CONTENTS

Chapter 8. Basic Java Syntax

Topics in This Chapter

- Primitive types
- Arithmetic, logical, and relational operators
- Conditional expressions and loops
- Mathematical methods
- Keyboard input and console output
- Execution of local non-Java programs
- References
- Strings
- Arrays
- Vectors
- Simple example of building a binary tree
- Exceptions

Now that you have a handle on object-oriented programming (Chapter 7), you are ready for a whirlwind tour of the basic syntax of the Java programming language. If you are a C++ programmer, you can skim much of the material: primitive types, operators, and loops are pretty similar to the C++ versions. However, you'll want to look more closely at later sections, including input and output (Section 8.5), execution of non-Java programs (Section 8.6), reference types (Section 8.7), vectors (Section 8.10), and exceptions (Section 8.12); they may be new to you.

8.1 Rules of Syntax

As you start writing real-world programs and try to decipher cryptic error messages from your compiler, it is nice to know the rules of the game. This short section describes some of the lexical and grammatical rules in Java.

Careful use of descriptive comments, white space, and indentation can make your source code easier to read. Two types of comments are permitted: block style and line style. Block-style comments are contained between `/*` (slash-star) and `*/` (star-slash) tokens; they can span multiple lines and are sometimes used to quickly comment-out a large block of code during testing. Line-style comments begin with two slashes, `//`; these comments extend just until the end of the line of code.

White space refers to a sequence of nondisplaying characters, for example, space, tab, and newline characters, which are used to separate pieces of code. White space is used principally for clarity, for example, to separate individual statements into different lines, to distinguish arguments in a function call, or to make expressions using mathematical operators easier to read. Another major use of white space is to provide indentation that helps to offset a section of code, typically sections of code in looping or if-else structures. Here is an example using comments, white space, and indentation:

```
/* Block style comments. These may span multiple
   lines. They may not be nested. */

// Single-line comment

while (int i=1; i<=5; i++) {
  if (i==2)
    System.out.println("Tea for two.");
  else
    System.out.println("Not two.");
}
```

All of the comments and most of the spaces in this example are optional. The following code snippet would be treated identically to the example above by a Java compiler. In fact, this code could be expressed on a single line and the compiler would still not complain.

```
while(int i=1;i<=5;i++){if(i==2)
System.out.println("Tea for two.");else
System.out.println("Not two.");}
```

You might ask, "Why is the space in the fragment '`int i`' required?" The answer is that this space separates two tokens, the Java keyword `int` and the variable `i`. In the same way that English sentences are made up of words, Java programs consist of tokens such as numbers, variable names, punctuation characters, and special keywords. Just as we use rules of grammar to properly construct sentences when we speak or write in human languages, computers use grammatical rules to make programs clear and unambiguous. Two of the key grammatical rules regarding tokenization are:

1. Tokens are case sensitive. For example, `While` or `WHILE` are not considered the same as the token `while`.

2. Identifiers (i.e., the names of variables and methods), must start with an alphabetical character or an underscore (_) and may contain letters, numbers, and underscores. Technically, dollar signs and other currency symbols are also allowed where underscores are permitted; however, such use is not recommended since some compilers use dollar signs as special symbols internally.

Finally, separators are special symbols used to delimit parts of code. Semicolons, periods, commas, parentheses, and braces all act as separators: semicolons indicate the end of an expression, periods qualify variable and method identifiers, commas separate arguments in a method call, parentheses clarify arithmetic expressions and enclose argument lists, curly braces denote the start and end of control blocks, and square brackets indicate the use of arrays.

For additional details about Java syntax, refer to *The Java Language Specification,* which you can view on-line at: http://java.sun.com/docs/books/jls/.

8.2 Primitive Types

Java has two fundamental kinds of data types: *primitive* and *reference.* Primitive types are those simple types that are not "objects" (described in the previous chapter)—integers, characters, floating-point numbers, and the like. There are eight primitive types: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`. A ninth type, `void`, is used only to indicate when a method does not return a value.

boolean This is a type with only two possible values: `true` and `false`. A boolean is an actual type, *not* merely a disguised `int`. For example:

```
boolean flag1 = false;
boolean flag2 = (6 < 7); // true
boolean flag3 = !true;   // false
boolean flag4 = 0;       // compiler error!
```

char This is a 16-bit unsigned integer value representing a Unicode character. You can specify values numerically or with a character enclosed inside single quotes. These characters can be keyboard chars, unicode escape chars (`\uxxxx` with *x* in hex), or one of the special escape sequences `\b` (backspace), `\t` (tab), `\n` (newline), `\f` (form feed), `\r` (carriage return), `\"` (double quote), `\'` (single quote), or `\\` (backslash). For instance:

```
char c0= 3;
char c1 = 'Q';
char c2 = '\u0000'; // Smallest value
char c3 = '\uFFFF'; // Biggest value
char c4 = '\b';     // Backspace
```

Learn more about the Unicode standard at http://www.unicode.org/.

byte This is an 8-bit, signed, two's-complement integer. See the description of `int` for ways to represent integral values. Bytes can range in value from `-`128 to `+`127.

short This is a 16-bit, signed, two's-complement integer. See the description of `int` for ways to represent integral values. Shorts range in value from `-`32768 to `+`32767.

int This is a 32-bit, signed, two's-complement integer. You can specify integer constants in base 10 (`1`, `10`, and `100` for 1, 10, and 100, respectively), octal with a prefix of `0` (`01`, `012`, `0144` for 1, 10, and 100), or hex by using a prefix of `0x` (`0x1`, `0xA`, `0x64` for 1, 10, and 100). For hexadecimal numbers, you can use uppercase or lowercase for `x`, `A`, `B`, `C`, `D`, `E`, and `F`. Ints range from `Integer.MIN_VALUE` (-2^{31}) to `Integer.MAX_VALUE` ($2^{31} - 1$). For example:

```
int i0 = 0;
int i1 = -12345;
int i2 = 0xCafeBabe; // Magic number of .class files
int i3 = 0777;       // Octal number, 511 in decimal
```

long This is a 64-bit, signed, two's-complement integer. Use a trailing `L` for literals. A lowercase `l` is also legal to designate a long literal, but discouraged. Depending on the

display font, the lowercase `l` can be confused with a numerical 1 (one). You can use base 10, 8, or 16 for values. Longs range from `Long.MIN_VALUE` (-2^{63}) to `Long.MAX_VALUE` ($2^{63} - 1$). Some examples:

```
long a0 = 0L;
long a1 = -12345l;
long a2 = 0xBabeL; // Tower of Babel?
long a3 = -06767l;
```

Except for `char`, the integral types (`byte`, `char`, `short`, `int`, and `long`) are signed values. There is no built-in support for unsigned integer values in Java as in the C or C++ programming languages. When using integral types, especially smaller types such as `byte` or `short`, you should ensure that the numbers you wish to represent do not exceed the representable range of the primitive type. For example, if you want to represent the day of the year, you cannot use a `byte`, because it can only store a number as large as 127. Instead, you need to use a `short`, `int`, or `long`. An `int` is probably sufficient for most applications.

float This is a 32-bit, IEEE 754 floating-point number. You must use a trailing `f` or `F` to distinguish single precision floating-point literals. In the example below, the `f` is required or else the compiler interprets the constant as a double precision floating-point number that needs a cast before being assigned to a float:

```
float f0 = -1.23f;
```

You can also use an `e` or `E` to indicate an exponent (power of 10). For instance, the first expression below assigns the value 6.02 x10^{23} to the variable `f1` while the second gives the value 4.5 x 10^{-17} to `f2`.

```
float f1 = 6.02E23F;
float f2 = 4.5e-17f;
```

Floats range from `Float.MIN_VALUE` (1.4 x 10^{-45}) to `Float.MAX_VALUE` (3.4 x 10^{38}). Floating-point arithmetic never generates an exception, even in divide-by-zero cases. The `Float` class defines the constants `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`, and `NaN` (not-a-number) to use for some of these special cases. Use `Float.isNaN` to compare to `NaN` because `(Float.NaN == Float.NaN)` returns `false`, in conformance with the IEEE specification, which states that all comparisons to NaNs should fail.

double This is a 64-bit, IEEE 754 floating-point number. You are allowed to append a trailing `d` or `D`, but it is normally omitted. Some examples:

```
double d0 = 1.23;
double d2 = -4.56d;
double d3 = 6.02214E+23;
double d4 = 1e-99;
```

Doubles range from `Double.MIN_VALUE` (4.9 x 10^{-324}) to `Double.MAX_VALUE` (1.7 x 10^{308}). As with `float`, double-precision arithmetic never generates an exception. To handle unusual situations, the `Double` class also defines fields named `POSITIVE_INFINITY`, `NEGATIVE_INFINITY`, and `NaN`. Use `Double.isNaN` to compare a `double` to `Double.NaN`.

Primitive-Type Conversion

The Java programming language requires explicit *typecasts* to convert a value stored in a larger primitive type into a type with a smaller range or different domain. For example:

```
Type2 type2Var = (Type2)type1Var;
```

Truncation or loss of some significant digits may result from these casts, but even so, an exception is not thrown. For instance:

```
int i = 3;
byte b = (byte)i; // Cast i to a byte; no effect
long x = 123456L;
short s = (short)x; // Cast x to a short; lossy
```

Note that in the last example `s` does not contain an equivalent value to `x` because `123456` is not representable as a 16-bit integer quantity. In addition to downward conversions, those from larger to smaller types, conversions from floating-point to integer types also require a cast:

```
double d = 3.1416;
float f = (float)d; // Cast from 64 to 32 bits
short s = (short)f; // Cast a float to a short
int i = s; // Upward conversion, no cast is required
```

To perform rounding instead of truncation when performing cast conversions, use the rounding methods in the `Math` class; see Section 8.4 for more information. Note that numeric types may not be cast to `boolean`, or vice versa. The Java Language Specification, http://java.sun.com/docs/books/jls/, explains in detail the cases when casts are required between types.

8.3 Operators, Conditionals, Iteration

This section describes the basic arithmetic operators (`+`, `-`, `*`, and so forth), conditionals (`if`, and `switch`), and looping constructs (`while`, `do`, and `for`). Some languages such as C++ permit programmers to adopt built-in operators as a syntactic shorthand for user-defined classes. For example, a C++ programmer might write the following code to add two geometric points together:

```
Point p1(3,4), p2(1,3); // C++ constructor syntax
p1 += p2; // To translate point p1 by p2
```

The Java programming language does not allow programmers to overload operators, a technique which some programmers find confusing when they try it with user-defined types. In such cases, method calls are used in the Java programming language instead, for example:

```
p1.translate(p2);
```

Arithmetic Operators

Table 8.1 summarizes the basic numerical operators. You might notice the lack of an exponentiation operator. Don't panic; exponentiation is possible through methods in the `Math` class (Section 8.4). Also, note that `+` can be used for `String` concatenation; see Section 8.8 for details.

