actualArray[i]`. This means that if the method modifies any element in the array, the modifications will be observable after the method execution has completed. Also note that a statement such as

```
formalArray = new int [ 20 ];
```

has no effect on `actualArray`. Finally, since array names are simply references, they can be returned.

2.4.2 Dynamic Array Expansion

Dynamic array expansion allows us to allocate arbitrary-sized arrays and make them larger if needed.

Suppose we want to read a sequence of numbers and store them in an array for processing. The fundamental property of an array requires us to declare a size so that the compiler can allocate the correct amount of memory. Also, we must make this declaration prior to the first access of the array. If we have no idea how many items to expect, then it is difficult to make a reasonable choice for the array size. This section shows how to expand arrays if the initial size is too small. This technique is called *dynamic array expansion* and allows us to allocate arbitrary-sized arrays and make them larger or smaller as the program runs.

The allocation method for arrays that we have seen thus far is

```
int [ ] arr = new int[ 10 ];
```

Suppose that we decide, after the declarations, that we really need 12 `int`s instead of 10. In this case, we can use the following maneuver, which is illustrated in Figure 2.5:

```
int [ ] original = arr;   // 1. Save reference to arr
arr = new int [ 12 ];     // 2. Have a reference more memory
for( int i = 0; i < 10; i++ )  // 3. Copy the old data over
    arr[ i ] = original[ i ];
original = null;          // 4. Unreference original array
```

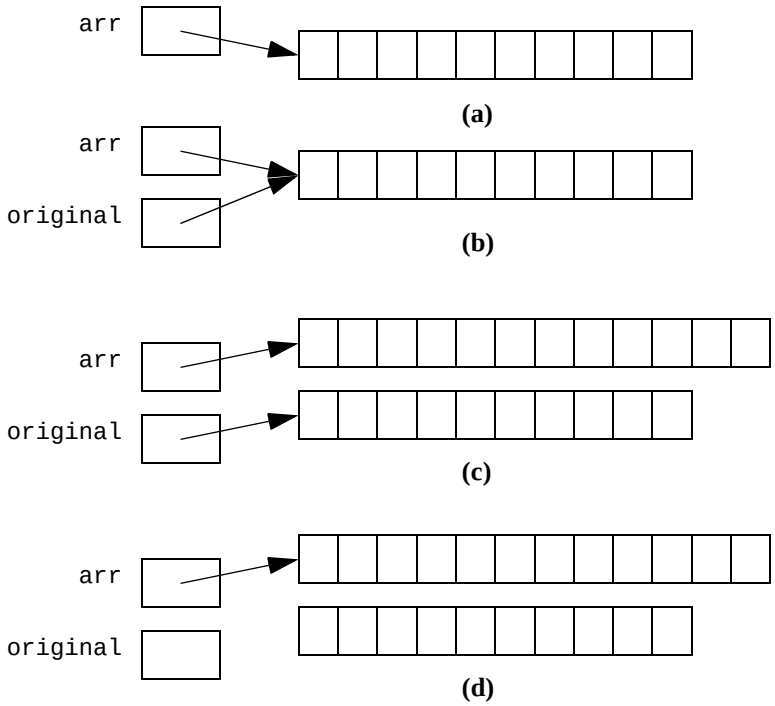

Figure 2.5 Array expansion, internally: (a) At the starting point, `arr` represents 10 integers; (b) after step 1, `original` represents the same 10 integers; (c) after steps 2 and 3, `arr` represents 12 integers, the first 10 of which are copied from `original`; and (d) after step 4, the 10 integers are available for reclamation.

Always expand the array to a size that is some multiplicative constant times as large. Doubling is a good choice.

A moment's thought will convince you that this is an expensive operation. This is because we copy all of the elements from `original` back to `arr`. If, for instance, this array expansion is in response to reading input, it would be inefficient to reexpand every time we read a few elements. Thus when array expansion is implemented, we always make it some *multiplicative* constant times as large. For instance, we might expand it to be twice as large. In this way, when we expand

the array from *N* items to 2*N* items, the cost of the *N* copies can be apportioned over the next *N* items that can be inserted into the array without an expansion.

```
 1 import java.io.InputStreamReader;
 2 import java.io.BufferedReader;
 3 import java.io.IOException;
 4
 5 public class ReadStrings
 6 {
 7     // Read an unlimited number of String; return a String [ ]
 8     // The minimal I/O details used here are not important for
 9     // this example and are discussed in Section 2.6.
10     public static String [ ] getStrings( )
11     {
12         BufferedReader in = new BufferedReader( new
13                                 InputStreamReader( System.in ) );
14         String [ ] array = new String[ 5 ];
15         String oneLine;
16         int itemsRead = 0;
17
18         System.out.println( "Enter strings, one per line; " );
19         System.out.println( "Terminate with empty line: " );
20
21         try
22         {
23             while( ( oneLine = in.readLine( ) ) != null &&
24                     !oneLine.equals( "" ) )
25             {
26                 if( itemsRead == array.length )
27                     array = resize( array, array.length * 2 );
28                 array[ itemsRead++ ] = oneLine;
29             }
30         }
31         catch( IOException e )
32         {
33             System.out.println( "Early abort of read." );
34         }
35
36         return resize( array, itemsRead );
37     }
```

