

This chapter explores the key elements that we use in a program: objects and primitive data. We develop the ability to create

and use objects for the services they provide. This ability is fundamental to the process of writing any program in an object-oriented language such as Java. We use objects to manipulate character strings, obtain information from the user, perform complex calculations, and format output. In the Graphics Track of this chapter, we explore the relationship between Java and the Web, and delve into Java's abilities to manipulate color and draw shapes.

### chapter objectives

- ▶ Establish the difference between primitive data and objects.
- ▶ Declare and use variables.
- ▶ Perform mathematical computations.
- ▶ Create objects and use them for the services they provide.
- ▶ Explore the difference between a Java application and a Java applet.
- ▶ Create graphical programs that draw shapes.

## 2.0 an introduction to objects

As we stated in Chapter 1, Java is an object-oriented language. As the name implies, an *object* is a fundamental entity in a Java program. This book is centered on the idea of developing software by defining objects with which we can interact and that interact with each other.

In addition to objects, a Java program also manages primitive data. *Primitive data* include common, fundamental values such as numbers and characters. An object usually represents something more specialized or complex, such as a bank account. An object often contains primitive values and is in part defined by them. For example, an object that represents a bank account might contain the account balance, which is stored as a primitive numeric value.

### key concept

The information we manage in a Java program is either represented as primitive data or as objects.

A *data type* defines a set of values and the operations that can be performed on those values. We perform operations on primitive types using *operators* that are built into the programming language. For example, the addition operator `+` is used to add two numbers together. We discuss Java's primitive data types and their operators later in this chapter.

An object is defined by a *class*, which can be thought of as the data type of the object. The operations that can be performed on the object are defined by the methods in the class. As we discussed in Chapter 1, a method is a collection of programming statements that is given a specific name so that we can invoke the method as needed.

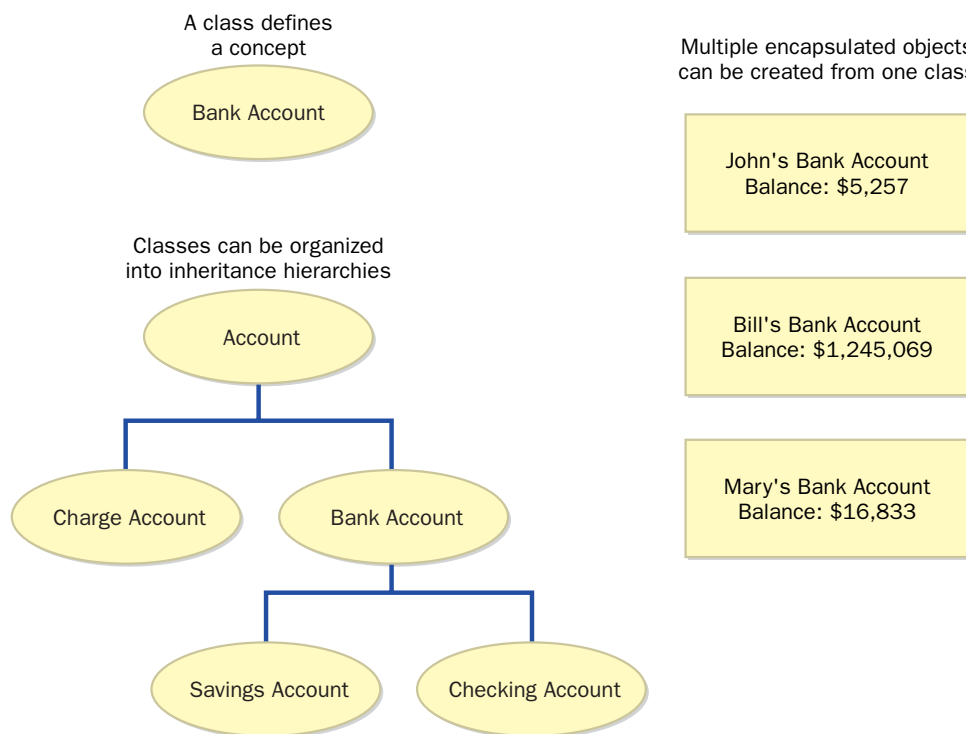
Once a class has been defined, multiple objects can be created from that class. For example, once we define a class to represent the concept of a bank account, we can create multiple objects that represent specific, individual bank accounts. Each bank account object would keep track of its own balance. This is an example of *encapsulation*, meaning that each object protects and manages its own information. The methods defined in the bank account class would allow us to perform operations on individual bank account objects. For instance, we might withdraw money from a particular account. We can think of these operations as services that the object performs. The act of invoking a method on an object sometimes is referred to as sending a *message* to the object, requesting that the service be performed.

Classes can be created from other classes using *inheritance*. That is, the definition of one class can be based on another class that already exists. Inheritance

is a form of software *reuse*, capitalizing on the similarities between various kinds of classes that we may want to create. One class can be used to derive several new classes. Derived classes can then be used to derive even more classes. This creates a hierarchy of classes, where characteristics defined in one class are inherited by its children, which in turn pass them on to their children, and so on. For example, we might create a hierarchy of classes that represent various types of accounts. Common characteristics are defined in high-level classes, and specific differences are defined in derived classes.

Classes, objects, encapsulation, and inheritance are the primary ideas that make up the world of object-oriented software. They are depicted in Fig. 2.1.

This chapter focuses on how to use objects and primitive data. In Chapter 4, we explore how to define our own objects by writing our own classes and methods. In Chapter 7, we explore inheritance.



**figure 2.1** Various aspects of object-oriented software

## 2.1 using objects

In the `Lincoln` program in Chapter 1, we invoked a method through an object as follows:

```
System.out.println ("Whatever you are, be a good one.");
```

The `System.out` object represents an output device or file, which by default is the monitor screen. To be more precise, the object's name is `out` and it is stored in the `System` class. We explore that relationship in more detail at the appropriate point in the text.

The `println` method represents a service that the `System.out` object performs for us. Whenever we request it, the object will print a string of characters to the screen. We can say that we send the `println` message to the `System.out` object to request that some text be printed.

Each piece of data that we send to a method is called a *parameter*. In this case, the `println` method takes only one parameter: the string of characters to be printed.

The `System.out` object also provides another service we can use: the `print` method. Let's look at both of these services in more detail.

### the `print` and `println` methods

The difference between `print` and `println` is small but important. The `println` method prints the information sent to it, then moves to the beginning of the next line. The `print` method is similar to `println`, but does not advance to the next line when completed.

The program shown in Listing 2.1 is called `Countdown`, and it invokes both the `print` and `println` methods.

Carefully compare the output of the `Countdown` program to the program code. Note that the word `Liftoff` is printed on the same line as the first few words, even though it is printed using the `println` method. Remember that the `println` method moves to the beginning of the next line *after* the information passed to it is printed.

Often it is helpful to use graphics to show objects and their interaction. Figure 2.2 shows part of the situation that occurs in the `Countdown` program. The `Countdown` class, with its `main` method, is shown invoking the `println` method of the `System.out` object.