Table 8.1. Numerical Operators

Operators	*Meaning*	*Example*
	addition, subtraction	

`+, -`		`x = y + 5;`

`*, /, %`	multiplication, division, remainder	`int x = 3, y = 2;`
		`int z = x / y; // 1`
		
		(integer division truncates)

`++, --`	prefix/postfix increment/ decrement	`int i = 1, j = 1;`
		`int x = i++; // x=1, i=2`
		`int y = ++j; // y=2, j=2`

`<<, >>, >>>`	signed and unsigned shifts	`int x = 3;` `int y = x << 2; // 12`
`~`	bitwise complement	`int x = ~127; // -128`

`&, |, ^`	bitwise AND, OR, XOR	`int x = 127 & 2; // 2`
		`int y = 127 | 2; // 127`
		`int z = 127 ^ 2; // 125`

Java, like C and C++, lets you write

```
var op= val;
```

as a shorthand for

```
var = var op val;
```

For example, instead of

```
i = i + 5;
x = x * 3;
```

you can write

```
i += 5;
x *= 3;
```

Conditionals

The Java programming language has three conditional constructs: `if`, `switch`, and "`? :`". They are summarized in Table 8.2, with more details given in the following section. If you are familiar with C or C++, you can skip this section because these operators are virtually identical to their C and C++ counterparts, except that conditions must be `boolean` valued. Table 8.3 summarizes the `boolean` operators that are frequently used in conjunction with these three conditionals.

Table 8.2. Conditionals

Operator	***Standard Forms***

`if`	`if (boolean-expression) {`
	`  statement;`
	`}`
	`if (boolean-expression) {`
	`  statement1;`
	`} else {`
	`  statement2;`
	`}`

`? :`	`boolean-expression ? val1 : val2;`

switch	switch(someInt) {
	  case val1: statement1;
	  break;
	  case val2: statement2a;
	  statement2b;
	  break;
	  ...
	  default: statementN;
	}

Table 8.3. Boolean Operators

Operator	*Meaning*
==, !=	Equality, inequality. In addition to comparing primitive types, == tests if two objects are identical (the same object), not just if they appear equal (have the same fields).
<, <=, >, >=	Numeric less than, less than or equal to, greater than, greater than or equal to.
&&, ||	Logical AND, OR. Both use short-circuit evaluation to more efficiently compute the results of complicated expressions.
!	Logical negation.

if (boolean-expression) statement

if (boolean-expression) statement1 else statement2

The `if` keyword expects a `boolean` expression in parentheses. If the expression evaluates to `true`, the subsequent statement is executed. If the expression is `false`, the `else` statement (if present) is executed. Supplying a non-`boolean` expression results in a compile-time error, unlike the case with C and C++, which treat anything not equal to zero as `true`. Here is an example that returns the larger of two integers:

```
// See Math.max
public static int max2(int n1, int n2) {
  if (n1 >= n2)
    return(n1);
  else
    return(n2);
}
```

You can combine multiple statements in the if-else clause by enclosing them in curly braces. For instance:

```
public static int max2Verbose(int n1, int n2) {
  if (n1 >= n2) {
    System.out.println(n1 + " is larger.");
    return(n1);
  } else {
    System.out.println(n2 + " is larger.");
    return(n2);
  }
}
```

Next, note that an `else` always goes with the most recent preceding `if` that does not already have a matching `else`. For example:

```
public static int max3(int n1, int n2, int n3) {
  if (n1 >= n2)
    if (n1 >= n3)
      return(n1);
    else
      return(n3);
  else
    if (n2 >= n3)
      return(n2);
    else
      return(n3);
}
```

Two more points should be made here. First of all, note that the nested (indented) `if` statements are considered single statements; thus, curly braces are not required even though the statements flow over several lines. More importantly, see how indentation makes the association of the `else` clauses clearer.

Several different indentation styles are consistent with this approach. Following are a few of the most popular; we use the first style throughout most of the book.

```
// Indentation Style 1
public SomeType someMethod(...) {
  if {
    statement1;
    ....
    statementN
  } else {
    statementA;
    ...
    statementZ;
  }
}

// Indentation Style 2
public SomeType someMethod(...)
{ if
  { statement1;
    ...
    statementN;
  } else
  { statementA;
    ...
    statementZ;
  }
}

// Indentation Style 3
public SomeType someMethod(...)
{
  if
  {
    statement1;
```

```
    ...
    statementN;
  }
  else
  {
    statementA;
    ...
    statementZ;
  }
}
```

In certain cases, you can simplify multiple comparisons by combining them using logical operators `&&` (logical AND) or `||` (logical OR). As in C and C++, these operators perform "short circuit" evaluation, which means that they return an answer as soon as they have enough information to do so (even if all the subexpressions have not been evaluated). In particular, a comparison with `&&` evaluates to `false` whenever the leftmost conjunct is `false`; the right side is not then evaluated. Likewise, an expression with `||` immediately evaluates as `true` if the leftmost disjunct is `true`. Here's an example:

```
public static int max3(int n1, int n2, int n3) {
  if ((n1 >= n2) && (n1 >= n3))
    return(n1);
  else if ((n2 >= n1) && (n2 >= n3))
    return(n2);
  else
    return(n3);
}
```

There are single character forms of the logical operators, `&` and `|`, that do evaluate all of their operands. However, they are rarely used and may be confused with the bitwise arithmetic operators.

boolean-expression ? thenValue : elseValue

Since `if` statements do not return a value, Java provides a shortcut when the purpose of the `if` is to assign to a single variable. For example, you can express the following four lines

```
if (someCondition)
  someVar = value1;
else
  someVar = value2;
```

more simply as

```
someVar = (someCondition ? value1 : value2);
```

In fact, you can use this form whenever you need to return a value, although overuse can make code difficult to read. Here is a variation of `max2` that uses it:

```
public static int max2Short(int n1, int n2) {
  return((n1 >= n2) ? n1 : n2);
}
```

switch(integralExpression) { switchBody }

The `switch` construct provides a compact way to compare an expression to a variety of integer types (`char`, `byte`, `short`, or `int`, but not `long`). The idea is to supply an integer expression and then provide, inside the `switch` body, one or more `case` statements that designate different possible values of the expression. When one matches, it *and all subsequent cases* are executed. Here is an example that uses `switch` to generate the string representation of a single-digit integer:

```
public static String number(int digit) {
  switch(digit) {
    case 0: return("zero");
    case 1: return("one");
    case 2: return("two");
    case 3: return("three");
    case 4: return("four");
    case 5: return("five");
    case 6: return("six");
    case 7: return("seven");
    case 8: return("eight");
    case 9: return("nine");
    default: return("Not a single digit");
  }
}
```

Section 8.9 (Arrays) describes a simpler way to perform this integer-to-string conversion by using arrays. The most confusing thing about `switch` statements is that code "falls through" cases; *all* statements after the first matching `case` are executed. This is handy in some situations because it lets you combine cases, as follows:

```
switch(val) {
  case test1:
  case test2:
    actionForTest1and2();
  ...
}
```

However, it can catch you by surprise if the `case` statements do not contain an explicit `return` or `break`. For example, consider the following verbose variation of the `number` method:

```
// Incorrect version that forgets about case
// fall-through.

public static String numberVerbose(int digit) {
  String result;
  switch(digit) {
    case 0:  System.out.println("zero");
             result = "zero";
    case 1:  System.out.println("one");
             result = "one";
    case 2:  System.out.println("two");
             result = "two";
```

```
    case 3:  System.out.println("three");
             result = "three";
    case 4:  System.out.println("four");
             result = "four";
    case 5:  System.out.println("five");
             result = "five";
    case 6:  System.out.println("six");
             result = "six";
    case 7:  System.out.println("seven");
             result = "seven";
    case 8:  System.out.println("eight");
             result = "eight";
    case 9:  System.out.println("nine");
             result = "nine";
    default: System.out.println(
               "Not a single digit");
             result = "Not a single digit";
  }
  return(result);
}
```

Because there is no explicit direction to exit the `switch` after the first match, multiple cases are executed. For instance, here is the output when `numberVerbose(5)` is called:

```
five
six
seven
eight
nine
Not a single digit
```

The standard solution is to use the `break` statement to exit the `switch` after the first match, as in the following corrected version:

```
public static String numberVerboseFixed(int digit) {
  String result;
  switch(digit) {
    case 0: System.out.println("zero");
            result = "zero";
            break;
    case 1: System.out.println("one");
            result = "one";
            break;
    ...
    default: System.out.println("Not a single digit");
             result = "Not a single digit";
  }
  return(result);
}
```

Loops

The Java programming language supports the same basic looping constructs as do C and C++: `while`, `do`, and `for`. These constructs are summarized in Table 8.4 and described in more detail following the table. In addition, Java supports the `break` and `continue` keywords, which are used to exit the loop or to interrupt the body and restart at the beginning, respectively.

Table 8.4. Looping Constructs

Construct	*Standard Form*

`while`	`while (continueTest) {`
	`  body;`
	`}`

`do`	`do {`
	`  body;`
	`} while (continueTest);`

`for`	`for(init; continueTest; updateOp) {`
	`  body;`
	`}`

while loops The `while` construct tests the supplied `boolean` continuation test, executing the body as long as the test returns `true`. For example, the following method prints out the numbers in the range from 0 to `max`.

```
public static void listNums1(int max) {
  int i = 0;
  while (i <= max) {
    System.out.println("Number: " + i);
    i++;
  }
}
```

Executing `listNums1(5)` results in the following output:

```
0: zero
1: one
2: two
3: three
4: four
```

do loops The `do` construct differs from `while` in that the test is evaluated after the loop body rather than before. This means that the body will always be executed at least once, regardless of the test's value. Following is a variation of the `listNums` method, using `do`. The result of `listNums2(5)` is identical to that shown for `listNums1`(5); however, `listNums2(-5)` would print "`Number: 0`", while `listNums1(-5)` would print nothing. The reason is that with a `do` loop the test is performed after the loop body, whereas a `while` loop tests before the loop body. In `listNums1(-5)`, the loop body is never entered.

```
public static void listNums2(int max) {
  int i = 0;
  do {
    System.out.println("Number: " + i);
    i++;
  } while (i <= max); // Don't forget the semicolon
}
```

Forgetting the final semicolon in a `do` loop is a common syntax error.

for loops The `for` construct is by far the most common way to create loops with numbered counters. First, the init part is executed once. Then, as long as the continuation test evaluates to `true`, the statement is executed and then the update operation is performed. Here's an example that gives the same result as the previous two versions when passed a positive argument.

```
public static void listNums3(int max) {
  for(int i=0; i<max; i++) {
    System.out.println("Number: " + i);
  }
}
```

It is also legal to omit any (or all) of the three `for` clauses. A missing `continueTest` is treated as `true`. Thus,

```
for(;;) { body }
```

and

```
while(true) { body }
```

are equivalent. These forms are occasionally used when the termination test cannot be easily placed in the initial `for` or `while` clause. Instead, the body contains a conditional `return` or `break`.