Figure 2.6 Code to read an unlimited number of `String`s and output them (part 1)

To make things more concrete, Figures 2.6 and 2.7 show a program that reads an unlimited number of strings from the standard input and stores the result in a

dynamically expanding array. An empty line is used to signal the end of input. (The minimal I/O details used here are not important for this example and are discussed in Section 2.6.) The `resize` routine performs the array expansion (or shrinking), returning a reference to the new array. Similarly, the method `getStrings` returns (a reference to) the array where it will reside.

At the start of `getStrings`, `itemsRead` is set to 0 and we start with an initial five-element array. We repeatedly read new items at line 23. If the array is full, as indicated by a successful test at line 26, then the array is expanded by calling `resize`. Lines 42 to 48 perform the array expansion using the exact strategy outlined previously. At line 28, the actual input item is assigned to the array and the number of items read is incremented. If an error occurs on input, we simply stop processing. Finally, at line 36 we shrink the array to match the number of items read prior to returning.

```
38      // Resize a String[ ] array; return new array
39      public static String [ ] resize( String [ ] array,
40                                       int newSize )
41      {
42          String [ ] original = array;
43          int numToCopy = Math.min( original.length, newSize );
44
45          array = new String[ newSize ];
46          for( int i = 0; i < numToCopy; i++ )
47              array[ i ] = original[ i ];
48          return array;
49      }
50
51      public static void main( String [ ] args )
52      {
53          String [ ] array = getStrings( );
54          for( int i = 0; i < array.length; i++ )
55              System.out.println( array[ i ] );
56      }
57  }
```

Figure 2.7 Code to read an unlimited number of `String`s and output them (part 2)

2.4.3 `ArrayList`

The `ArrayList` is used for expanding arrays.

The technique used in Section 2.4.2 is so common that the Java Library contains an `ArrayList` type with built-in functionality to mimic it. The basic idea is that an `ArrayList` maintains not only a size, but also a capacity; the capacity is the amount of memory that it has reserved. The capacity of the `ArrayList` is really an internal detail, not something that you need worry about.

The `add` function increases the size by 1, adds a new item to the array at the appropriate position, expanding capacity if needed.

The `add` function increases the size by one, and adds a new item into the array at the appropriate position. This is a trivial operation if capacity has not been reached. If it has, the capacity is automatically expanded, using the strategy described in Section 2.4.2. The `ArrayList` is initialized with a size of `0`.

```
 1 import java.io.InputStreamReader;
 2 import java.io.BufferedReader;
 3 import java.io.IOException;
 4
 5 import java.util.ArrayList;
 6
 7 public class ReadStringsWithArrayList
 8 {
 9     public static void main( String [ ] args )
10     {
11         ArrayList array = getStrings( );
12         for( int i = 0; i < array.size( ); i++ )
13             System.out.println( array.get( i ) );
14     }
15
16     // Read an unlimited number of String; return an ArrayList
17     // The minimal I/O details used here are not important for
18     // this example and are discussed in Section 2.6.
19     public static ArrayList getStrings( )
20     {
21         BufferedReader in = new BufferedReader( new
22                                 InputStreamReader( System.in ) );
23         ArrayList array = new ArrayList( );
24         String oneLine;
25
26         System.out.println( "Enter strings, one per line; " );
27         System.out.println( "Terminate with empty line: " );
28
29         try
30         {
31             while( ( oneLine = in.readLine( ) ) != null &&
32                     !oneLine.equals( "" ) )
33                 array.add( oneLine );
34         }
35         catch( IOException e )
36         {
37             System.out.println( "Early abort of read." );
38         }
39
40         return array;
41     }
42 }
```

Figure 2.8 Code to read an unlimited number of `String`s and output them, using an `ArrayList`

Because indexing via `[]` is reserved only for primitive arrays, much as was the case for `String`s, we have to use a method to access the `ArrayList` items.

The `get` method returns the object at a specified index, and the `set` method can be used to change the value of a reference at a specified index; `get` thus behaves like the `charAt` method. We will be describing the implementation details of `ArrayList` at several points in the text, and eventually write our own version.