**listing**  
**2.1**

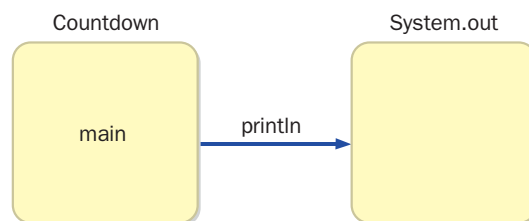
```
//*****
// Countdown.java      Author: Lewis/Loftus
//
// Demonstrates the difference between print and println.
//*****

public class Countdown
{
    //-----
    // Prints two lines of output representing a rocket countdown.
    //-----
    public static void main (String[] args)
    {
        System.out.print ("Three... ");
        System.out.print ("Two... ");
        System.out.print ("One... ");
        System.out.print ("Zero... ");

        System.out.println ("Liftoff!"); // appears on first output line
        System.out.println ("Houston, we have a problem.");
    }
}
```

**output**

```
Three . . . Two . . . One . . . Zero . . . Liftoff!
Houston, we have a problem.
```

**figure 2.2** Sending a message

We mentioned in the previous section that the act of invoking a method is referred to, in object-oriented terms, as sending a message. The diagram in Figure 2.2 supports this interpretation by showing the method name—the message—on the arrow. We could also have shown the information that makes up the rest of the message: the parameters to the methods.

As we explore objects and classes in more detail throughout this book, we will formalize the types of diagrams we use to represent various aspects of an object-oriented program. The more complex our programs get, the more helpful such diagrams become.

## abstraction

An object is an *abstraction*, meaning that the precise details of how it works are irrelevant from the point of view of the user of the object. We don't really need to know how the `println` method prints characters to the screen as long as we can count on it to do its job. Of course, it is sometimes helpful to understand such information, but it is not necessary in order to *use* the object.

Sometimes it is important to hide or ignore certain details. Humans are capable of mentally managing around seven (plus or minus two) pieces of information in short-term memory. Beyond that, we start to lose track of some of the pieces. However, if we group pieces of information together, those pieces can be managed as one “chunk” in our minds. We don't actively deal with all of the details in the chunk, but we can still manage it as a single entity. Therefore, we can deal with large quantities of information by organizing them into chunks. An object is a construct that organizes information and allows us to hide the details inside. An object is therefore a wonderful abstraction.

We use abstractions every day. Think about a car for a moment. You don't necessarily need to know how a four-cycle combustion engine works in order to drive a car. You just need to know some basic operations: how to turn it on, how to put it in gear, how to make it move with the pedals and steering wheel, and how to stop it. These operations define the way a person interacts with the car. They mask the details of what is happening inside the car that allow it to function. When you're driving a car, you're not usually thinking about the spark plugs igniting the gasoline that drives the piston that turns the crankshaft that turns the axle that turns the wheels. If you had to worry about all of these underlying details, you'd probably never be able to operate something as complicated as a car.

Initially, all cars had manual transmissions. The driver had to understand and deal with the details of changing gears with the stick shift. Eventually, automatic transmissions were developed, and the driver no longer had to worry about shifting gears. Those details were hidden by raising the *level of abstraction*.

An abstraction hides details. A good abstraction hides the right details at the right time so that we can manage complexity.

key  
concept

Of course, someone has to deal with the details. The car manufacturer has to know the details in order to design and build the car in the first place. A car mechanic relies on the fact that most people don't have the expertise or tools necessary to fix a car when it breaks.

Thus, the level of abstraction must be appropriate for each situation. Some people prefer to drive a manual transmission car. A race car driver, for instance, needs to control the shifting manually for optimum performance.

Likewise, someone has to create the code for the objects we use. Soon we will define our own objects by defining classes and their methods. For now, we can make use of objects that have been defined for us already. Abstraction makes that possible.

## 2.2 string literals

A character string is an object in Java, defined by the class `String`. Because strings are so fundamental to computer programming, Java provides the ability to use a *string literal*, delimited by double quotation characters, as we've seen in previous examples. We explore the `String` class and its methods in more detail later in this chapter. For now, let's explore two other useful details about strings: concatenation and escape sequences.

### string concatenation

The program called `Facts` shown in Listing 2.2 contains several `println` statements. The first one prints a sentence that is somewhat long and will not fit on one line of the program. A character string, delimited by the double quotation character, cannot be split between two lines of code. One way to get around this problem is to use the *string concatenation* operator, the plus sign (+). String concatenation produces one string in which the second string is appended to the first. The string concatenation operation in the first `println` statement results in one large string that is passed to the method and printed.

**listing**  
**2.2**

```

/*****
//  Facts.java          Author: Lewis/Loftus
//
//  Demonstrates the use of the string concatenation operator and the
//  automatic conversion of an integer to a string.
*****/

public class Facts
{
    //-----
    //  Prints various facts.
    //-----
    public static void main (String[] args)
    {
        // Strings can be concatenated into one long string
        System.out.println ("We present the following facts for your "
                            + "extracurricular edification:");

        System.out.println ();

        // A string can contain numeric digits
        System.out.println ("Letters in the Hawaiian alphabet: 12");

        // A numeric value can be concatenated to a string
        System.out.println ("Dialing code for Antarctica: " + 672);

        System.out.println ("Year in which Leonardo da Vinci invented "
                            + "the parachute: " + 1515);

        System.out.println ("Speed of ketchup: " + 40 + " km per year");
    }
}

```

**output**

```

We present the following facts for your extracurricular edification:

Letters in the Hawaiian alphabet: 12
Dialing code for Antarctica: 672
Year in which Leonardo da Vinci invented the parachute: 1515
Speed of ketchup: 40 km per year

```

Note that we don't have to pass any information to the `println` method, as shown in the second line of the `Facts` program. This call does not print any vis-



ible characters, but it does move to the next line of output. In this case, the call to `println` passing in no parameters has the effect of printing a blank line.

The rest of the calls to `println` in the `Facts` program demonstrate another interesting thing about string concatenation: Strings can be concatenated with numbers. Note that the numbers in those lines are not enclosed in double quotes and are therefore not character strings. In these cases, the number is automatically converted to a string, and then the two strings are concatenated.

Because we are printing particular values, we simply could have included the numeric value as part of the string literal, such as:

```
"Speed of ketchup: 40 km per year"
```

Digits are characters and can be included in strings as needed. We separate them in the `Facts` program to demonstrate the ability to concatenate a string and a number. This technique will be useful in upcoming examples.

As we've mentioned, the `+` operator is also used for arithmetic addition. Therefore, what the `+` operator does depends on the types of data on which it operates. If either or both of the operands of the `+` operator are strings, then string concatenation is performed.

The `Addition` program shown in Listing 2.3 demonstrates the distinction between string concatenation and arithmetic addition. The `Addition` program uses the `+` operator four times. In the first call to `println`, both `+` operations perform string concatenation. This is because the operators execute left to right. The first operator concatenates the string with the first number (24), creating a larger string. Then that string is concatenated with the second number (45), creating an even larger string, which gets printed.

In the second call to `println`, parentheses are used to group the `+` operation with the two numbers. This forces that operation to happen first. Because both operands are numbers, the numbers are added in the arithmetic sense, producing the result 69. That number is then concatenated with the string, producing a larger string that gets printed.