The various forms of loops, `while`, `do`, and `for`, are all equally expressive, and any loop that can be written with one can be rewritten with another instead. Individual programmers usually adopt a style for using each. Some programmers tend to use `for` loops predominantly, especially when looping over arrays or ranges of numbers. Some prefer `while` loops in cases where the number of times the loop will be executed is not generally known in advance, for example, based on the number of lines in an input file.

Listing 8.1 contains an example of using a `while` loop to determine how long it takes a ball to fall from the top of the Washington Monument. The body of the loop is executed seven times. The loop is terminated when the ball has dropped a distance greater than the height of the monument; thus, the ball falls for slightly less than 6 seconds. The output for `DropBall` is shown in Listing 8.2.

Listing 8.1 `DropBall.java`

```
/** Simulating dropping a ball from the top of the Washington
 *  Monument. The program outputs the height of the ball each
 *  second until the ball hits the ground.
 */

public class DropBall {
  public static void main(String[] args) {
    int time = 0;
    double start = 550.0, drop = 0.0;
    double height = start;
    while (height > 0) {
      System.out.println("After " + time +
                  (time==1 ? " second, " : " seconds,") +
```

```
                    "the ball is at " + height + " feet.");
      time++;
      drop = freeFall(time);
      height = start - drop;
    }
    System.out.println("Before " + time + " seconds could " +
                       "expire, the ball hit the ground!");
  }

  /** Calculate the distance in feet for an object in
   *  free fall.
   */

  public static double freeFall (float time) {
    // Gravitational constant is 32 feet per second squared
    return(16.0 * time * time); // 1/2 gt^2
  }
}
```

Listing 8.2 Output from `DropBall`

```
Prompt> java DropBall
After 0 seconds,the ball is at 550.0 feet.
After 1 second, the ball is at 534.0 feet.
After 2 seconds,the ball is at 486.0 feet.
After 3 seconds,the ball is at 406.0 feet.
After 4 seconds,the ball is at 294.0 feet.
After 5 seconds,the ball is at 150.0 feet.
Before 6 seconds could expire, the ball hit the ground!
```

8.4 The Math Class

`Math` provides a range of arithmetic methods not available as built-in operators. All of these methods are `static`, so there would never be any reason to create an instance of the `Math` class.

Constants

public static final double E

This constant is *e,* the base for natural logarithms, 2.7182818284590452354.

public static final double PI

This constant is π, 3.14159265358979323846.

General-Purpose Methods

public static int abs(int num)

public static long abs(long num)

public static float abs(float num)

public static double abs(double num)

These methods return the absolute value of the specified number.

public static double ceil(double num)

public static double floor(double num)

The `ceil` method returns a `double` corresponding to the smallest integer greater than or equal to the specified number; `floor` returns a `double` corresponding to the largest integer less than or equal to the number.

public static exp(double num)

This method returns e^{num}.

public static double IEEEremainder(double f1, double f2)

This method returns the remainder of `f1` divided by `f2`, as specified in the IEEE 754 standard.

public static double log(double num)

This method returns the natural logarithm of the specified number. Java does not provide a method for calculating logs in other common bases (e.g., 10 or 2), but following is a method that does this computation, using the relationship:

$$\log_{b1}(n) = \frac{\log_{b2}(n)}{\log_{b2}(b1)}$$

```
public static double log(double num, double base) {
  return(Math.log(num) / Math.log(base));
}
```

public static int max(int num1, int num2)

public static long max(long num1, long num2)

public static float max(float num1, float num2)

public static double max(double num1, double num2)

public static int min(int num1, int num2)

public static long min(long num1, long num2)

public static float min(float num1, float num2)

public static double min(double num1, double num2)

These methods return the bigger (`max`) or smaller (`min`) of the two numbers.

public static double pow(double base, double exponent)

The `pow` method returns $base^{exponent}$.

public static double random()

This method returns a random number from `0.0` (inclusive) to `1.0` (exclusive). For more control over random numbers, use the `java.util.Random` class.

public static double rint(double num)

public static int round(float num)

public static long round(double num)

These methods round the number toward the nearest integer. They differ in their return types and what they do for a number of the form `xxx.5`. The `round` methods round up in such a case; `rint` rounds to the nearest even number, as specified in the IEEE 754 standard. Although less intuitive, the behavior of `rint` avoids skewing sums of rounded numbers upwards.

public static double sqrt(double num)

This method returns $\sqrt{num}$ for nonnegative numbers and returns `Double.NaN` if the input is `NaN` or is negative.

Trigonometric Methods

public static double sin(double radians)

public static double cos(double radians)

public static double tan(double radians)

These methods return the sine, cosine, and tangent of the specified number, interpreted as an angle *in radians.*

public static double toDegrees(double radians)

public static double toRadians(double degrees)

These methods perform conversions between angles expressed in radians and degrees. For example, the following computes the sine of 60 degrees:

```
Math.sin(Math.toRadians(60.0))
```

public static double acos(double val)

public static double asin(double val)

public static double atan(double val)

These methods return the arc cosine, arc sine, and arc tangent of the specified value. The result is expressed *in radians.*

public static double atan2(double y, double x)

This method returns the θ part of the polar coordinate (r,θ) that corresponds to the cartesian coordinate (x,y). This is the `atan` of y/x that is in the range -π to π

BigInteger and BigDecimal

The Java programming language supports two arbitrary-precision number formats: `java.math.BigInteger` and `java.math.BigDecimal`. These classes contain methods for addition, multiplication, division, exponentiation, primality testing, greatest common divisors, and more. See the API for details, but the key point is that these classes can be used to obtain any desired level of accuracy. For instance, *every* digit in a `BigInteger` is significant, and it cannot overflow. To illustrate this, Listing 8.3 uses `BigInteger` to represent the exact value of *N*! (the factorial of *N,* i.e., (*N*)(*N* - 1)(*N* - 2) … (1)) for large values of *N.* Listing 8.4 shows the result.

Listing 8.3 `Factorial.java`

```
import java.math.BigInteger;

/** Computes an exact factorial, using a BigInteger. */

public class Factorial {
  public static void main(String[] args) {
    for(int i=1; i<=256; i*=2) {
      System.out.println(i + "!=" + factorial(i));
    }
  }

  public static BigInteger factorial(int n) {
    if (n <= 1) {
      return(new BigInteger("1"));
    } else {
      BigInteger bigN = new BigInteger(String.valueOf(n));
      return(bigN.multiply(factorial(n - 1)));
    }
  }
}
```

Listing 8.4 `Factorial` Output

```
Prompt> java Factorial
1!=1
2!=2
4!=24
8!=40320
16!=20922789888000
32!=263130836933693530167218012160000000
64!=126886932185884164103433389335161480802865516174545192198801894375214704230400000000000000
128!=3856204823625804217356770659234636406174931095902235902788284032763734025751655435606861685885073615340300518330589163475921729322624988577661149552450393577600346447092792476924955852800000000000000000000000000
256!=857817775342842654119082271681232625157781520279485619859655650377269452553147589377440291360451408450375885342336584306157196834693696475322289288497426025679637332563368786442675207626794560187968867971521143307702077526646451464709187326100832876325702818980773671781454170250523018608495319068138257481070252817559459476987034665712738139286205234756808218860701203611083152093501947437109101726968262861606263662435022840944191408424615936000000
```

```
000000000000000000000000000000000000000000000000000000000000
```

Core Warning

Do not confuse the `java.math` package, where classes `BigInteger` and `BigDecimal` are found, with the `java.lang.Math` class. The `Math` class is more commonly used; it provides all of the methods in the standard mathematical library.

8.5 Input and Output

The simplest forms of I/O are discussed in this section: printing to standard output and reading from standard input.

Printing to Standard Output

You've already seen examples of printing with `System.out.println`. As you probably guessed from the Java naming conventions, `out` is a `static` variable in the `System` class, and `println` is a method in the class that belongs to `out`. That class is `PrintStream`; it also contains a `print` method that works just like `println` except that it omits the trailing newline. Both `println` and `print` take a single argument of any type. Primitive types are converted to strings by the `String.valueOf` method, and non-`String` objects are converted to strings by their `toString` method. Longer strings can be formed from components joined through the `+` concatenation operator. For example:

```
System.out.println(
   2 + 'B' + " || ! " + new Integer(2) + "B");
```

In addition to `println` and `print`, the `PrintStream` class also contains a `flush` method. This method is useful when you want to be sure that output has not been cached (e.g., when reading from standard input just after writing a prompt). For full details, see the on-line API for `java.io.PrintStream`.

Surprisingly, Java's way of controlling the spacing and formatting of numbers is not as simple as C's `printf` or `sprintf` functions. The `java.text.DecimalFormat` class provides this sort of functionality indirectly. `DecimalFormat` works by creating an object that describes the desired formatting of your number(s). You then use this object's `format` method as needed to convert numbers into formatted strings. Listing 8.5 gives a simple example with results shown in Listing 8.6.

Listing 8.5 `NumFormat.java`

```
import java.text.*;

/** Formatting real numbers with DecimalFormat. */

public class NumFormat {
  public static void main (String[] args) {
    DecimalFormat science = new DecimalFormat("0.000E0");
    DecimalFormat plain = new DecimalFormat("0.0000");

    for(double d=100.0; d<140.0; d*=1.10) {
      System.out.println("Scientific: " + science.format(d) +
                         " and Plain: " + plain.format(d));
```

```
    }
  }
}
```

Listing 8.6 `NumFormat` Output

```
Prompt> java NumFormat
Scientific: 1.000E2 and Plain: 100.0000
Scientific: 1.100E2 and Plain: 110.0000
Scientific: 1.210E2 and Plain: 121.0000
Scientific: 1.331E2 and Plain: 133.1000
```

In the previous example, the `DecimalFormat` constructor used expects a `String` defining the pattern in which to display the number when converting to a `String` by the format method. The pattern can indicate the number of digits to display and possible separators. Table 8.5 summarizes some of the available characters for creating the formatting pattern. For example, `,###.0` would format a number with a single digit after the decimal point and would display a comma separator if the number is in the thousands: the number `23767.82` would format as `23,767.8`, the number `0.43` would format as `.4` (no leading zero).