The code in Figure 2.8 shows how `add` is used in `getStrings`; it is clearly much simpler than the `getStrings` function in Section 2.4.2. It is important to mention, however, that only objects (which are accessed by reference variables) can be added into an `ArrayList`. The eight primitive types cannot. However, there is an easy workaround for that, which we will discuss in Section 4.6.2.

2.4.4 Multidimensional Arrays

A *multidimensional array* is an array that is accessed by more than one index.

Sometimes arrays need to be accessed based on more than one index. A common example of this is a matrix. A *multidimensional array* is an array that is accessed by more than one index. It is allocated by specifying the size of its indices, and each element is accessed by placing each index in its own pair of brackets. As an example, the declaration

```
int [ ][ ] x = new int[ 2 ][ 3 ];
```

defines the two-dimensional array `x`, with the first index (representing the number of rows) ranging from 0 to 1 and the second index (the number of columns) ranging from 0 to 2 (for a total of six objects). The compiler sets aside six memory locations for these objects.

In the example above, the two dimensional array is actually an array of arrays. As such, the number of rows is `x.length`, which is 2. The number of columns is `x[0].length` or `x[1].length`, both of which are 3.

Figure 2.9 illustrates how to print the contents of a two-dimensional array. The code works not only for rectangular two-dimensional arrays, but also for *ragged two-dimensional arrays*, in which the number of columns varies from row to row. This is easily handled by using `m[i].length` at line 11 to represent the number of columns in row `i`. We also handle the possibility that rows might be `null` (which is different than length 0), with the test at line 7. The `main` routine illustrates the declaration of two-dimensional arrays for the case where initial values are known. It is simply an extension of the one-dimensional case discussed in Section 2.4.1. Array `a` is a straightforward rectangular matrix, array `b` has a `null` row, and array `c` is ragged.

```
 1 class MatrixDemo
 2 {
 3     public static void printMatrix( int [][] m )
 4     {
 5         for( int i = 0; i < m.length; i++ )
 6         {
 7             if( m[ i ] == null )
 8                 System.out.println( "(null)" );
 9             else
10             {
11                 for( int j = 0; j < m[i].length; j++ )
12                     System.out.print( m[ i ][ j ] + " " );
13                 System.out.println( );
14             }
15         }
16     }
17
18     public static void main( String [] args )
19     {
20         int [][] a = { { 1, 2 }, { 3, 4 }, { 5, 6 } };
21         int [][] b = { { 1, 2 }, null, { 5, 6 } };
22         int [][] c = { { 1, 2 }, { 3, 4, 5 }, { 6 } };
23
24         System.out.println( "a: " ); printMatrix( a );
25         System.out.println( "b: " ); printMatrix( b );
26         System.out.println( "c: " ); printMatrix( c );
27     }
28 }
```

Figure 2.9 Printing a two-dimensional array

2.4.5 Command-line Arguments

Command-line arguments are available by examining the parameter to `main`.

Command-line arguments are available by examining the parameter to `main`. The array of strings represents the additional command-line arguments. For instance, when the program is invoked,

```
java Echo this that
```

`args[0]` references the `String` `"this"` and `args[1]` references the `String` `"that"`. Thus the program in Figure 2.10 implements the `echo` command.

```
 1 public class Echo
 2 {
 3     // List the command-line arguments
 4     public static void main( String [ ] args )
 5     {
 6         for( int i = 0; i < args.length - 1; i++ )
 7             System.out.print( args[ i ] + " " );
 8         if( args.length != 0 )
 9             System.out.println( args[ args.length - 1 ] );
10         else
11             System.out.println( "No arguments to echo" );
12     }
13 }
```

Figure 2.10 The `echo` command

2.5 Exception Handling

Exceptions are used to handle exceptional occurrences such as errors.

Exceptions are objects that store information and are transmitted outside the normal return sequence. They are propagated back through the calling sequence until some routine *catches* the exception. At that point, the information stored in the object can be extracted to provide error handling. Such information will always include details about where the exception was created. The other important piece of information is the type of the exception object. For instance, when an `ArrayIndexOutBoundsException` is propagated, it is clear that the basic problem is a bad index. Exceptions are used to signal *exceptional occurrences* such as errors.

2.5.1 Processing Exceptions

A *try block* encloses code that might generate an exception.