We revisit this type of situation later in this chapter when we formalize the rules that define the order in which operators get evaluated.

## escape sequences

Because the double quotation character (`"`) is used in the Java language to indicate the beginning and end of a string, we must use a special technique to print the quotation character. If we simply put it in a string (`"\""`), the compiler gets

**listing**  
**2.3**

```

/*****
//  Addition.java          Author: Lewis/Loftus
//
//  Demonstrates the difference between the addition and string
//  concatenation operators.
/*****

public class Addition
{
    //-----
    //  Concatenates and adds two numbers and prints the results.
    //-----
    public static void main (String[] args)
    {
        System.out.println ("24 and 45 concatenated: " + 24 + 45);

        System.out.println ("24 and 45 added: " + (24 + 45));
    }
}

```

**output**

```

24 and 45 concatenated: 2445
24 and 45 added: 69

```

confused because it thinks the second quotation character is the end of the string and doesn't know what to do with the third one. This results in a compile-time error.

To overcome this problem, Java defines several *escape sequences* to represent special characters. An escape sequence begins with the backslash character (\), and indicates that the character or characters that follow should be interpreted in a special way. Figure 2.3 lists the Java escape sequences.

The program in Listing 2.4, called *Roses*, prints some text resembling a poem. It uses only one `println` statement to do so, despite the fact that the poem is several lines long. Note the escape sequences used throughout the string. The `\n` escape sequence forces the output to a new line, and the `\t` escape sequence represents a tab character. The `\` escape sequence ensures that the quote character is treated as part of the string, not the termination of it, which enables it to be printed as part of the output.

| Escape Sequence | Meaning         |
|-----------------|-----------------|
| \b              | backspace       |
| \t              | tab             |
| \n              | newline         |
| \r              | carriage return |
| \"              | double quote    |
| \'              | single quote    |
| \\              | backslash       |

figure 2.3 Java escape sequences

**listing**  
**2.4**

```
/** *****
//  Roses.java      Author: Lewis/Loftus
//
//  Demonstrates the use of escape sequences.
// *****

public class Roses
{
    //-----
    //  Prints a poem (of sorts) on multiple lines.
    //-----
    public static void main (String[] args)
    {
        System.out.println ("Roses are red,\n\tViolets are blue,\n" +
            "Sugar is sweet,\n\tBut I have \"commitment issues\", \n\t" +
            "So I'd rather just be friends\n\tAt this point in our " +
            "relationship.");
    }
}
```

**output**

```
Roses are red,
    Violets are blue,
Sugar is sweet,
    But I have "commitment issues",
    So I'd rather just be friends
    At this point in our relationship.
```

## 2.3 variables and assignment

Most of the information we manage in a program is represented by variables. Let's examine how we declare and use them in a program.

### variables

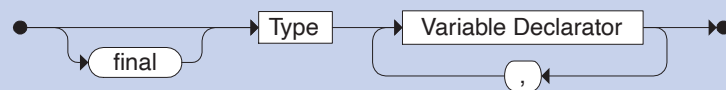
A *variable* is a name for a location in memory used to hold a data value. A variable declaration instructs the compiler to reserve a portion of main memory space large enough to hold a particular type of value and indicates the name by which we refer to that location.

key  
concept

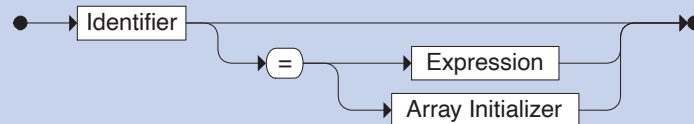
A variable is a name for a memory location used to hold a value of a particular data type.

Consider the program `PianoKeys`, shown in Listing 2.5. The first line of the `main` method is the declaration of a variable named `keys` that holds an integer (`int`) value. The declaration also gives `keys` an initial value of 88. If an initial value is not specified for a variable, the value is undefined. Most Java compilers give errors or warnings if you attempt to use a variable before you've explicitly given it a value.

#### Local Variable Declaration



#### Variable Declarator



A variable declaration consists of a `Type` followed by a list of variables. Each variable can be initialized in the declaration to the value of the specified `Expression`. If the `final` modifier precedes the declaration, the identifiers are declared as named constants whose values cannot be changed once set.

Examples:

```
int total;
double num1, num2 = 4.356, num3;
char letter = 'A', digit = '7';
final int MAX = 45;
```

**listing**  
**2.5**

```
//*****
//  PianoKeys.java      Author: Lewis/Loftus
//
//  Demonstrates the declaration, initialization, and use of an
//  integer variable.
//*****

public class PianoKeys
{
    //-----
    //  Prints the number of keys on a piano.
    //-----
    public static void main (String[] args)
    {
        int keys = 88;

        System.out.println ("A piano has " + keys + " keys.");
    }
}
```

**output**

A piano has 88 keys.

In the `PianoKeys` program, two pieces of information are provided in the call to the `println` method. The first is a string, and the second is the variable `keys`. When a variable is referenced, the value currently stored in it is used. Therefore when the call to `println` is executed, the value of `keys` is obtained. Because that value is an integer, it is automatically converted to a string so it can be concatenated with the initial string. The concatenated string is passed to `println` and printed.

Note that a variable declaration can have multiple variables of the same type declared on one line. Each variable on the line can be declared with or without an initializing value.

## the assignment statement

Let's examine a program that changes the value of a variable. Listing 2.6 shows a program called `Geometry`. This program first declares an integer variable called `sides` and initializes it to 7. It then prints out the current value of `sides`.

**Listing  
2.6**

```

//*****
//  Geometry.java      Author: Lewis/Loftus
//
//  Demonstrates the use of an assignment statement to change the
//  value stored in a variable.
//*****

public class Geometry
{
    //-----
    //  Prints the number of sides of several geometric shapes.
    //-----
    public static void main (String[] args)
    {
        int sides = 7; // declaration with initialization
        System.out.println ("A heptagon has " + sides + " sides.");

        sides = 10; // assignment statement
        System.out.println ("A decagon has " + sides + " sides.");

        sides = 12;
        System.out.println ("A dodecagon has " + sides + " sides.");
    }
}

```

**output**

```

A heptagon has 7 sides.
A decagon has 10 sides.
A dodecagon has 12 sides.

```

The next line in main changes the value stored in the variable sides:

```
sides = 10;
```

This is called an *assignment statement* because it assigns a value to a variable. When executed, the expression on the right-hand side of the assignment operator (=) is evaluated, and the result is stored in the memory location indicated by the variable on the left-hand side. In this example, the expression is simply a number, 10. We discuss expressions that are more involved than this in the next section.

### Basic Assignment



The basic assignment statement uses the assignment operator (=) to store the result of the Expression into the specified Identifier, usually a variable.

Examples:

```
total = 57;  
count = count + 1;  
value = (min / 2) * lastValue;
```

A variable can store only one value of its declared type. A new value overwrites the old one. In this case, when the value 10 is assigned to `sides`, the original value 7 is overwritten and lost forever. However, when a reference is made to a variable, such as when it is printed, the value of the variable is not changed.