`DecimalFormat` provides only coarse control when printing numbers. If you find `DecimalFormat` too cumbersome, you can use one of the public domain `printf` substitutes. One of the best ones is from Jef Poskanzer; see http://www.acme.com/java/software/Acme.Fmt.html.

Finally, output sent to `System.out` can be redirected to a different stream instead. One situation where redirection is useful occurs when you want to produce a log file instead of sending messages to the console window. To accomplish this, you simply call `System.setOut` once with the `PrintStream` to which you want to send output.

Table 8.5. Formatting Characters

Symbol	*Meaning*
`0`	Placeholder for a digit.
`#`	Placeholder for a digit. If the digit is leading or trailing zero, then don't display.
`.`	Location of decimal point.
`,`	Display comma at this location.
`-`	Minus sign.
`E`	Scientific notation. Indications the location to separate the mantissa from the exponent.
`%`	Multiply the value by 100 and display as a percent.

Printing to Standard Error

In addition to the `System.out` variable, the `System.err` variable can be used on operating systems that maintain a distinction between normal output and error output. It, too, is a `PrintStream` and can be used in exactly the same way as `System.out`. It also can be redirected; you use `System.setErr` for this purpose.

Reading from Standard Input

It is relatively uncommon to read from standard input in the Java programming language. Nongraphical applications typically use the command-line arguments for the data they need, and graphical programs use a textfield, button, slider, or other GUI control. If you *do* need to read input

this way, the standard approach is first to turn `System.in` into a `BufferedReader` as follows:

```
BufferedReader keyboard =
  new BufferedReader(
    new InputStreamReader(System.in));
```

Then, you can use `keyboard.readLine` to retrieve a line of input:

```
String line = keyboard.readLine();
```

8.6 Execution of Non-Java Programs

Because of default security restrictions, applets (Java programs embedded in Web pages) cannot execute system programs; however, applications (stand-alone Java programs) can execute system programs. Starting a local program involves the following four steps:

1. **Get the special Runtime object.** Use the static `getRuntime` method of the `Runtime` class for this, as follows:

   ```
   Runtime rt = Runtime.getRuntime();
   ```

2. **Execute the program.** Use the `exec` method, which returns a `Process` object, as illustrated below:

   ```
   Process proc = rt.exec("someProgram");
   ```

   This starts the program but does *not* wait for the program to terminate or print the program results. Note that the `exec` method does not make use of your `PATH` environment variable (used on Windows 98/NT and Unix to identify where to look for programs), so you need to specify the full pathname. Also, `exec` does not start a shell, so special shell characters (such as `>` or `|` in Unix) will not be recognized. Finally, note that the `Runtime` class provides versions of `exec` that take an array of strings and pass the strings as command-line arguments to a program. For example,

   ```
   String[] args = { "-l", "*.java" };
   rt.exec("ls", args); // Directory listing.
   ```

3. **Optional: wait for the program to exit.** The `exec` method returns immediately, regardless of how long it takes the program to run. This behavior enables you to start long-running programs such as Netscape and still continue processing; however, this behavior catches many users off guard. If you want the program to wait until the program exits before returning, use the `waitFor` method after you start the process:

   ```
   proc.waitFor();
   ```

   While you might simply use `exec` to start Netscape, you would add `waitFor` if one program needs to terminate before another can begin. For example, you might call `javac` on a source file and use `java` to execute the result, but the compilation must finish before the program can invoke a new instance of the JVM to execute the complied class file. Note that `waitFor` does not print the program results; it simply waits for the program to finish before returning.

4. **Optional: process the results.** Rather than simply waiting for the program to terminate, you might want to capture its output. For example, telling the system to do a directory listing is not too useful unless you access the results. You can print the results by attaching a `BufferedReader` to the process, then reading from it, as in the following example:

```
    try {
      BufferedReader buffer =
       new BufferedReader(
         new InputStreamReader(proc.getInputStream()));
      String s = null;
      while ((s = buffer.readLine()) != null) {
        System.out.println("Output: " + s);
      }
      buffer.close();
    } catch(Exception e) {
      /* Ignore read errors */
    }
```

Listing 8.7 shows the entire process, wrapping all four steps into an easy-to-use `Exec` class containing `exec`, `execPrint`, and `execWait` methods. Listing 8.8 illustrates its use.

Finally, because starting processes and then communicating with them requires native operating system support, some risk is associated with their use. In particular, there are situations where the virtual machine can hang due to a deadlocked process.

Core Warning

Be judicious when starting system processes. In situations where input or output streams are not processed efficiently, deadlock can occur.

Listing 8.7 `Exec.java`

```
import java.io.*;

/** A class that eases the pain of running external processes
 *  from applications. Lets you run a program three ways:
 *  <OL>
 *    <LI><B>exec</B>: Execute the command, returning
 *        immediately even if the command is still running.
 *        This would be appropriate for printing a file.
 *    <LI><B>execWait</B>: Execute the command, but don't
 *        return until the command finishes. This would be
 *        appropriate for sequential commands where the first
 *        depends on the second having finished (e.g.,
 *        <CODE>javac</CODE> followed by <CODE>java</CODE>).
 *    <LI><B>execPrint</B>: Execute the command and print the
 *         output. This would be appropriate for the Unix
 *         command <CODE>ls</CODE>.
 *  </OL>
 *  Note that the PATH is not taken into account, so you must
 *  specify the <B>full</B> pathname to the command, and shell
 *  built-in commands will not work. For instance, on Unix the
 *  above three examples might look like:
 *  <OL>
 *    <LI><PRE>Exec.exec("/usr/ucb/lpr Some-File");</PRE>
 *    <LI><PRE>Exec.execWait("/usr/local/bin/javac Foo.java");
 *        Exec.execWait("/usr/local/bin/java Foo");</PRE>
```

```
 *    <LI><PRE>Exec.execPrint("/usr/bin/ls -al");</PRE>
 *  </OL>
 */
public class Exec {

  private static boolean verbose = true;

  /** Determines if the Exec class should print which commands
    * are being executed, and prints error messages if a problem
    * is found. Default is true.
    *
    * @param verboseFlag true: print messages, false: don't.
    */

  public static void setVerbose(boolean verboseFlag) {
    verbose = verboseFlag;
  }

  /** Will Exec print status messages? */

  public static boolean getVerbose() {
    return(verbose);
  }

  /** Starts a process to execute the command. Returns
    * immediately, even if the new process is still running.
    *
    * @param command The <B>full</B> pathname of the command to
    * be executed. No shell built-ins (e.g., "cd") or shell
    * meta-chars (e.g.">") are allowed.
    * @return false if a problem is known to occur, but since
    * this returns immediately, problems aren't usually found
    * in time. Returns true otherwise.
    */

  public static boolean exec(String command) {
    return(exec(command, false, false));
  }

  /** Starts a process to execute the command. Waits for the
    * process to finish before returning.
    *
    * @param command The <B>full</B> pathname of the command to
    * be executed. No shell built-ins or shell metachars are
    * allowed.
    * @return false if a problem is known to occur, either due
    * to an exception or from the subprocess returning a
    * nonzero value. Returns true otherwise.
    */

  public static boolean execWait(String command) {
    return(exec(command, false, true));
```

```
}

/** Starts a process to execute the command. Prints any output
  * the command produces.
  *
  * @param command The <B>full</B> pathname of the command to
  * be executed. No shell built-ins or shell meta-chars are
  * allowed.
  * @return false if a problem is known to occur, either due
  * to an exception or from the subprocess returning a
  * nonzero value. Returns true otherwise.
  */

public static boolean execPrint(String command) {
  return(exec(command, true, false));
}

/** This creates a Process object via Runtime.getRuntime.exec()
  * Depending on the flags, it may call waitFor on the process
  * to avoid continuing until the process terminates, and open
  * an input stream from the process to read the results.
  */

private static boolean exec(String command,
                            boolean printResults,
                            boolean wait) {
  if (verbose) {
    printSeparator();
    System.out.println("Executing '" + command + "'.");
  }
  try {
    // Start running command, returning immediately.
    Process p  = Runtime.getRuntime().exec(command);

    // Print the output. Since we read until there is no more
    // input, this causes us to wait until the process is
    // completed.
    if(printResults) {
      BufferedReader buffer = new BufferedReader(
        new InputStreamReader(p.getInputStream()));
      String s = null;
      try {
        while ((s = buffer.readLine()) != null) {
          System.out.println("Output: " + s);
        }
        buffer.close();
        if (p.exitValue() != 0) {
          if (verbose) {
            printError(command + " - p.exitValue() != 0");
          }
          return(false);
        }
```

```
        } catch (Exception e) {
          // Ignore read errors; they mean the process is done.
        }

      // If not printing the results, then we should call waitFor
      // to stop until the process is completed.
      } else if (wait) {
        try {
          System.out.println(" ");
          int returnVal = p.waitFor();
          if (returnVal != 0) {
            if (verbose) {
              printError(command);
            }
            return(false);
          }
        } catch (Exception e) {
          if (verbose) {
            printError(command, e);
          }
          return(false);
        }
      }
    } catch (Exception e) {
      if (verbose) {
        printError(command, e);
      }
      return(false);
    }
    return(true);
  }

  private static void printError(String command,
                                 Exception e) {
    System.out.println("Error doing exec(" + command + "): " +
                       e.getMessage());
    System.out.println("Did you specify the full " +
                       "pathname?");
  }

  private static void printError(String command) {
    System.out.println("Error executing '" + command + "'.");
  }

  private static void printSeparator() {
    System.out.println
      ("================================================");
  }
}
```

Listing 8.8 illustrates the `Exec` class in action on a Unix system, with the result shown in Listing 8.9.

Listing 8.8 `ExecTest.java`

```
/** A test of the Exec class. */

public class ExecTest {
  public static void main(String[] args) {
    // Note: no trailing "&" -- special shell chars not
    // understood, since no shell started. Besides, exec
    // doesn't wait, so the program continues along even
    // before Netscape pops up.
    Exec.exec("/usr/local/bin/netscape");

    // Run commands, printing results.
    Exec.execPrint("/usr/bin/ls");
    Exec.execPrint("/usr/bin/cat Test.java");

    // Don't print results, but wait until this finishes.
    Exec.execWait("/usr/java1.3/bin/javac Test.java");

    // Now Test.class should exist.
    Exec.execPrint("/usr/bin/ls");
  }
}
```

Listing 8.9 `ExecTest` Output

```
Unix> java ExecTest
================================================
Executing '/usr/local/bin/netscape'.
================================================
Executing '/usr/bin/ls'.
Output: Exec.class
Output: Exec.java
Output: ExecTest.class
Output: ExecTest.java
Output: Test.java
================================================
Executing '/usr/bin/cat Test.java'.
Output: public class Test {
Output:   boolean flag;
Output: }
================================================
Executing '/usr/java1.3/bin/javac Test.java'.
================================================
Executing '/usr/bin/ls'.
Output: Exec.class
Output: Exec.java
Output: ExecTest.class
Output: ExecTest.java
Output: Test.class
Output: Test.java
```

8.7 Reference Types

Values that are objects (i.e., class instances or arrays; anything nonprimitive) are known as *reference values* or simply *references.* In the Java world, we normally say that the value of such and such a variable "is" an object. Because the Java programming language has no explicit referencing or dereferencing of pointers or pointer arithmetic, it is commonly but erroneously stated that Java does not have pointers. Wrong! In fact, *all* nonprimitive variables in Java are pointers. So, a C programmer might find it clearer to say that such and such a nonprimitive variable "points to" an object. This is the only kind of nonprimitive type in Java; there is no distinction between variables that *are* objects and variables that *point to* objects as in some languages.