The code in Figure 2.11 illustrates the use of exceptions. Code that might result in an exception's being propagated is enclosed in a `try` block. The `try` block extends from lines 17 to 21. Immediately following the `try` block are the exception handlers. This part of the code is jumped to only if an exception is raised; at the point the exception is raised, the `try` block in which it came from is considered terminated. Each `catch` block is attempted in order until a matching handler is found. An `IOException` is generated by `readLine` if some unexpected error occurs, and a `NumberFormatException` is generated by `parseInt` if `oneLine` is not convertible to an `int`.

```
 1 import java.io.BufferedReader;
 2 import java.io.InputStreamReader;
 3 import java.io.IOException;
 4
 5 public class DivideByTwo
 6 {
 7     public static void main( String [ ] args )
 8     {
 9             // BufferedReader is discussed in Section 2.6
10         BufferedReader in = new BufferedReader( new
11                                 InputStreamReader( System.in ) );
12         int x;
13         String oneLine;
14
15         System.out.println( "Enter an integer: " );
16         try
17         {
18             oneLine = in.readLine( );
19             x = Integer.parseInt( oneLine );
20             System.out.println( "Half of x is " + ( x / 2 ) );
21         }
22         catch( IOException e )
23           { System.out.println( e ); }
24         catch( NumberFormatException e )
25           { System.out.println( e ); }
26     }
27 }
```

Figure 2.11 Simple program to illustrate exceptions

A *catch block* processes an exception.

The code in the `catch` block — in this case line 23 or 25 — is executed if the appropriate exception is matched. Then the `catch` block and the the `try`/`catch` sequence is considered terminated.[1] A meaningful message is printed from the exception object `e`. Alternatively, additional processing and more detailed error messages could be given.

1. Note that both `try` and `catch` require a block and not simply a single statement. Thus braces are not optional. To save space, we often place simple catch clauses on a single line with their braces, indented two additional spaces, rather than use three lines. Later in the text we will use this style for one-line methods.

2.5.2 The `finally` Clause

The *`finally` clause* is always executed to prior to completion of a block, regardless of exceptions.

Some objects that are created in a `try` block must be cleaned up. For instance, files that are opened in the `try` block may need to be closed prior to leaving the `try` block. One problem with this is that if an exception object is thrown during execution of the `try` block, the clean up might be omitted because the exception will cause an immediate break from the `try` block. Although we can place the clean up immediately after the last `catch` clause, this works only if the exception is caught by one of the `catch` clauses. And this may be difficult to guarantee.

The `finally` clause that may follow the last `catch` block (or the `try` block if there are no `catch` blocks) is used in this situation. The `finally` clause consists of the keyword `finally` followed by the `finally` block. There are three basic scenarios.

1. If the `try` block executes without exception, control passes to the `finally` block. This is true even if the `try` block exits prior to the last statement via a `return`, `break`, or `continue`.
2. If an uncaught exception is encountered inside the `try` block, control passes to the `finally` block. Then, after executing the `finally` block, the exception propagates.
3. If a caught exception is encountered in the `try` block, control passes to the appropriate `catch` block. Then, after executing the `catch` block, the `finally` block is executed.

2.5.3 Common Exceptions

Run-time exceptions do not have to be handled.

There are several types of standard exceptions in Java. The *standard run-time exceptions* include events such as integer divide-by-zero and illegal array access. Since these events can happen virtually anywhere, it would be overly burdensome

to require exception handlers for them. If a `catch` block is provided, these exceptions behave like any other exception. If a `catch` block is not provided for a standard exception, and a standard exception is thrown, then it propagates as usual, possibly past `main`. In this case, it causes an abnormal program termination, with an error message. Some of the common standard run-time exceptions are shown in Figure 2.12. Generally speaking, these are programming errors and should not be caught. `NumberFormatException` is a notable violation of this principle, but `NullPointerException` is more typical.

Standard Run-time Exception	Meaning
`ArithmeticException`	Overflow or integer division by zero.
`NumberFormatException`	Illegal conversion of `String` to numeric type.
`IndexOutOfBoundsException`	Illegal index into an array or `String`.
`NegativeArraySizeException`	Attempt to create a negative-length array.
`NullPointerException`	Illegal attempt to use a null reference.
`SecurityException`	Run-time security violation.

Figure 2.12 Common standard run-time exceptions

Checked exceptions must be handled or listed in a *`throws` clause.*

Most exceptions are of the *standard checked exception* variety. If a method is called that might either directly or indirectly throw a standard checked exception, then the programmer must either provide a `catch` block for it, or explicitly indicate that the exception is to be propagated by use of a *`throws` clause* in the method declaration. Note that eventually it should be handled because it is terrible style for `main` to have a `throws` clause. Some of the common standard checked exceptions are shown in Figure 2.13.

Errors are virtual machine problems. The most common error is `OutOfMemoryError`. Others include `InternalError` and the infamous `UnknownError`, in which the virtual machine has decided that it is in trouble, does not know why, but does not want to continue. Generally speaking an `Error` is unrecoverable and should not be cauught.

Errors are unrecoverable exceptions.