A variable can store only one value of its declared type.

key  
concept

The Java language is *strongly typed*, meaning that we are not allowed to assign a value to a variable that is inconsistent with its declared type. Trying to combine incompatible types will generate an error when you attempt to compile the program. Therefore, the expression on the right-hand side of an assignment statement must evaluate to a value compatible with the type of the variable on the left-hand side.

Java is a strongly typed language. Each variable is associated with a specific type for the duration of its existence, and we cannot assign a value of one type to a variable of an incompatible type.

key  
concept

## constants

Sometimes we use data that is constant throughout a program. For instance, we might write a program that deals with a theater that can hold no more than 427 people. It is often helpful to give a constant value a name, such as `MAX_OCCUPANCY`, instead of using a literal value, such as 427, throughout the code. Literal values such as 427 are sometimes referred to as “magic” numbers because their meaning in a program is mystifying.

Constants are identifiers and are similar to variables except that they hold a particular value for the duration of their existence. In Java, if you precede a declaration with the reserved word `final`, the identifier is made a constant. By

convention, uppercase letters are used when naming constants to distinguish them from regular variables, and individual words are separated using the underscore character. For example, the constant describing the maximum occupancy of a theater could be declared as follows:

```
final int MAX_OCCUPANCY = 427;
```

key  
concept

Constants are similar to variables, but they hold a particular value for the duration of their existence.

The compiler will produce an error message if you attempt to change the value of a constant once it has been given its initial value. This is another good reason to use them. Constants prevent inadvertent coding errors because the only valid place to change their value is in the initial assignment.

There is a third good reason to use constants. If a constant is used throughout a program and its value needs to be modified, then you have to change it in only one place. For example, if the capacity of the theater changes (because of a renovation) from 427 to 535, then you have to change only one declaration, and all uses of `MAX_OCCUPANCY` automatically reflect the change. If the literal 427 had been used throughout the code, each use would have to be found and changed. If you were to miss one or two, problems would surely arise.

## 2.4 primitive data types

There are eight primitive data types in Java: four subsets of integers, two subsets of floating point numbers, a character data type, and a boolean data type. Everything else is represented using objects. Let's examine these eight primitive data types in some detail.

### integers and floating points

key  
concept

Java has two kinds of numeric values: integers and floating point. There are four integer data types (`byte`, `short`, `int`, and `long`) and two floating point data types (`float` and `double`).

Java has two basic kinds of numeric values: integers, which have no fractional part, and floating points, which do. There are four integer data types (`byte`, `short`, `int`, and `long`) and two floating point data types (`float` and `double`). All of the numeric types differ by the amount of memory space used to store a value of that type, which determines the range of values that can be represented. The size of each data type is the same for all hardware platforms. All numeric types are *signed*, meaning that both positive and negative values can be stored in them. Figure 2.4 summarizes the numeric primitive types.

Remember from our discussion in Chapter 1 that a bit can be either a 1 or a 0. Because each bit can represent two different states, a string of  $n$  bits can be



| Type   | Storage | Min Value  | Max Value   |
|--------|---------|--|---|
| byte   | 8 bits  | -128   | 127   |
| short  | 16 bits | -32,768  | 32,767  |
| int    | 32 bits | -2,147,483,648   | 2,147,483,647   |
| long   | 64 bits | -9,223,372,036,854,775,808                                     | 9,223,372,036,854,775,807                                     |
| float  | 32 bits | Approximately $-3.4\text{E}+38$<br>with 7 significant digits   | Approximately $3.4\text{E}+38$<br>with 7 significant digits   |
| double | 64 bits | Approximately $-1.7\text{E}+308$<br>with 15 significant digits | Approximately $1.7\text{E}+308$<br>with 15 significant digits |

**figure 2.4** The Java numeric primitive types

used to represent  $2^n$  different values. Appendix B describes number systems and these kinds of relationships in more detail.

**web  
bonus**

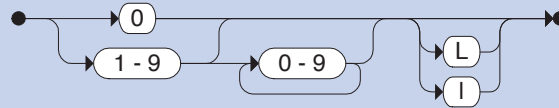
The book's Web site includes a description of the internal storage representation of primitive data types.

When designing a program, we sometimes need to be careful about picking variables of appropriate size so that memory space is not wasted. For example, if a value will not vary outside of a range of 1 to 1000, then a two-byte integer (`short`) is large enough to accommodate it. On the other hand, when it's not clear what the range of a particular variable will be, we should provide a reasonable, even generous, amount of space. In most situations memory space is not a serious restriction, and we can usually afford generous assumptions.

Note that even though a `float` value supports very large (and very small) numbers, it only has seven significant digits. Therefore if it is important to accurately maintain a value such as 50341.2077, we need to use a `double`.

A *literal* is an explicit data value used in a program. The various numbers used in programs such as `Facts` and `Addition` and `PianoKeys` are all *integer literals*. Java assumes all integer literals are of type `int`, unless an `L` or `l` is appended to the end of the value to indicate that it should be considered a literal of type `long`, such as `45L`.

Likewise, Java assumes that all *floating point literals* are of type `double`. If we need to treat a floating point literal as a `float`, we append an `F` or `f` to the end

**Decimal Integer Literal**

An integer literal is composed of a series of digits followed by an optional suffix to indicate that it should be considered a `long` integer. Negation of a literal is considered a separate operation.

Examples:

```
5
2594
4920328L
```

of the value, as in `2.718F` or `123.45f`. Numeric literals of type `double` can be followed by a `D` or `d` if desired.

The following are examples of numeric variable declarations in Java:

```
int answer = 42;
byte smallNumber1, smallNumber2;
long countedStars = 86827263927L;
float ratio = 0.2363F;
double delta = 453.523311903;
```

## characters

Characters are another fundamental type of data used and managed on a computer. Individual characters can be treated as separate data items, and as we've seen in several example programs, they can be combined to form character strings.

A *character literal* is expressed in a Java program with single quotes, such as `'b'` or `'J'` or `';`. You will recall that *string literals* are delineated using double quotation marks, and that the `String` type is not a primitive data type in Java, it is a class name. We discuss the `String` class in detail later in this chapter.

Note the difference between a digit as a character (or part of a string) and a digit as a number (or part of a larger number). The number `602` is a numeric value

that can be used in an arithmetic calculation. But in the string “602 Greenbriar Court” the 6, 0, and 2 are characters, just like the rest of the characters that make up the string.

The characters we can manage are defined by a *character set*, which is simply a list of characters in a particular order. Each programming language supports a particular character set that defines the valid values for a character variable in that language. Several character sets have been proposed, but only a few have been used regularly over the years. The *ASCII character set* is a popular choice. ASCII stands for the American Standard Code for Information Interchange. The basic ASCII set uses seven bits per character, providing room to support 128 different characters, including:

- uppercase letters, such as ‘A’, ‘B’, and ‘C’
- lowercase letters, such as ‘a’, ‘b’, and ‘c’
- punctuation, such as the period (‘.’), semicolon (‘;’), and comma (‘,’)
- the digits ‘0’ through ‘9’
- the space character, ‘ ’
- special symbols, such as the ampersand (‘&’), vertical bar (‘|’), and backslash (‘\’)
- control characters, such as the carriage return, null, and end-of-text marks

The *control characters* are sometimes called nonprinting or invisible characters because they do not have a specific symbol that represents them. Yet they are as valid as any other character and can be stored and used in the same ways. Many control characters have special meaning to certain software applications.