If you are not already familiar with pointers, the basic idea is that you can pass big complicated objects around efficiently; Java doesn't copy them every time you pass them from one method to another. If you've used pointers extensively in other languages, be aware that Java forbids dereferencing pointers; given a referenced object, a method cannot modify a reference so that it refers to a different object.

Listing 8.10 gives an example, with the result shown in Listing 8.11.

Listing 8.10 `ReferenceTest.java`

```
import java.awt.Point;

public class ReferenceTest {
  public static void main(String[] args) {
    Point p1 = new Point(1, 2); // Assign Point to p1
    Point p2 = p1; // p2 is new reference to *same* Point
    print("p1", p1); // (1, 2)
    print("p2", p2); // (1, 2)
    triple(p2); // Doesn't change p2
    print("p2", p2); // (1, 2)
    p2 = triple(p2); // Have p2 point to *new* Point
    print("p2", p2); // (3, 6)
    print("p1", p1); // p1 unchanged: (1, 2)
  }

  public static Point triple(Point p) {
    p = new Point(p.x * 3, p.y * 3); // Redirect p
    return(p);
  }

  public static void print(String name, Point p) {
    System.out.println("Point " + name + "= (" +
                       p.x + ", " + p.y + ").");
  }
}
```

Listing 8.11 `ReferenceTest` Output

```
Prompt> java ReferenceTest
Point p1= (1, 2).
Point p2= (1, 2).
Point p2= (1, 2).
Point p2= (3, 6).
```

```
Point p1= (1, 2).
```

Notice that changing the local variable `p` in the `triple` method didn't change the variable passed in (`p2`); it merely made `p` point someplace new, leaving `p2` referring to the original place. To change `p2` to a new object, we assigned it to the return value of `triple`. Although it is not possible for a method to change where an external variable points (i.e., the *object* to which it refers), it is possible for a method to change the *fields* of an object, assuming that the field's access permissions are appropriate. This is illustrated in Listing 8.12, with the result shown in Listing 8.13.

Listing 8.12 `ModificationTest.java`

```
import java.awt.Point;

public class ModificationTest extends ReferenceTest {
  public static void main(String[] args) {
    Point p1 = new Point(1, 2); // Assign Point to p1
    Point p2 = p1; // p2 is new reference to *same* Point
    print("p1", p1); // (1, 2)
    print("p2", p2); // (1, 2)
    munge(p2); // Changes fields of the *single* Point
    print("p1", p1); // (5, 10)
    print("p2", p2); // (5, 10)
  }

  public static void munge(Point p) {
    p.x = 5;
    p.y = 10;
  }
}
```

Listing 8.13 `ModificationTest` Output

```
Prompt> java ModificationTest
Point p1= (1, 2).
Point p2= (1, 2).
Point p1= (5, 10).
Point p2= (5, 10).
```

Java Argument-Passing Conventions

Now, if you are already familiar with the terms "call by value" and "call by reference," you may be puzzled as to which scheme Java uses. It cannot be call by reference, because the change to `p` in `triple` didn't change the external value. But the convention doesn't look like call by value either, because the `munge` method showed that methods don't get copies of objects. Don't worry about the definitions—simply remember the following rule.

Core Note

*If you pass a variable to a method in Java, the method cannot change which **object** the variable references but might be able to change the **fields** of that object.*

If you are absolutely determined to pin the definition down, then you can say that Java uses call by value, but that the values themselves are references (restricted pointers).

The instanceof Operator

The `instanceof` operator returns `true` only if the left-hand argument is a direct or indirect instance of the class or interface named by the right-hand argument. For example:

```
if (item instanceof Breakable) {
  add(item, chinaCabinet);
}
```

Use this operator with caution; `instanceof` can often be replaced with polymorphism, yielding a faster, simpler, and more maintainable result. One problem with `instanceof` is that you need to know the name of the class or interface when you write your code. To provide a general solution that will work in all situations, a dynamic version of this test was added in Java 1.1. This second approach involves calling the `isInstance` method of an instance of `java.lang.Class` with the object you want to examine. See the example in Listing 8.14 and its output in Listing 8.15.

Listing 8.14 `InstanceOf.java`

```
interface Barking {}

class Mammal {}

class Canine extends Mammal {}

class Dog extends Canine implements Barking {}

class Retriever extends Dog {}

public class InstanceOf {
  public static void main(String[] args) {
    Canine wolf = new Canine();
    Retriever rover = new Retriever();

    System.out.println("Testing instanceof:");
    report(wolf, "wolf");
    System.out.println();
    report(rover, "rover");

    System.out.println("\nTesting isInstance:");
    Class barkingClass = Barking.class;
    Class dogClass = Dog.class;
    Class retrieverClass = Retriever.class;
    System.out.println("  Does a retriever bark? " +
                       barkingClass.isInstance(rover));
    System.out.println("  Is a retriever a dog? " +
                       dogClass.isInstance(rover));
    System.out.println("  Is a dog necessarily a retriever? " +
                       retrieverClass.isInstance(new Dog()));
  }

  public static void report(Object object, String name) {
    System.out.println("  " + name + " is a mammal: " +
                       (object instanceof Mammal));
```

```
    System.out.println("  " + name + " is a canine: " +
                       (object instanceof Canine));
    System.out.println("  " + name + " is a dog: " +
                       (object instanceof Dog));
    System.out.println("  " + name + " is a retriever: " +
                       (object instanceof Retriever));
  }
}
```

Listing 8.15 `InstanceOf` Output

```
prompt> java InstanceOf
Testing instanceof:
  wolf is a mammal: true
  wolf is a canine: true
  wolf is a dog: false
  wolf is a retriever: false

  rover is a mammal: true
  rover is a canine: true
  rover is a dog: true
  rover is a retriever: true

Testing isInstance:
  Does a retriever bark? true
  Is a retriever a dog? true
  Is a dog necessarily a retriever? false
```

8.8 Strings

In Java, strings are real objects, members of the `java.lang.String` class. However, because they are so frequently used, you are allowed to create them simply by using double quotes, as follows:

```
String s1 = "This is a String";
```

The normal object-creation approach of using `new` is legal also, for example,

```
String s2 = new String("This is a String too");
```

but is rarely used.

The most unusual thing about the `String` class is that strings are immutable; once created they cannot be changed. "Hold on," you say, "I know there is no `setCharacterAt` method, but I've seen string concatenation used lots of places." That's a good point; the `+` character can be used to concatenate strings, as follows:

```
String test = "foo" + "bar"; // "foobar"
```

However, in this example *three* strings are created: `foo`, `bar`, and a new third string `foobar`. This distinction doesn't seem important in the previous example but is very significant in the following code:

```
String foo = "foo";
```

```
String bar = "bar";
String test = foo + bar;
```

The key point here is that neither `foo` nor `bar` is modified by the concatenation performed on the third line. This is a convenient feature; it means that it is safe to pass strings to arbitrary methods without worrying about them being modified. On the other hand, to implement this unchangeable nature, Java has to copy the strings when concatenation is performed. This can be expensive, so Java supplies a `StringBuffer` class that is mutable.

Note that `+` is the only overloaded operators in Java; it has a totally different meaning for strings than it has for numbers. You cannot define your own operators or overload existing ones. One other thing you should know about `String` is that it is a final class and therefore you cannot create a subclass of `String`.

String Methods

The Java programming language provides a number of useful methods for working with strings. They are summarized below.

public char charAt(int index)

This method returns the character at the specified location.

public int compareTo(String comparison)

public int compareTo(Object object)

The `compareTo` method compares the current string to the supplied string, character by character, checking Unicode ordering. It returns `0` if the strings are equal (have the same characters), a negative number if the current string is lexicographically less than the comparison, and a positive number otherwise. This method is generally used for determining if strings are in order. The actual number is the difference in Unicode values between the first nonmatching characters, or the difference in lengths if the shorter string is a prefix of the longer one. To satisfy the `Comparable` interface, the method for objects was added in Java 1.2. It acts like the string `compareTo` method but throws a `ClassCastException` if the input is not a string.

public String concat(String suffix)

The `concat` method concatenates two strings, forming a new `String`. The following two forms are identical:

```
String result = someString.concat(someOtherString);
String result = someString + someOtherString;
```

Neither `someString` nor `someOtherString` is modified in these examples; instead a new `String` is created.

public static String copyValueOf(char[ ] characters)

public static String copyValueOf(char[ ] data, int startIndex, int count)

These static methods convert character arrays to strings.

public boolean endsWith(String suffix)

This method checks for a suffix of a string.

public boolean equals(Object comparison)

If the comparison object is not a `String`, `equals` returns `false`. Otherwise, it compares character by character. Thus, two different strings with the same characters will be `equals` but not `==`. For example, Listing 8.16 compares the first input argument to a fixed string, using `equals` and `==`. As Listing 8.17 shows, the `==` test fails but the `equals` test succeeds. Also note that different occurrences of literal strings may or may not be `==` since the compiler may collapse such constants.