2.5.4 The `throw` and `throws` Clauses

The programmer can generate an exception by use of the *`throw` clause*. For instance, we can create and then throw an `ArithmeticException` object by

The *`throw` clause* is used to throw an exception.

```
throw new ArithmeticException( "Divide by zero" );
```

Since the intent is to signal to the caller that there is a problem, you should never throw an exception only to catch it a few lines later in the same scope. In other words, do not place a throws clause in a try block, and then handle it immediately in the corresponding catch block. Instead, let it leave unhandled, and pass the exception up to the caller. Otherwise, you are using exceptions as a cheap goto statement, which is not good programming, and is certainly not what an exception—signalling an exceptional occurrence—is to be used for.

Standard Checked Exception	Meaning
`java.io.EOFException`	End-of-file before completion of input.
`java.io.FileNotFoundException`	File not found to open.

Figure 2.13 Common standard checked exceptions

Standard Checked Exception	Meaning
`java.io.IOException`	Includes most I/O exceptions.
`InterruptedException`	Thrown by the `Thread.sleep` method.

Figure 2.13 Common standard checked exceptions

```
 1 import java.io.IOException;
 2
 3 public class ThrowDemo
 4 {
 5     public static void processFile( String toFile )
 6                                             throws IOException
 7     {
 8         // Omitted implementation propagates all
 9         // thrown IOException back to the caller
10     }
11
12     public static void main( String [ ] args )
13     {
14         for( int i = 0; i < args.length; i++ )
15         {
16             try
17               { processFile( args[ i ] ); }
18             catch( IOException e )
19               { System.err.println( e ); }
20         }
21     }
22 }
```

Figure 2.14 Illustration of the `throws` clause

Java allows programmers to create their own exception types. Details on creating and throwing user-defined exceptions are provided in Chapter 4.

The *`throws` clause* indicates propagated exceptions.

As mentioned earlier, standard checked exceptions must either be caught or explicitly propagated to the calling routine, but they should, as a last resort, eventually be handled in `main`. To do the latter, the method that is unwilling to catch the exception must indicate, via a *`throws` clause*, which exceptions it may propagate. The `throws` clause is attached at the end of the method header. Figure

2.14 illustrates a method that propagates any `IOException`s that it encounters; these must eventually be caught in `main` (since we will not place a `throws` clause in `main`).

2.6 Input and Output

Input and output (I/O) in Java is achieved through the use of the `java.io` package. The types in the I/O package are all prefixed with `java.io`, including as we have seen, `java.io.IOException`. The import directive allows you to avoid using complete names. For instance, with

```
import java.io.IOException
```

at the top of your code, you can use `IOException` as a shorthand for `java.io.IOException`. (Many common types, such as `String` and `Math` do not require import directives, as they are automatically visible by the shorthands by virtue of being in `java.lang`.)

The Java library is very sophisticated and has a host of options. Here, we examine only the most basic uses, concentrating entirely on formatted I/O. In Section 4.5.3, we will discuss the design of the library.

2.6.1 Basic Stream Operations

Like many languages, Java uses the notion of streams for I/O. To perform I/O to the terminal, a file, or over the Internet, the programmer creates an associated *stream*. Once that is done, all I/O commands are directed to that stream. A pro-

grammer defines a stream for each I/O target (for instance, each file requiring input or output).

The predefined streams are `System.in`, `System.out`, and `System.err`.

Three streams are predefined for terminal I/O: `System.in`, the standard input; `System.out`, the standard output; and `System.err`, the standard error.

As already mentioned, the `print` and `println` methods are used for formatted output. Any type can be converted to a `String` suitable for printing by calling its `toString` method; in many cases, this is done automatically. Unlike with C and C++, which have an enormous number of formatting options, output in Java is done almost exclusively by `String` concatenation, with no built-in formatting.

`BufferedReader` is used for line-at-a-time input.

A simple method for reading formatted input is to read a single line into a `String` object using `readLine`. The `readline` method reads until it encounters a line terminator or end of file. The characters that are read, minus the line terminator (if read), are returned as a newly constructed `String`. To use `readLine`, we must first construct a `BufferedReader` object from an `InputStreamReader` object that is itself constructed from `System.in`. This was illustrated in Figure 2.11 at lines 10 and 11.

If an immediate end of file is encountered, then `null` is returned. If a read error occurs for some reason other than end of file, then some `IOException` is generated. Note that the `IOException`, which is a standard checked exception, must eventually be caught. In many instances, the `IOException` is allowed to propagate back to a `catch` block in the `main` method; this technique was illustrated in Figure 2.14.