As computing became a worldwide endeavor, users demanded a more flexible character set containing other language alphabets. ASCII was extended to use eight bits per character, and the number of characters in the set doubled to 256. The extended ASCII contains many accented and diacritical characters not used in English.

However, even with 256 characters, the ASCII character set cannot represent the world’s alphabets, especially given the various Asian alphabets and their many thousands of ideograms. Therefore the developers of the Java programming language chose the *Unicode character set*, which uses 16 bits per character, supporting 65,536 unique characters. The characters and symbols from many languages are included in the Unicode definition. ASCII is a subset of the Unicode character set. Appendix C discusses the Unicode character set in more detail.

In Java, the data type `char` represents a single character. The following are some examples of character variable declarations in Java:

```
char topGrade = 'A';
char symbol1, symbol2, symbol3;
char terminator = ';', separator = ' ';
```

## booleans

A boolean value, defined in Java using the reserved word `boolean`, has only two valid values: `true` and `false`. A boolean variable is usually used to indicate whether a particular condition is true, but it can also be used to represent any situation that has two states, such as a light bulb being on or off.

A boolean value cannot be converted to any other data type, nor can any other data type be converted to a boolean value. The words `true` and `false` are reserved in Java as *boolean literals* and cannot be used outside of this context.

The following are some examples of boolean variable declarations in Java:

```
boolean flag = true;
boolean tooHigh, tooSmall, tooRough;
boolean done = false;
```

## 2.5 arithmetic expressions

An *expression* is a combination of one or more operators and operands. Expressions usually perform a calculation. The value calculated does not have to be a number, but it often is. The operands used in the operations might be literals, constants, variables, or other sources of data. The manner in which expressions are evaluated and used is fundamental to programming.

For now we will focus on arithmetic expressions that use numeric operands and produce numeric results. The usual arithmetic operations are defined for both integer and floating point numeric types, including addition (+), subtraction (−), multiplication (\*), and division (/). Java also has another arithmetic operation: The *remainder operator* (%) returns the remainder after dividing the second operand into the first. The sign of the result of a remainder operation is the sign of the numerator. Therefore,  $17\%4$  equals 1,  $-20\%3$  equals −2,  $10\%-5$  equals 0, and  $3\%8$  equals 3.

### key concept

Many programming statements involve expressions. Expressions are combinations of one or more operands and the operators used to perform a calculation.

As you might expect, if either or both operands to any numeric operator are floating point values, the result is a floating point value. However, the division operator produces results that are less intuitive, depending on the types of the operands. If both operands are integers, the `/` operator performs *integer division*, meaning that any fractional part of the result is discarded. If one or the other or both operands are floating point values, the `/` operator performs *floating point division*, and the fractional part of the result is kept. For example, the result of `10/4` is 2, but the results of `10.0/4` and `10/4.0` and `10.0/4.0` are all 2.5.

## operator precedence

Operators can be combined to create more complex expressions. For example, consider the following assignment statement:

```
result = 14 + 8 / 2;
```

The entire right-hand side of the assignment is evaluated, and then the result is stored in the variable. But what is the result? It is 11 if the addition is performed first, or it is 18 if the division is performed first. The order of operator evaluation makes a big difference. In this case, the division is performed before the addition, yielding a result of 18. You should note that in this and subsequent examples we have used literal values rather than variables to simplify the expression. The order of operator evaluation is the same if the operands are variables or any other source of data.

All expressions are evaluated according to an *operator precedence hierarchy* that establishes the rules that govern the order in which operations are evaluated. In the case of arithmetic operators, multiplication, division, and the remainder operator all have equal precedence and are performed before addition and subtraction. Any arithmetic operators at the same level of precedence are performed left to right. Therefore we say the arithmetic operators have a *left-to-right association*.

Precedence, however, can be forced in an expression by using parentheses. For instance, if we really wanted the addition to be performed first in the previous example, we could write the expression as follows:

```
result = (14 + 8) / 2;
```

Any expression in parentheses is evaluated first. In complicated expressions, it is good practice to use parentheses even when it is not strictly necessary in order to make it clear how the expression is evaluated.

Java follows a well-defined set of rules that govern the order in which operators will be evaluated in an expression. These rules form an operator precedence hierarchy.

key  
concept

Parentheses can be nested, and the innermost nested expressions are evaluated first. Consider the following expression:

```
result = 3 * ((18 - 4) / 2);
```

In this example, the result is 21. First, the subtraction is performed, forced by the inner parentheses. Then, even though multiplication and division are at the same level of precedence and usually would be evaluated left to right, the division is performed first because of the outer parentheses. Finally, the multiplication is performed.

After the arithmetic operations are complete, the computed result is stored in the variable on the left-hand side of the assignment operator (=). In other words, the assignment operator has a lower precedence than any of the arithmetic operators.

Figure 2.5 shows a precedence table with the relationships between the arithmetic operators, parentheses, and the assignment operator. Appendix D includes a full precedence table showing all Java operators.

A *unary operator* has only one operand, while a *binary operator* has two. The + and – arithmetic operators can be either unary or binary. The binary versions accomplish addition and subtraction, and the unary versions represent positive and negative numbers. For example, 1 is an example of using the unary negation operator to make the value negative.

| Precedence Level | Operator | Operation            | Associates |
|------------------|----------|----------------------|------------|
| 1                | +        | unary plus           | R to L     |
|                  | –        | unary minus          |            |
| 2                | *        | multiplication       | L to R     |
|                  | /        | division             |            |
|                  | %        | remainder            |            |
| 3                | +        | addition             | L to R     |
|                  | –        | subtraction          |            |
|                  | +        | string concatenation |            |
| 4                | =        | assignment           | R to L     |

**figure 2.5** Precedence among some of the Java operators

For an expression to be syntactically correct, the number of left parentheses must match the number of right parentheses and they must be properly nested. The following examples are *not* valid expressions:

```
result = ((19 + 8) % 3) - 4);    // not valid
result = (19 (+ 8 %) 3 - 4);    // not valid
```

The program in Listing 2.7, called `TempConverter`, converts a Celsius temperature value to its equivalent Fahrenheit value. Note that the operands to the division operation are double to ensure that the fractional part of the number

### listing 2.7



```
/**
 * TempConverter.java      Author: Lewis/Loftus
 *
 * Demonstrates the use of primitive data types and arithmetic
 * expressions.
 */

public class TempConverter
{
    //-----
    // Computes the Fahrenheit equivalent of a specific Celsius
    // value using the formula F = (9/5)C + 32.
    //-----
    public static void main (String[] args)
    {
        final int BASE = 32;
        final double CONVERSION_FACTOR = 9.0 / 5.0;

        int celsiusTemp = 24; // value to convert
        double fahrenheitTemp;

        fahrenheitTemp = celsiusTemp * CONVERSION_FACTOR + BASE;

        System.out.println ("Celsius Temperature: " + celsiusTemp);
        System.out.println ("Fahrenheit Equivalent: " + fahrenheitTemp);
    }
}
```

### output

```
Celsius Temperature: 24
Fahrenheit Equivalent: 75.2
```

is kept. The precedence rules dictate that the multiplication happens before the addition in the final conversion computation, which is what we want.