Core Warning

Two different `String` objects that contain the same characters will not be `==`. They will, however, be `equals`. In general, two different objects are not `==` even when their fields have identical values.

public boolean equalsIgnoreCase(String comparison)

This method performs a case-insensitive, character-by-character comparison.

public byte[ ] getBytes()

public byte[ ] getBytes(String encoding)

These methods convert a string to a byte array.

public void getChars(int sourceStart, int sourceEnd, char[ ] destination, int destinationStart)

This method copies the characters from `sourceStart` (inclusive) to `sourceEnd` (exclusive) into the specified part of the destination array.

public int indexOf(int character)

public int indexOf(int character, int startIndex)

public int indexOf(String subString)

public int indexOf(String subString, int startIndex)

These methods return the index of the first occurrence of the specified target.

public native String intern()

The `intern` method returns a canonical `String` containing the same characters as the supplied string. The interned result of two strings is `==` if and only if the strings themselves are `equals`.

public int lastIndexOf(int character)

public int lastIndexOf(int character, int startIndex)

public int lastIndexOf(String subString)

public int lastIndexOf(String subString, int startIndex)

These methods return the index of the last occurrence of the specified target.

public int length()

This method gives the length of the string. Note that this is a method call, not an instance variable. So, don't forget that for strings you have to do

```
int len = someString.length(); // length()
```

and for arrays you do

```
int len = someArray.length; // No parens
```

public boolean regionMatches(int startIndex1, String string2, int startIndex2, int count)

public boolean regionMatches(boolean ignoreCase, int startIndex1, String string2, int startIndex2, int count)

These methods perform a case-sensitive or -insensitive comparison of two substrings.

public String replace(char oldChar, char newChar)

The `replace` method returns a *new* `String` that is the result of replacing all occurrences of `oldChar` by `newChar`. The original string is not modified.

public boolean startsWith(String prefix)

public boolean startsWith(String prefix, int startIndex)

These methods check for string prefixes.

public String substring(int startIndex, int endIndex)

public String substring(int startIndex)

These methods return substrings in the specified range. If no ending index is supplied, the substring goes to the end of the original string.

public char[ ] toCharArray()

Use this method to generate a character array.

public String toLowerCase()

public String toLowerCase(Locale locale)

public String toUpperCase()

public String toUpperCase(Locale locale)

These methods convert the entire string to lower case or upper case, optionally using the rules of the specified locale.

public String trim()

This method returns a *new* `String` with leading and trailing white space and control characters removed. The original `String` is not modified.

public static String valueOf(boolean b)

public static String valueOf(char c)

public static String valueOf(char[ ] data)

public static String valueOf(char[ ] data, int startIndex, int count)

public static String valueOf(double d)

public static String valueOf(float f)

public static String valueOf(int i)

public static String valueOf(long l)

These static methods convert the specified primitive values to strings.

public static String valueOf(Object o)

This static method uses the object's `toString` method to generate a string.

Listing 8.16 is an example that demonstrates the use of many `String` methods. The results are shown in Listing 8.17.

Listing 8.16 `StringTest.java`

```
public class StringTest {
  public static void main (String[] args) {
    String str = "";
    if (args.length > 0) {
        str = args[0];
    }
    if (str.length()>8) {
      System.out.println("String is \"" + str + "\"\n");
      System.out.println("  charAt(3) ------------------ " +
                         str.charAt(3));
      System.out.println("  compareTo(Moscow) ---------- " +
                         str.compareTo("Moscow"));
      System.out.println("  concat(SuFFiX) ------------- " +
                         str.concat("SuFFiX"));
      System.out.println("  endsWith(hic) -------------- " +
                         str.endsWith("hic"));
      System.out.println("  == Geographic -------------- " +
                         (str == "Geographic"));
      System.out.println("  equals(geographic) --------- " +
                         str.equals("geographic"));
      System.out.println("  equalsIgnoreCase(geographic) " +
                         str.equalsIgnoreCase("geographic"));
      System.out.println("  indexOf('o') --------------- " +
                         str.indexOf('o'));
      System.out.println("  indexOf('i',5) ------------- " +
                         str.indexOf('i',5));
      System.out.println("  indexOf('o',5) ------------- " +
                         str.indexOf('o',5));
```

```
      System.out.println("  indexOf(rap) --------------- " +
                         str.indexOf("rap"));
      System.out.println("  indexOf(rap, 5) ------------ " +
                         str.indexOf("rap", 5));
      System.out.println("  lastIndexOf('o') ----------- " +
                         str.lastIndexOf('o'));
      System.out.println("  lastIndexOf('i',5) --------- " +
                         str.lastIndexOf('i',5));
      System.out.println("  lastIndexOf('o',5) --------- " +
                         str.lastIndexOf('o',5));
      System.out.println("  lastIndexOf(rap) ----------- " +
                         str.lastIndexOf("rap"));
      System.out.println("  lastIndexOf(rap, 5) -------- " +
                         str.lastIndexOf("rap", 5));
      System.out.println("  length() ------------------- " +
                         str.length());
      System.out.println("  replace('c','k') ----------- " +
                         str.replace('c','k'));
      System.out.println("  startsWith(eog,1) ---------- " +
                         str.startsWith("eog",1));
      System.out.println("  startsWith(eog) ------------ " +
                         str.startsWith("eog"));
      System.out.println("  substring(3) --------------- " +
                         str.substring(3));
      System.out.println("  substring(3,8) ------------- " +
                         str.substring(3,8));
      System.out.println("  toLowerCase() -------------- " +
                         str.toLowerCase());
      System.out.println("  toUpperCase() -------------- " +
                         str.toUpperCase());
      System.out.println("  trim() --------------------- " +
                         str.trim());
      System.out.println("\nString is still \"" + str + "\"\n");
    }
  }
}
```

Listing 8.17 `StringTest` Output

```
Prompt> java StringTest Geographic
String is "Geographic"
charAt(3) ------------------ g
compareTo(Moscow) ---------- -6
concat(SuFFiX) ------------- GeographicSuFFiX
endsWith(hic) -------------- true
== Geographic -------------- false
equals(geographic) --------- false
equalsIgnoreCase(geographic) true
indexOf('o') --------------- 2
indexOf('i',5) ------------- 8
indexOf('o',5) ------------- -1
indexOf(rap) --------------- 4
```

```
  indexOf(rap, 5) ------------ -1
  lastIndexOf('o') ----------- 2
  lastIndexOf('i',5) --------- -1
  lastIndexOf('o',5) --------- 2
  lastIndexOf(rap) ----------- 4
  lastIndexOf(rap, 5) -------- 4
  length() ------------------- 10
  replace('c','k') ----------- Geographik
  startsWith(eog,1) ---------- true
  startsWith(eog) ------------ false
  substring(3) --------------- graphic
  substring(3,8) ------------- graph
  toLowerCase() -------------- geographic
  toUpperCase() -------------- GEOGRAPHIC
  trim() --------------------- Geographic

String is still "Geographic"
```

Constructors

public String()

This constructor builds a zero-length but non-`null` string.

public String(byte[ ] bytes)

public String(byte[ ] bytes, String encoding)

public String(byte[ ] bytes, int startIndex, int count)

public String(byte[ ] bytes, int startIndex, int count, String encoding)

These constructors build a string from byte arrays.

public String(char[ ] chars)

public String(char[ ] chars, int startIndex, int count)

These constructors build a string from character arrays.

public String(String string)

This constructor copies the string. The result is `equals` but not `==` to the input.

public String(StringBuffer stringBuffer)

This constructor converts a `StringBuffer` to a `String`.

8.9 Arrays

An array is a simple and efficient data structure used in virtually all programming languages. Arrays are used principally to provide constant-time access to a fixed-size collection of primitive datatypes or objects and are a way of referring to many distinct values by a single identifier. Arrays are implemented as real objects with the following properties:

- Their length can be determined through the `length` field.
- They can be assigned to variables of type `Object` as well as to variables of their specific type.
- Arrays are efficiently passed to methods by reference just like any other `Object`.

Array indexing is zero-based, so elements in an array of 10 values can be referred to with subscripts from 0 through 9. Arrays are normally created in two steps: allocation and assignment of values. For an array of primitive datatypes, each element of the array is initialized to the default value for the primitive datatype. For an array of objects, each element of the array is initialized to `null`. When accessing array elements, you should be careful to avoid referring to an array location larger than the size of the array.

Following we present two approaches for allocating an array. The first approach allocates an array in two steps, first declaring the size of the array and then assigning values. The second approach allocates the array in one stop, assigning the values at the same time the array is declared.

Two-Step Array Allocation

In the first step, an array of the proper size and desired type is allocated:

```
int[] values = new int[2]; // a 2-element array
Point[] points = new Point[5]; // a 5-element array
int numNames = askHowManyNames(); // Set at runtime
String[] names = new String[numNames];
```

This step does not build any of the objects that actually go into the array. That building is done in a separate step by means of the `arrayReference[index]` notation to access array locations. For example:

```
values[0] = 10;
values[1] = 100;
for(int i=0; i<points.length; i++) {
  points[i] = new Point(i*2, i*4);
}
for(int j=0; j<names.length; j++) {
  names[j] = "Name " + j;
}
```

A common error is to forget the second step. If you get a `NullPointerException` whenever you access an array element, check for this problem first.

Core Warning

*The following allocates n **references** to `SomeObject`; it doesn't actually build any **instances** of `SomeObject`:*

```
SomeObject[] objArray = new SomeObject[n];
```

By the way, you are allowed to assign values in an array over time; that is, the array does not have to be completely initialized immediately after it is created, though it is considered good form to do so. Also, if you are a die-hard C hacker, you are permitted to declare arrays with the syntax

```
Type someVar[] = ...
```

instead of

```
Type[] someVar = ...
```

Just be aware that this reduces your JHF (Java Hipness Factor) by 2.5 units.

One-Step Array Allocation

You can also allocate arrays and assign to them in one fell swoop by specifying comma-separated elements inside curly braces in the initialization portion of a variable declaration. For instance:

```
int[] values = { 10, 100 };
Point[] points = { new Point(0, 0),
                   new Point(2, 4),
                   new Point(4, 8),
                   ... };
```

The Java Virtual Machine will count the number of elements you are placing in the array initially and determine the length of the array accordingly. Listing 8.18 is an example that uses one-step initialization of arrays and passes arrays to other methods. Listing 8.19 shows the result of `Golf.java`.