2.6.2 The `StringTokenizer` Type

Recall that to read a single primitive type, such as an `int`, we use `readLine` to read the line as a `String` and then apply a method to generate the primitive type from the `String`. For `int`, we can use `parseInt`.

```
 1 import java.io.InputStreamReader;
 2 import java.io.BufferedReader;
 3 import java.io.IOException;
 4 import java.util.StringTokenizer;
 5
 6 public class MaxTest
 7 {
 8     public static void main( String args[ ] )
 9     {
10         BufferedReader in = new BufferedReader( new
11                                 InputStreamReader( System.in ) );
12
13         String oneLine;
14         StringTokenizer str;
15         int x;
16         int y;
17
18         System.out.println( "Enter 2 ints on one line: " );
19         try
20         {
21             oneLine = in.readLine( );
22             if( oneLine == null )
23                 return;
24
25             str = new StringTokenizer( oneLine );
26             if( str.countTokens( ) != 2 )
27             {
28                 System.out.println( "Error: need two ints" );
29                 return;
30             }
31             x = Integer.parseInt( str.nextToken( ) );
32             y = Integer.parseInt( str.nextToken( ) );
33             System.out.println( "Max: " + Math.max( x, y ) );
34         }
35         catch( IOException e )
36           { System.err.println( "Unexpected IO error" ); }
37         catch( NumberFormatException e )
38           { System.err.println( "Error: need two ints" ); }
39
40     }
41 }
```

Figure 2.15 Program that demonstrates the string tokenizer

`StringTokenizer` is used to extract delimited substrings from a large string.

Sometimes we have several items on a line. For instance, suppose each line has two `int`s. Java provides the `StringTokenizer` type to separate a `String` into tokens. To use it by its shortened name, provide the `import` directive

```
import java.util.StringTokenizer;
```

Use of the string tokenizer is illustrated in Figure 2.15. First, at line 25, we construct a `StringTokenizer` object by providing the `String` representing the line of input. The `countTokens` method, shown on line 26, will provide the number of tokens in the `String`; in this example, this should be two, or else the input is in error. Then, the `nextToken` method returns the next token as a `String`. This method throws `NoSuchElementException` if there is no token, but this is a runtime exception and does not have to be caught. At lines 31 and 32, we use `nextToken` followed by `parseInt` to obtain an `int`. All errors, including the failure to provide exactly two tokens, are handled in the `catch` blocks.

By default, tokens are separated by whitespace. The `StringTokenizer` can be constructed to recognize other characters as delimiters and to include these delimiters as tokens.

2.6.3 Sequential Files

`FileReader` is used for file input.

One of the basic rules of Java is that what works for terminal I/O also works for files. To deal with a file, we do not construct a `BufferedReader` object from

an `InputStreamReader`. Instead, we construct it from a `FileReader` object, which itself can be constructed by providing a filename.

An example that illustrates these basic ideas is shown in Figure 2.16. Here, we have a program that will list the contents of the text files that are specified as command-line arguments. The `main` routine simply steps through the command-line arguments, passing each one to `listFile`. In `listFile`, we construct the `FileReader` object at line 24, and then use it to construct a `BufferedReader` object — `fileIn` — at line 25. At that point, reading is identical to what we have already seen.

After we are done with the file, we must close it; otherwise, we could eventually run out of streams. Note that this cannot be done at the end of the `try` block, since an exception could cause a premature exit from the block. Thus we close the file in a `finally` block, which is guaranteed to be started whether there are no exceptions, handled exceptions, or unhandled exceptions. The code to handle the `close` is complex because:

1. `fileIn` must be declared outside of the try block in order to be visible in the finally block.
2. `fileIn` must be initialized to `null` to avoid compiler complaints about a possible uninitialized variable.
3. Prior to calling `close`, we must check that `fileIn` is not `null` to avoid generating a `NullPointerException` (`fileIn` would be `null` if the file was not found, resulting in an `IOException` prior to its assignment).
4. `close` might itself throw a checked exception, and requires a try/catch block.

```
 1 import java.io.FileReader;
 2 import java.io.BufferedReader;
 3 import java.io.IOException;
 4
 5 public class ListFiles
 6 {
 7     public static void main( String [ ] args )
 8     {
 9         if( args.length == 0 )
10             System.out.println( "No files specified" );
11         for( int i = 0; i < args.length; i++ )
12             listFile( args[ i ] );
13     }
14
15     public static void listFile( String fileName )
16     {
17         FileReader theFile;
18         BufferedReader fileIn = null;
19         String oneLine;
20
21         System.out.println( "FILE: " + fileName );
22         try
23         {
24             theFile = new FileReader( fileName );
25             fileIn  = new BufferedReader( theFile );
26             while( ( oneLine = fileIn.readLine( ) ) != null )
27                 System.out.println( oneLine );
28         }
29         catch( IOException e )
30           {  System.out.println( e ); }
31         finally
32         {
33             // Close the stream
34             try
35             {
36                 if(fileIn != null )
37                     fileIn.close( );
38             }
39             catch( IOException e )
40               { }
41         }
42     }
43 }
```