## data conversion

Because Java is a strongly typed language, each data value is associated with a particular type. It is sometimes helpful or necessary to convert a data value of one type to another type, but we must be careful that we don't lose important information in the process. For example, suppose a `short` variable that holds the number 1000 is converted to a `byte` value. Because a `byte` does not have enough bits to represent the value 1000, some bits would be lost in the conversion, and the number represented in the `byte` would not keep its original value.

A conversion between one primitive type and another falls into one of two categories: widening conversions and narrowing conversions. *Widening conversions* are the safest because they usually do not lose information. They are called widening conversions because they go from one data type to another type that uses an equal or greater amount of space to store the value. Figure 2.6 lists the Java widening conversions.

For example, it is safe to convert from a `byte` to a `short` because a `byte` is stored in 8 bits and a `short` is stored in 16 bits. There is no loss of information. All widening conversions that go from an integer type to another integer type, or from a floating point type to another floating point type, preserve the numeric value exactly.

Although widening conversions do not lose any information about the magnitude of a value, the widening conversions that result in a floating point value can lose precision. When converting from an `int` or a `long` to a `float`, or from

| From  | To                                 |
|-------|------------------------------------|
| byte  | short, int, long, float, or double |
| short | int, long, float, or double        |
| char  | int, long, float, or double        |
| int   | long, float, or double             |
| long  | float or double                    |
| float | double                             |

figure 2.6 Java widening conversions



a `long` to a `double`, some of the least significant digits may be lost. In this case, the resulting floating point value will be a rounded version of the integer value, following the rounding techniques defined in the IEEE 754 floating point standard.

*Narrowing conversions* are more likely to lose information than widening conversions are. They often go from one type to a type that uses less space to store a value, and therefore some of the information may be compromised. Narrowing conversions can lose both numeric magnitude and precision. Therefore, in general, they should be avoided. Figure 2.7 lists the Java narrowing conversions.

Avoid narrowing conversions because they can lose information.

key  
concept

An exception to the space-shrinking situation in narrowing conversions is when we convert a `byte` (8 bits) or `short` (16 bits) to a `char` (16 bits). These are still considered narrowing conversions because the sign bit is incorporated into the new character value. Since a character value is unsigned, a negative integer will be converted into a character that has no particular relationship to the numeric value of the original integer.

Note that `boolean` values are not mentioned in either widening or narrowing conversions. A `boolean` value cannot be converted to any other primitive type and vice versa.

In Java, conversions can occur in three ways:

- assignment conversion
- arithmetic promotion
- casting

| From   | To                                     |
|--------|--|
| byte   | char                                   |
| short  | byte or char                           |
| char   | byte or short                          |
| int    | byte, short, or char                   |
| long   | byte, short, char, or int              |
| float  | byte, short, char, int, or long        |
| double | byte, short, char, int, long, or float |

figure 2.7 Java narrowing conversions

*Assignment conversion* occurs when a value of one type is assigned to a variable of another type during which the value is converted to the new type. Only widening conversions can be accomplished through assignment. For example, if `money` is a `float` variable and `dollars` is an `int` variable, then the following assignment statement automatically converts the value in `dollars` to a `float`:

```
money = dollars;
```

Therefore, if `dollars` contains the value 25, after the assignment, `money` contains the value 25.0. However, if we attempt to assign `money` to `dollars`, the compiler will issue an error message alerting us to the fact that we are attempting a narrowing conversion that could lose information. If we really want to do this assignment, we have to make the conversion explicit using a cast.

*Arithmetic promotion* occurs automatically when certain arithmetic operators need to modify their operands in order to perform the operation. For example, when a floating point value called `sum` is divided by an integer value called `count`, the value of `count` is promoted to a floating point value automatically, before the division takes place, producing a floating point result:

```
result = sum / count;
```

*Casting* is the most general form of conversion in Java. If a conversion can be accomplished at all in a Java program, it can be accomplished using a cast. A cast is a Java operator that is specified by a type name in parentheses. It is placed in front of the value to be converted. For example, to convert `money` to an integer value, we could put a cast in front of it:

```
dollars = (int) money;
```

The cast returns the value in `money`, truncating any fractional part. If `money` contained the value 84.69, then after the assignment, `dollars` would contain the value 84. Note, however, that the cast does not change the value in `money`. After the assignment operation is complete, `money` still contains the value 84.69.

Casts are helpful in many situations where we need to treat a value temporarily as another type. For example, if we want to divide the integer value `total` by the integer value `count` and get a floating point result, we could do it as follows:

```
result = (float) total / count;
```

First, the cast operator returns a floating point version of the value in `total`. This operation does not change the value in `total`. Then, `count` is treated as a floating point value via arithmetic promotion. Now the division operator will

perform floating point division and produce the intended result. If the cast had not been included, the operation would have performed integer division and truncated the answer before assigning it to `result`. Also note that because the cast operator has a higher precedence than the division operator, the cast operates on the value of `total`, not on the result of the division.

## 2.6 creating objects

A variable can hold either a primitive value or a *reference to an object*. Like variables that hold primitive types, a variable that serves as an object reference must be declared. A class is used to define an object, and the class name can be thought of as the type of an object. The declarations of object references have a similar structure to the declarations of primitive variables.

The following declaration creates a reference to a `String` object:

```
String name;
```

That declaration is like the declaration of an integer, in that the type is followed by the variable name we want to use. However, no string object actually exists yet. To create an object, we use the `new` operator:

```
name = new String ("James Gosling");
```

The act of creating an object using the `new` operator is called *instantiation*. An object is said to be an *instance* of a particular class. After the `new` operator creates the object, a *constructor* is invoked to help set it up initially. A constructor has the same name as the class and is similar to a method. In this example, the parameter to the constructor is a string literal that specifies the characters that the string object will hold.

The act of declaring the object reference variable and creating the object itself can be combined into one step by initializing the variable in the declaration, just as we do with primitive types:

```
String name = new String ("James Gosling");
```

After an object has been instantiated, we use the *dot operator* to access its methods. We've used the dot operator many times in previous programs, such as in calls to `System.out.println`. The dot operator is appended directly after the object reference, followed by the method being invoked. For example, to invoke

The `new` operator returns a reference to a newly created object.

key  
concept

the `length` method defined in the `String` class, we use the dot operator on the `name` reference variable:

```
count = name.length()
```

The `length` method does not take any parameters, but the parentheses are still necessary to indicate that a method is being invoked. Some methods produce a value that is *returned* when the method completes. The purpose of the `length` method of the `String` class is to determine and return the length of the string (the number of characters it contains). In this example, the returned value is assigned to the variable `count`. For the string “James Gosling”, the `length` method returns 13 (this includes the space between the first and last names). Some methods do not return a value.

An object reference variable (such as `name`) actually stores the address where the object is stored in memory. We explore the nuances of object references, instantiation, and constructors in later chapters.