Listing 8.18 `Golf.java`

```
/** Report on a round of golf at St. Andy's. */

public class Golf {
  public static void main(String[] args) {
    int[] pars   = { 4,5,3,4,5,4,4,3,4 };
    int[] scores = { 5,6,3,4,5,3,2,4,3 };
    report(pars, scores);
  }

  /** Reports on a short round of golf. */
  public static void report(int[] pars, int[] scores) {
    for(int i=0; i<scores.length; i++) {
      int hole = i+1;
      int difference = scores[i] - pars[i];
      System.out.println("Hole " + hole + ": " +
                         diffToString(difference));
    }
  }

  /** Convert to English. */

  public static String diffToString(int diff) {
    String[] names = {"Eagle", "Birdie", "Par", "Bogey",
                      "Double Bogey", "Triple Bogey", "Bad"};
    // If diff is -2, return names[0], or "Eagle".
    int offset = 2;
    return(names[offset + diff]);
  }
}
```

Listing 8.19 `Golf` Output

```
Prompt> java Golf
Hole 1: Bogey
Hole 2: Bogey
Hole 3: Par
Hole 4: Par
Hole 5: Par
Hole 6: Birdie
Hole 7: Eagle
Hole 8: Bogey
Hole 9: Birdie
```

Multidimensional Arrays

In the Java programming language, multidimensional arrays are implemented by arrays-of-arrays, just as in C and C++. For instance, the following allocates and fills a 12 x 14 array.

```
int[][] values = new int[12][14];
for(int i=0; i<12; i++) {
  for(int j=0; j<14; j++) {
    values[i][j] = someFunctionOf(i, j);
  }
}
```

You can access individual elements as follows:

```
int someVal = values[i][j]; // i<12, j<14
values[i][j] = someInt;
```

You can also access entire rows by omitting the second subscript:

```
int[] someArray = values[i]; // 0<=i<=11
values[i] = someOtherArray;
```

You can generalize this process to dimensions higher than 2. Also, the internal arrays need not be of the same length. To implement nonrectangular arrays, omit the rightmost size declarations. Here is an example:

```
String[][] names = new String[3][];
String[] name0 = { "John", "Q.", "Public" };
String[] name1 = { "Jane", "Doe" };
String[] name2 = { "Pele" };
names[0] = name0; // 3 elements
names[1] = name1; // 2 elements
names[2] = name2; // 1 element
```

The "shorthand" array declaration is legal with multidimensional arrays also:

```
String[][] altNames = { { "John", "Q.", "Public" },
                        { "Jane", "Doe" },
                        { "Pele" }
                      };
```

8.10 Vectors

Arrays are extremely useful, but they are limited by the fact that they cannot grow or change in size over time. To address this limitation, Java provides a "stretchable" array class: `java.util.Vector`. It is used for many of the same purposes as linked lists because you can insert or remove elements at any location. Arrays are used for the underlying implementation of `Vector`. Therefore, it only takes constant time to access a specified location, but takes time proportional to the number of elements contained to insert elements at the beginning or in the middle of the `Vector`.

Following is a summary of the `Vector` methods; see Section 8.11 for an example. Notice that the insertion and retrieval methods return elements of type `Object`. This choice makes it difficult, but not impossible, to make vectors that can hold objects only of a particular type and that let you retrieve values without typecasting the return value.

Constructors

public Vector()

public Vector(int initialCapacity)

public Vector(int initialCapacity, int capacityIncrement)

These constructors build an empty `Vector`. The initial capacity (size of the underlying array) is 10 if not specified, but Java automatically copies the data into a bigger array if more elements are added than the current vector size allows.

Methods

public void addElement(Object object)

public void insertElementAt(Object object, int index)

public void setElementAt(Object object, int index)

These synchronized methods add elements to the `Vector`. The `addElement` method inserts at the end; the other two methods use the location specified. With `insertElementAt`, the objects at and to the right of the specified location are shifted one location to the right. With `setElementAt`, the object at the specified location is replaced.

public int capacity()

This method returns the size of the underlying array, that is, the number of elements the `Vector` can hold before it will be resized.

public boolean contains(Object object)

The `contains` method determines whether the `Vector` contains an object that `equals` the one specified.

public void copyInto(Object[ ] newArray)

This synchronized method copies the object references into the specified array, in order.

public Object elementAt(int index)

The synchronized `elementAt` method returns the element at the specified location.

public Enumeration elements()

The `java.util.Enumeration` class defines an interface used by several enumerable classes. You can use `elements` to get an `Enumeration` object corresponding to the `Vector`.

public void ensureCapacity(int minimum)

This synchronized method guarantees that the underlying array has at least the specified number of elements.

public Object firstElement()

public Object lastElement()

These synchronized methods return the first and last entry in the `Vector`, respectively.

public int indexOf(Object object)

public int indexOf(Object object, int startIndex)

public int lastIndexOf(Object object)

public int lastIndexOf(Object object, int startIndex)

These synchronized methods return the leftmost or rightmost index of the element that `equals` the object specified.

public boolean isEmpty()

This method returns `false` if the `Vector` has any elements; `true` otherwise.

public boolean removeElement(Object object)

public void removeElementAt(int index)

public void removeAllElements()

These synchronized methods let you remove entries from the `Vector`.

public void setSize(int newSize)

This synchronized method sets a specific size for the `Vector`. It differs from `ensureCapacity` in that it will truncate the `Vector` if the `Vector` is larger than the specified size.

public int size()

The `size` method returns the number of elements in the `Vector` (not the size of the underlying array, which might be larger).

public void trimToSize()

This synchronized method sets the underlying array to be exactly the same size as the current number of elements. You should avoid this method while elements are being

added and removed, but it might save memory if the method is used once the `Vector` elements are fixed.

Many of the `Vector` methods are synchronized to prevent potential race conditionals as a result of multiple threads accessing the same data. Synchronization does incur a performance hit. Therefore, Java 2 added two unsynchronized classes, similar in function to `Vector` class: `ArrayList` and `LinkedList`. However, these two new classes are only available in JDK 1.2 and later and are not available in most browsers supporting applets. For additional information on synchronization and multithreaded programs, see Chapter 16 (Concurrent Programming with Java Threads).

8.11 Example: A Simple Binary Tree

This section shows how references and the `Vector` class can be used to create a binary tree class. This data structure includes a `depthFirstSearch` method, which traverses the tree in depth-first order (staying to the left and going as deep as possible until having to backtrack). Notice that this method is recursive; recursion is natural for depth-first search. The `depthFirstSearch` method also uses the `NodeOperator` interface (see Listing 8.21) to generalize the operation that will be performed on each node. This interface lets you change what to do with a tree without modifying the `Node` class, which is shown in Listing 8.20. `Leaf` nodes are implemented as a subclass of `Node`, see Listing 8.22. The data structure also includes a `breadthFirstSearch` method that uses a `Vector` to build a queue that traverses the tree in breadth-first order (visiting all nodes on a given level before moving on to the next).

Note that many data structures like this have been added as a core part of the language in Java 2. For details, see http://java.sun.com/j2se/1.3/docs/guide/collections/. The Java Collections Framework comes with an extensible API for building, manipulating, and iterating over data structures. Implementations of hash tables, growing arrays, doubly linked lists, and balanced binary trees are provided. Although existing classes like `Vector` and `Hashtable` provide some of the same functionality, the older classes are explicitly synchronized and therefore possibly slower in certain situations. Collections also provides standard methods for sorting, filling arrays, and binary search that were missing in earlier Java releases.

Listing 8.20 `Node.java`

```
import java.util.Vector;

/** A data structure representing a node in a binary tree.
 *  It contains a node value and a reference (pointer) to
 *  the left and right subtrees.
 */

public class Node {
  private Object nodeValue;
  private Node leftChild, rightChild;

 /** Build Node with specified value and subtrees. */

  public Node(Object nodeValue, Node leftChild,
              Node rightChild) {
    this.nodeValue = nodeValue;
    this.leftChild = leftChild;
    this.rightChild = rightChild;
  }
```

```
  /** Build Node with specified value and L subtree. R child
   *  will be null. If you want both children to be null, use
   *  the Leaf constructor.
   */

  public Node(Object nodeValue, Node leftChild) {
    this(nodeValue, leftChild, null);
  }

  /** Return the value of this node. */

  public Object getNodeValue() {
    return(nodeValue);
  }

  /** Specify the value of this node. */

  public void setNodeValue(Object nodeValue) {
    this.nodeValue = nodeValue;
  }

 /** Return the L subtree. */

  public Node getLeftChild() {
    return(leftChild);
  }

  /** Specify the L subtree. */

  public void setLeftChild(Node leftChild) {
    this.leftChild = leftChild;
  }

  /** Return the R subtree. */

  public Node getRightChild() {
    return(rightChild);
  }

  /** Specify the R subtree. */

  public void setRightChild(Node rightChild) {
    this.rightChild = rightChild;
  }

  /** Traverse the tree in depth-first order, applying
   *  the specified operation to each node along the way.
   */

  public void depthFirstSearch(NodeOperator op) {
    op.operateOn(this);
    if (leftChild != null) {
```

```
      leftChild.depthFirstSearch(op);
    }
    if (rightChild != null) {
      rightChild.depthFirstSearch(op);
    }
  }

  /** Traverse the tree in breadth-first order, applying the
   *  specified operation to each node along the way.
   */

  public void breadthFirstSearch(NodeOperator op) {
    Vector nodeQueue = new Vector();
    nodeQueue.addElement(this);
    Node node;
    while(!nodeQueue.isEmpty()) {
      node = (Node)nodeQueue.elementAt(0);
      nodeQueue.removeElementAt(0);
      op.operateOn(node);
      if (node.getLeftChild() != null) {
        nodeQueue.addElement(node.getLeftChild());
      }
      if (node.getRightChild() != null) {
        nodeQueue.addElement(node.getRightChild());
      }
    }
  }
}
```

Listing 8.21 `NodeOperator.java`

```
/** An interface used in the Node class to ensure that
 *  an object has an operateOn method.
 */

public interface NodeOperator {
  void operateOn(Node node);
}
```

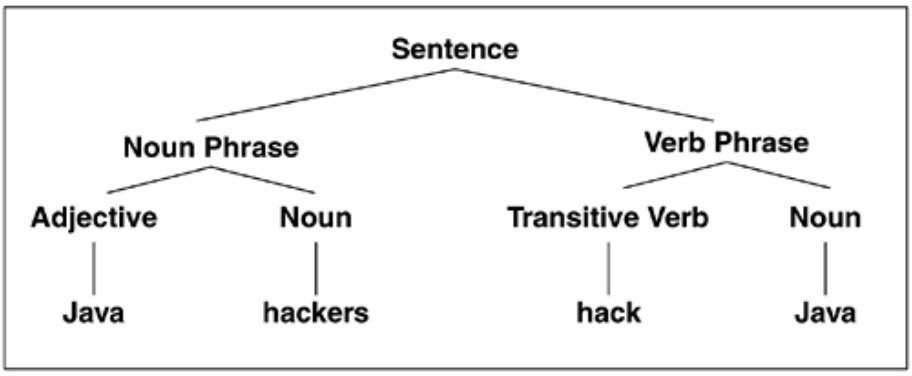
Listing 8.22 `Leaf.java`

```
/** Leaf node: a node with no subtrees. */

public class Leaf extends Node {
  public Leaf(Object value) {
    super(value, null, null);
  }
}
```

Now that we have a general data structure, let's build a specific test case. Figure 8-1 shows a simple tree; Listing 8.23 represents the tree by using the `Node` and `Leaf` classes just shown and makes a `NodeOperator` that does nothing but print the value of each node it visits. Listing 8.24 shows the results.

Figure 8-1. Parse tree for "Java hackers hack Java."