Figure 2.16 Program to list contents of a file

```
// Double space files specified on command line.

import java.io.FileReader;
import java.io.BufferedReader;
import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.IOException;

public class DoubleSpace
{
    public static void main( String [ ] args )
    {
        for( int i = 0; i < args.length; i++ )
            doubleSpace( args[ i ] );
    }

    public static void doubleSpace( String fileName )
    {
        PrintWriter   fileOut = null;
        BufferedReader fileIn = null;

        try
        {
            fileIn  = new BufferedReader(
                          new FileReader( fileName ) );
            fileOut = new PrintWriter(
                          new FileWriter( fileName + ".ds" ) );

            String oneLine;
            while( ( oneLine = fileIn.readLine( ) ) != null )
                fileOut.println( oneLine + "\n" );
        }
        catch( IOException e )
          { e.printStackTrace( ); }

        finally
        {
            try
            {
                if( fileOut != null )
                    fileOut.close( );
                if( fileIn != null )
                    fileIn.close( );
            }
            catch( IOException e )
              { e.printStackTrace( ); }
        }
    }
}
```

Figure 2.17 Program to double-space files

`FileWriter` is used for file output.

Formatted file output is similar to file input. `FileWriter`, `PrintWriter`, and `println` replace `FileReader`, `BufferedReader`, and `readLine`, respectively. Figure 2.17 illustrates a program that double-spaces files that are specified on the command line (the resulting files are placed in a file with a `.ds` extension).

This description of Java I/O, while enough to do basic formatted I/O, hides an interesting object-oriented design that is discussed in more detail in Section 4.5.3.

Summary

This chapter examined reference types. A reference is a variable that stores either the memory address where an object resides or the special reference `null`. Only objects may be referenced; any object can be referenced by several reference variables. When two references are compared via `==`, the result is `true` if both references refer to the same object. Similarly, `=` makes a reference variable reference another object. Only a few other operations are available. The most significant is the dot operator, which allows the selection of an object's method or access of its internal data.

Because there are only eight primitive types, virtually everything of consequence in Java is an object and is accessed by a reference. This includes `String`s, arrays, exception objects, data and file streams, and a string tokenizer.

The `String` is a special reference type because `+` and `+=` can be used for concatenation. Otherwise, a `String` is like any other reference; `equals` is required to test if the contents of two `String`s are identical. An array is a collec-

tion of identically typed values. The array is indexed starting at 0, and index range checking is guaranteed to be performed. Arrays can be expanded dynamically by using `new` to allocate a larger amount of memory and then copying over individual elements.

Exceptions are used to signal exceptional events. An exception is signaled by the `throw` clause; it is propagated until handled by a `catch` block that is associated with a `try` block. Except for the run-time exceptions and errors, each method must signal the exceptions that it might propagate by using a `throws` list.

`StringTokenizer`s are used to parse a `String` into other `String`s. Typically, it is used in conjunction with other input routines. Input is handled by `BufferedReader`, `InputStreamReader`, and `FileReader` objects.

The next chapter shows how to design new types by defining a *class*.

Objects of the Game

aggregate A collection of objects stored in one unit. (58)

array Stores a collection of identically typed objects. (58)

array indexing operator [] Provides access to any element in the array. (58)

`ArrayList` Stores a collection of objects in array-like format, with easy expansion via the `add` method. (67)

`BufferedReader` Used for line-at-a-time input. (80)

call by reference In many programming languages, means that the formal parameter is a reference to the actual argument. This is the natural effect achieved in Java when call-by-value is used on reference types. (51)

`catch` block Used to process an exception. (74)

checked exception Must be either caught or explicitly allowed to propagate by a `throws` clause. (76)

command-line argument Accessed by a parameter to `main`. (71)

construction For objects, is performed via the `new` keyword. (47)

dot member operator (`.`) Allows access to each member of an object. (46)

dynamic array expansion Allows us to make arrays larger if needed. (63)

`equals` Used to test if the values stored in two objects are the same. (53)

`Error` An unrecoverable exception. (77)

exception Used to handle exception occurrences, such as errors. (72)

`FileReader` Used for file input. (83)

`FileWriter` Used for file output. (87)

`finally` clause Always executed prior to exiting a `try`/`catch` sequence. (75)

garbage collection Automatic reclaiming of unreferenced memory. (48)

immutable Object whose state cannot change. Specifically, the `String`s are immutable. (54)

input and output (I/O) Achieved through the use of the `java.io` package. (79)