## the string class

Let’s examine the `String` class in more detail. Strings in Java are objects represented by the `String` class. Figure 2.8 lists some of the more useful methods of the `String` class. The method headers are listed, and they indicate the type of information that must be passed to the method. The type shown in front of the method name is called the *return type* of the method and indicates the type of information that will be returned, if anything. A return type of `void` indicates that the method does not return a value. The returned value can be used in the calling method as needed.

Once a `String` object is created, its value cannot be lengthened, shortened, nor can any of its characters change. Thus we say that a `String` object is *immutable*. However, several methods in the `String` class return new `String` objects that are often the result of modifying the original string’s value.

Note also that some of the `String` methods refer to the *index* of a particular character. A character in a string can be specified by its position, or index, in the string. The index of the first character in a string is zero, the index of the next character is one, and so on. Therefore in the string “Hello”, the index of the character ‘H’ is zero and the character at index four is ‘o’.

Several `String` methods are exercised in the program called `StringMutation`, shown in Listing 2.8.

```
String (String str)
    Constructor: creates a new string object with the same characters as str.

char charAt (int index)
    Returns the character at the specified index.

int compareTo (String str)
    Returns an integer indicating if this string is lexically before (a negative return
    value), equal to (a zero return value), or lexically after (a positive return value),
    the string str.

String concat (String str)
    Returns a new string consisting of this string concatenated with str.

boolean equals (String str)
    Returns true if this string contains the same characters as str (including
    case) and false otherwise.

boolean equalsIgnoreCase (String str)
    Returns true if this string contains the same characters as str (without
    regard to case) and false otherwise.

int length ()
    Returns the number of characters in this string.

String replace (char oldChar, char newChar)
    Returns a new string that is identical with this string except that every
    occurrence of oldChar is replaced by newChar.

String substring (int offset, int endIndex)
    Returns a new string that is a subset of this string starting at index offset
    and extending through endIndex-1.

String toLowerCase ()
    Returns a new string identical to this string except all uppercase letters are
    converted to their lowercase equivalent.

String toUpperCase ()
    Returns a new string identical to this string except all lowercase letters are
    converted to their uppercase equivalent.
```

**figure 2.8** Some methods of the String class

Figure 2.9 shows the String objects that are created in the StringMutation program. Compare this diagram to the program code and the output. Keep in mind that this is not a single String object that changes its data; this program creates five separate String objects using various methods of the String class.

**Listing  
2.8**

```

//*****
//  StringMutation.java      Author: Lewis/Loftus
//
//  Demonstrates the use of the String class and its methods.
//*****

public class StringMutation
{
    //-----
    //  Prints a string and various mutations of it.
    //-----
    public static void main (String[] args)
    {
        String phrase = new String ("Change is inevitable");
        String mutation1, mutation2, mutation3, mutation4;

        System.out.println ("Original string: \"" + phrase + "\"");
        System.out.println ("Length of string: " + phrase.length());

        mutation1 = phrase.concat (" , except from vending machines.");
        mutation2 = mutation1.toUpperCase();
        mutation3 = mutation2.replace ('E', 'X');
        mutation4 = mutation3.substring (3, 30);

        // Print each mutated string
        System.out.println ("Mutation #1: " + mutation1);
        System.out.println ("Mutation #2: " + mutation2);
        System.out.println ("Mutation #3: " + mutation3);
        System.out.println ("Mutation #4: " + mutation4);

        System.out.println ("Mutated length: " + mutation4.length());
    }
}

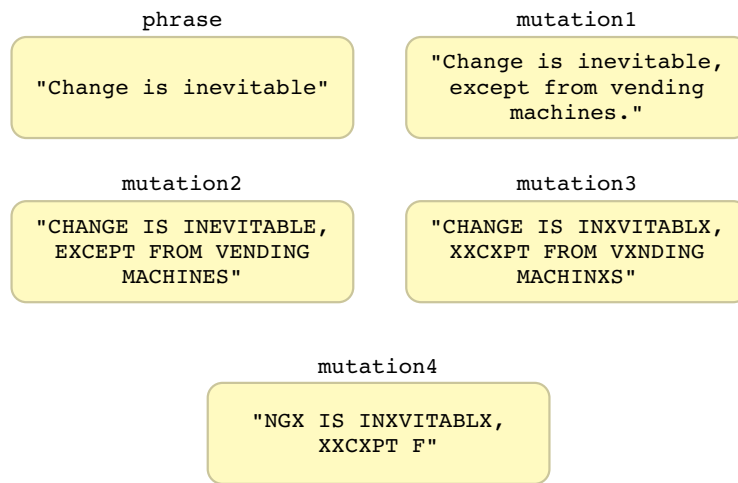
```

**output**

```

Original string: "Change is inevitable"
Length of string: 20
Mutation #1: Change is inevitable, except from vending machines.
Mutation #2: CHANGE IS INEVITABLE, EXCEPT FROM VENDING MACHINES.
Mutation #3: CHANGX IS INXVITABLX, XXCXPT FROM VXNDING MACHINXS.
Mutation #4: NGX IS INXVITABLX, XXCXPT F
Mutated length: 27

```



**figure 2.9** The String objects created in the StringMutation program

Even though they are not primitive types, strings are so fundamental and so often used that Java defines string literals delimited by double quotation marks, as we've seen in various examples. This is a shortcut notation. Whenever a string literal appears, a `String` object is created. Therefore the following declaration is valid:

```
String name = "James Gosling";
```

That is, for `String` objects, the explicit use of the `new` operator and the call to the constructor can be eliminated. In most cases, we will use this simplified syntax.

## 2.7 class libraries and packages

A *class library* is a set of classes that supports the development of programs. A compiler often comes with a class library. Class libraries can also be obtained separately through third-party vendors. The classes in a class library contain methods that are often valuable to a programmer because of the special functionality they offer. In fact, programmers often become dependent on the methods in a class library and begin to think of them as part of the language. However, technically, they are not in the language definition.

key  
concept

The Java standard class library is a useful set of classes that anyone can use when writing Java programs.

The `String` class, for instance, is not an inherent part of the Java language. It is part of the Java *standard class library* that can be found in any Java development environment. The classes that make up the library were created by employees at Sun Microsystems, the people who created the Java language.

The class library is made up of several clusters of related classes, which are sometimes called Java APIs, or *Application Programmer Interface*. For example, we may refer to the Java Database API when we're talking about the set of classes that help us write programs that interact with a database. Another example of an API is the Java Swing API, which refers to a set of classes that define special graphical components used in a graphical user interface (GUI). Sometimes the entire standard library is referred to generically as the Java API, though we generally avoid that use.

key  
concept

A package is a Java language element used to group related classes under a common name.

The classes of the Java standard class library are also grouped into *packages*, which, like the APIs, let us group related classes by one name. Each class is part of a particular package. The `String` class, for example, is part of the `java.lang` package. The `System` class is part of the `java.lang` package as well. Figure 2.10 shows the organizations of packages in the overall library.

The package organization is more fundamental and language based than the API names. Though there is a general correspondence between package and API names, the groups of classes that make up a given API might cross packages. We primarily refer to classes in terms of their package organization in this text.