Listing 8.23 `TreeTest.java`

```
/** A NodeOperator that prints each node. */

class PrintOperator implements NodeOperator {
  public void operateOn(Node node) {
    System.out.println(node.getNodeValue());
  }
}

/** A sample tree representing a parse tree of
 *  the sentence "Java hackers hack Java", using
 *  some simple context-free grammar.
 */

public class TreeTest {
  public static void main(String[] args) {
    Node adjective =
      new Node("  Adjective", new Leaf("   Java"));
    Node noun1 =
      new Node("  Noun", new Leaf("   hackers"));
    Node verb =
      new Node("  TransitiveVerb", new Leaf("   hack"));
    Node noun2 =
      new Node("  Noun", new Leaf("   Java"));
    Node np = new Node(" NounPhrase", adjective, noun1);
    Node vp = new Node(" VerbPhrase", verb, noun2);
    Node sentence = new Node("Sentence", np, vp);
    PrintOperator printOp = new PrintOperator();
    System.out.println("Depth first traversal:");
    sentence.depthFirstSearch(printOp);
    System.out.println("\nBreadth first traversal:");
    sentence.breadthFirstSearch(printOp);
  }
}
```

Listing 8.24 `TreeTest` Output

```
Prompt> java TreeTest
```

```
Depth first traversal:
Sentence
 NounPhrase
  Adjective
   Java
  Noun
   hackers
 VerbPhrase
  TransitiveVerb
   hack
  Noun
   Java

Breadth first traversal:
Sentence
 NounPhrase
 VerbPhrase
  Adjective
  Noun
  TransitiveVerb
  Noun
   Java
   hackers
   hack
   Java
```

8.12 Exceptions

The Java programming language has a very nice error-handling system: *exceptions.* Exceptions can be "thrown" (generated) in one block of code and "caught" (handled) in an outer block or in a method that called the current one. Java exceptions differ from C++ exceptions in two major ways. The first major difference is that the exception-handling construct (`try`/`catch`) has a `finally` clause that always gets executed, regardless of whether or not an exception was thrown. The second significant difference is that you can *require* users of your methods to handle exceptions your methods generate; if they fail to do that, their code will not compile.

Basic Form

The simplest form of exception handling is a block of the following form:

```
try {
  statement1;
  statement2;
  ...
} catch(SomeException someVar) {
  handleTheException(someVar);
}
```

For example, the constructor for `java.net.URL` potentially generates a `java.net.MalformedURLException`, and the `readLine` method of `java.io.BufferedReader` potentially generates a `java.io.IOException`. Listing 8.25 uses both the `URL` constructor and the `readLine` method to read a `URL` from the user and print descriptive information about it, so it needs to catch both possible exceptions. As Listing 8.26 shows,

`MalformedURLException` verifies that the `URL` is in legal format; it doesn't retrieve the referenced file or even check that it exists. You can see plenty of examples of reading the contents of such files in Chapter 17 (Network Programming).

Listing 8.25 `URLTest.java`

```
import java.net.*; // For URL, MalformedURLException
import java.io.*;  // For BufferedReader

/** A small class to demonstrate try/catch blocks. */

public class URLTest {
  public static void main(String[] args) {
    URLTest test = new URLTest();
    test.getURL();
    test.printURL();
  }

  private URL url = null;

  /** Read a string from user and create a URL from it. If
   *  reading fails, give up and report error. If reading
   *  succeeds but URL is illegal, try again.
   */

  public URL getURL() {
    if (url != null) {
      return(url);
    }
    System.out.print("Enter URL: ");
    System.out.flush();
    BufferedReader in = new BufferedReader(
                          new InputStreamReader(System.in));
    String urlString;
    try {
      urlString = in.readLine();
    } catch(IOException ioe) {
      System.out.println("IOError when reading input: " + ioe);
      ioe.printStackTrace(); // Show stack dump.
      return(null);
    }
    try {
      url = new URL(urlString);
    } catch(MalformedURLException mue) {
      System.out.println(urlString + " is not valid.\n" +
                         "Try again.");
      getURL();
    }
    return(url);
  }

  /** Print info on URL. */
```

```
  public void printURL() {
    if (url == null) {
      System.out.println("No URL.");
    } else {
      String protocol = url.getProtocol();
      String host = url.getHost();
      int port = url.getPort();
      if (protocol.equals("http") && (port == -1)) {
        port = 80;
      }
      String file = url.getFile();
      System.out.println("Protocol: " + protocol +
                         "\nHost: " + host +
                         "\nPort: " + port +
                         "\nFile: " + file);
    }
  }
}
```

Listing 8.26 `URLTest` Output

```
> java URLTest
> Enter URL: http://java.sun.com/ConvertingToActiveX.html
Protocol: http
Host: java.sun.com
Port: 80
File: /ConvertingToActiveX.html
```

Note the use of the `printStackTrace` method in `getURL`. This shows the method call stack at the point the exception occurred. In many implementations, it even includes line numbers in the source files. This is such a useful debugging tool that it is sometimes used even when no exceptions are generated. For instance, the following simply prints a stack dump:

```
new Throwable().printStackTrace();
```

Multiple Catch Clauses

A single `try` can have more than one `catch`. If an exception is generated, Java executes the first `catch` clause that matches the type of exception thrown. Since exceptions can be created hierarchically like other Java classes, you must catch a more specific exception before a more general one. For instance, although the `getURL` method could be simplified to use a single `try` block with two `catch` clauses (Listing 8.27), the order of the `catch` clauses needs to be reversed since a `MalformedURLException` is actually a subclass of `IOException`.

Core Approach

If you have multiple `catch` clauses, you must order them from the most specific to the most general.

Listing 8.27 Simplified `getURL` Method

```
public URL getURL() {
```

```
  if (url != null) {
    return(url);
  }
  System.out.print("Enter URL: ");
  System.out.flush();
  BufferedReader in = new BufferedReader(
                        new InputStreamReader(System.in));
  String urlString = null;
  try {
    urlString = in.readLine();
    url = new URL(urlString);
  } catch(MalformedURLException mue) {
    System.out.println(urlString + " is not valid.\n" +
                       "Try again.");
    getURL();
  } catch(IOException ioe) {
    System.out.println("IOError when reading input: " + ioe);
    ioe.printStackTrace(); // Show stack dump
    return(null);
  }
  return(url);
}
```

The Finally Clause

After the last `catch` clause, you are permitted a `finally` clause that *always* gets executed, regardless of whether or not exceptions are thrown. It is executed even if `break`, `continue`, or `return` is used within the `try` or `catch` clauses. Listing 8.28 shows a third version of `getURL` that uses this approach.

Listing 8.28 Further Simplified `getURL` Method

```
public URL getURL() {
  if (url != null) {
    return(url);
  }
  System.out.print("Enter URL: ");
  System.out.flush();
  BufferedReader in = new BufferedReader(
                        new InputStreamReader(System.in));
  String urlString = null;
  try {
    urlString = in.readLine();
    url = new URL(urlString);
  } catch(MalformedURLException mue) {
    System.out.println(urlString + " is not valid.\n" +
                       "Try again.");
    getURL();
  } catch(IOException ioe) {
    System.out.println("IOError when reading input: " + ioe);
    ioe.printStackTrace(); // Can skip return(null) now
  } finally {
    return(url);
```

```
  }
}
```

Thrown Exceptions

If you write a method that potentially generates one or more exceptions and you don't handle them explicitly, you need to declare them with the `throws` construct, as follows:

```
public SomeType someMethod(...) throws SomeException {
```

or

```
public SomeType someMethod(...)
       throws ExceptionType1, ExceptionType2 {
```

This declaration lets you do two things. First, it permits you to write methods that have enforced safety checking; users are required to handle the exception when calling the methods. Second, it permits you to postpone exception handling to a method higher in the method call chain by declaring them in the method declaration but ignoring them in the method body.

If you want to explicitly generate an exception, use the `throw` construct as illustrated:

```
throw new IOException("Blocked by firewall");
throw new MalformedURLException(
            "Invalid protocol: telephone");
```

Using `throw` is more common with exceptions you define yourself. You can make your own exception classes by subclassing any of the existing exception types. Listing 8.29 gives an example of an exception type you might use when creating geometric objects that require nonnegative widths, heights, radii, and so forth. The class is demonstrated in Listing 8.30.

Listing 8.29 `NegativeLengthException.java`

```
import java.io.*;

public class NegativeLengthException extends Exception {

  /** Test NegativeLengthException */

  public static void main(String[] args) {
    try {
      int lineLength = readLength();
      for(int i=0; i<lineLength; i++) {
        System.out.print("*");
      }
      System.out.println();
    } catch (NegativeLengthException nle) {
      System.out.println("NegativeLengthException: " +
                         nle.getMessage());
    }
  }

  public NegativeLengthException() {
    super("Negative dimensions not permitted.");
```

```
  }

  public NegativeLengthException(String message) {
    super(message);
  }
  // readLength catches IOExceptions locally but lets the
  // calling method handle NegativeLengthExceptions.
  private static int readLength() throws NegativeLengthException {
    BufferedReader in = new BufferedReader(
                          new InputStreamReader(System.in));
    System.out.print("Enter length: ");
    System.out.flush();
    int len = 0;
    try {
      String line = in.readLine();
      len = Integer.parseInt(line);
      if (len < 0) {
        throw new NegativeLengthException();
      }
    } catch (IOException ioe) {
      System.out.println("Problem reading from keyboard");
    }
    return(len);
  }
}
```

Listing 8.30 Throwing `NegativeLengthExceptions`

```
> java NegativeLengthException
> Enter length: 4
****

> java NegativeLengthException
> Enter length: -345
NegativeLengthException: Negative dimensions not permitted.
```

Unchecked Exceptions

The exceptions discussed so far have been *checked exceptions;* exceptions that you are required to handle. Java also includes two classes of *unchecked exceptions:* `Error` and `RuntimeException`. You are permitted to handle these exceptions but are not required to, since the number of places from which they could be generated is too large. For instance, members of the `Error` class include `OutOfMemoryError` (e.g., array allocation or call to `new` failed because of insufficient memory) and `ClassFormatError` (e.g., a `.class` file in illegal format, perhaps because text mode instead of binary mode was used when the file was FTP'd).

The `RuntimeException` subclasses include `ArithmeticException` (e.g., an integer division-by-zero) and `ArrayIndexOutOfBoundsException` (e.g., you forgot to bound a loop at the array's length). A catch-all `RuntimeException` often extended in user code is `IllegalArgumentException`. This exception lets you create hooks in your code for the handling of unusual cases without requiring users to catch the exceptions.

8.13 Summary

This chapter briefly reviewed the fundamental syntax of Java programs: primitive and reference types, operators, the `Math` class, strings, arrays, vectors, and exceptions. Once you have assimilated this and the object-oriented programming coverage of Chapter 7, you are ready to focus on the more specific Java topics covered in the next chapters.

 CONTENTS