`java.io` Package that is imported for nontrivial I/O. (79)

`length` field Used to determine the size of an array. (59)

`length` method Used to determine the length of a string. (57)

`lhs` and `rhs` Stands for left-hand side and right-hand side, respectively. (49)

multidimensional array An array that is accessed by more than one index. (69)

`new` Used to construct an object. (47)

`null` reference The value of an object reference that does not refer to any object. (42)

`NullPointerException` Generated when attempting to apply a method to a `null` reference. (47)

object A nonprimitive entity. (45)

reference type Any type that is not a primitive type. (45)

run-time exception Does not have to be handled. Examples include `ArithmeticException` and `NullPointerException`. (75)

`String` A special object used to store a collection of characters. (54)

string concatenation Performed with `+` and `+=` operators. (55)

`StringTokenizer` Used to extract delimited `String`s from a single `String`. Found in the `java.util` package. (83)

`System.in`, **`System.out`**, and **`System.err`** The predefined I/O streams. (80)

`throw` clause Used to throw an exception. (77)

`throws` clause Indicates that a method might propagate an exception. (78)

`toString` method Converts a primitive type or object to a `String`. (57)

`try` block Encloses code that might generate an exception. (73)

Common Errors

1. For reference types and arrays, = does not make a copy of object values. Instead, it copies addresses.
2. For reference types and strings, `equals` should be used instead of == to test if two objects have identical states.
3. Off-by-one errors are common in all languages.
4. Reference types are initialized to `null` by default. No object is constructed without calling `new`. An "uninitialized reference variable" or `NullPointerException` indicates that you forgot to allocate the object.
5. In Java, arrays are indexed from 0 to `N-1`, where `N` is the array size. However, range checking is performed, so an out-of-bounds array access is detected at run-time.
6. Two-dimensional arrays are indexed as `A[i][j]`, not `A[i,j]`.
7. Checked exceptions must either be caught or explicitly allowed to propagate with a `throws` clause.
8. Use `" "` and not `' '` for outputting a blank.

On the Internet

Following are the available files for this chapter. Everything is self-contained, and nothing is used later in the text.

RandomNumbers.java Contains the code for the example in Figure 2.4.

ReadStrings.java Contains the code for the example in Figures 2.6 and 2.7.

ReadStringsWithArrayList.java Contains the code for the example in Figure 2.8.

MatrixDemo.java Contains the code for the example in Figure 2.9.

Echo.java Contains the code for the example in Figure 2.10.

DivideByTwo.java Contains the code for the example in Figure 2.11

MaxTest.java Contains the code for the example in Figure 2.15.

ListFiles.java Contains the code for the example in Figure 2.16.

DoubleSpace.java Contains the code for the example in Figure 2.17.

Exercises

In Short

2.1. List the major differences between reference types and primitive types.

2.2. List five operations that can be applied to a reference type.

2.3. What are the differences between an array and `ArrayList`?

2.4. Describe how exceptions work in Java.

2.5. List the basic operations that can be performed on strings.

In Theory

2.6. If `x` and `y` have the values of 5 and 7, respectively, what is output by the following:

```
System.out.println( x + ' ' + y );
System.out.println( x + " " + y );
```

In Practice

2.7. A *checksum* is the 32-bit integer that is the sum of the Unicode characters in a file (we allow silent overflow, but silent overflow is unlikely if all the characters are ASCII). Two identical files have the same checksum. Write a program to compute the checksum of a file that is supplied as a command-line argument.

2.8. Modify the program in Figure 2.16 so that if no command-line arguments are given, then the standard input is used.

2.9. Write a method that returns `true` if `String str1` is a prefix of `String str2`. Do not use any of the general string searching routines except `charAt`.

Programming Projects

2.10. Write a program that outputs the number of characters, words, and lines in the files that are supplied as command-line arguments.

2.11. In Java, floating point divide-by-zero is legal and does not result in an exception (instead, it gives a representation of infinity, negative infinity, or a special not-a-number symbol).

a. Verify the above description by performing some floating point divisions.

b. Write a static `divide` method that takes two parameters, and returns their quotient. If the dividend is 0.0, throw an `ArithmeticException`. Is a throws clause needed?

c. Write a `main` program that calls `divide` and catches the `ArithmeticException`. In which method should the catch clause be placed?

2.12. Implement a text file copy program. Include a test to make sure that the source and destination files are different.

2.13. Each line of a file contains a name (as a string) and an age (as an integer).

a. Write a program that outputs the oldest person; in case of ties, output any person.

b. Write a program that outputs the oldest person; in case of ties, output all oldest people (*Hint*: maintain the current group of oldest people in an `ArrayList`).

References

More information can be found in the references at the end of Chapter 1.