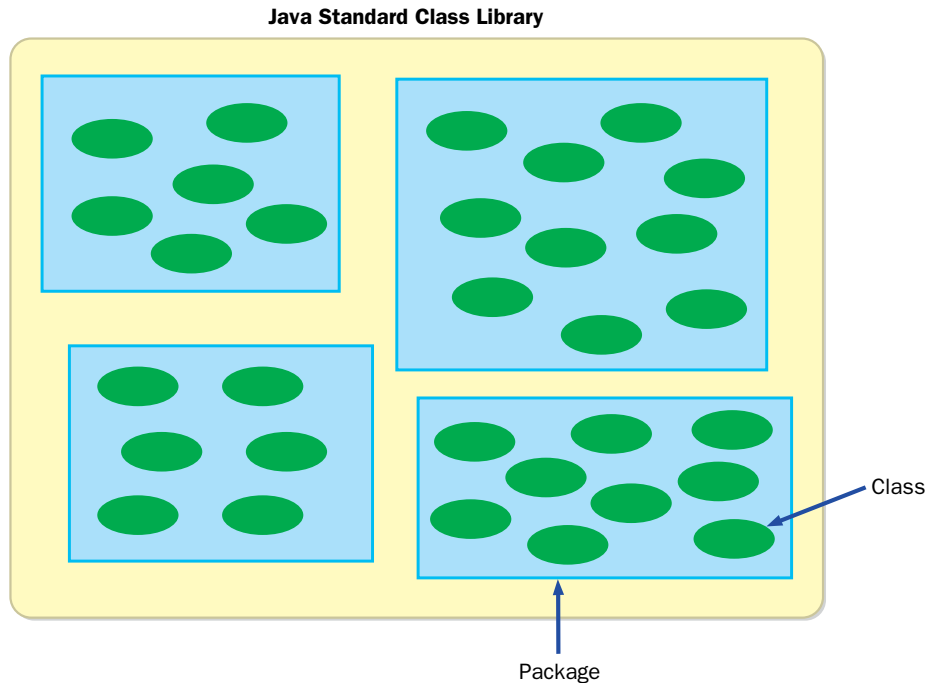
Figure 2.11 describes some of the packages that are part of the Java standard class library. These packages are available on any platform that supports Java software development. Many of these packages support highly specific programming techniques and will not come into play in the development of basic programs.

Various classes of the Java standard class library are discussed throughout this book. Appendix M serves as a general reference for many of the classes in the Java class library.

## the import declaration

The classes of the package `java.lang` are automatically available for use when writing a program. To use classes from any other package, however, we must either *fully qualify* the reference, or use an *import declaration*.





**figure 2.10** Classes organized into packages in the Java standard class library

When you want to use a class from a class library in a program, you could use its fully qualified name, including the package name, every time it is referenced. For example, every time you want to refer to the `Random` class that is defined in the `java.util` package, you can write `java.util.Random`. However, completely specifying the package and class name every time it is needed quickly becomes tiring. Java provides the import declaration to simplify these references.

The import declaration identifies the packages and classes that will be used in a program so that the fully qualified name is not necessary with each reference. The following is an example of an import declaration:

```
import java.util.Random;
```

This declaration asserts that the `Random` class of the `java.util` package may be used in the program. Once this import declaration is made, it is sufficient to use the simple name `Random` when referring to that class in the program.

Another form of the import declaration uses an asterisk (\*) to indicate that any class inside the package might be used in the program. Therefore, the following

| Package                        | Provides support to   |
|--------------------------------|---|
| <code>java.applet</code>       | Create programs (applets) that are easily transported across the Web.                                       |
| <code>java.awt</code>          | Draw graphics and create graphical user interfaces; AWT stands for Abstract Windowing Toolkit.              |
| <code>java.beans</code>        | Define software components that can be easily combined into applications.                                   |
| <code>java.io</code>           | Perform a wide variety of input and output functions.   |
| <code>java.lang</code>         | General support; it is automatically imported into all Java programs.                                       |
| <code>java.math</code>         | Perform calculations with arbitrarily high precision.   |
| <code>java.net</code>          | Communicate across a network.   |
| <code>java.rmi</code>          | Create programs that can be distributed across multiple computers; RMI stands for Remote Method Invocation. |
| <code>java.security</code>     | Enforce security restrictions.  |
| <code>java.sql</code>          | Interact with databases; SQL stands for Structured Query Language.  |
| <code>java.text</code>         | Format text for output.   |
| <code>java.util</code>         | General utilities.  |
| <code>javax.swing</code>       | Create graphical user interfaces with components that extend the AWT capabilities.                          |
| <code>javax.xml.parsers</code> | Process XML documents; XML stands for eXtensible Markup Language.   |

**figure 2.11** Some packages in the Java standard class library

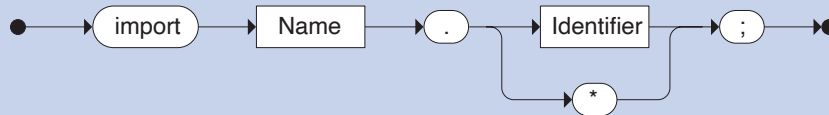
declaration allows all classes in the `java.util` package to be referenced in the program without the explicit package name:

```
import java.util.*;
```

If only one class of a particular package will be used in a program, it is usually better to name the class specifically in the `import` statement. However, if two or more will be used, the `*` notation is fine. Once a class is imported, it is as if its code has been brought into the program. The code is not actually moved, but that is the effect.

The classes of the `java.lang` package are automatically imported because they are fundamental and can be thought of as basic extensions to the language.

### Import Declaration



An import declaration specifies an Identifier (the name of a class) that will be referenced in a program, and the Name of the package in which it is defined. The \* wildcard indicates that any class from a particular package may be referenced.

Examples:

```
import java.util.*;
import cs1.Keyboard;
```

Therefore, any class in the `java.lang` package, such as `String`, can be used without an explicit `import` statement. It is as if all programs automatically contain the following statement:

```
import java.lang.*;
```

## the Random class

The need for random numbers occurs frequently when writing software. Games often use a random number to represent the roll of a die or the shuffle of a deck of cards. A flight simulator may use random numbers to determine how often a simulated flight has engine trouble. A program designed to help high school students prepare for the SATs may use random numbers to choose the next question to ask.

The `Random` class implements a *pseudorandom number generator*. A random number generator picks a number at random out of a range of values. A program that serves this role is technically pseudorandom, because a program has no means to actually pick a number randomly. A pseudorandom number generator might perform a series of complicated calculations, starting with an initial *seed value*, and produces a number. Though they are technically not random (because they are calculated), the values produced by a pseudorandom number generator

usually appear random, at least random enough for most situations. Figure 2.12 lists some of the methods of the `Random` class.

The `nextInt` method can be called with no parameters, or we can pass it a single integer value. The version that takes no parameters generates a random number across the entire range of `int` values, including negative numbers. Usually, though, we need a random number within a more specific range. For instance, to simulate the roll of a die we might want a random number in the range of 1 to 6. If we pass a value, say `N`, to `nextInt`, the method returns a value from 0 to `N-1`. For example, if we pass in 100, we'll get a return value that is greater than or equal to 0 and less than or equal to 99.

Note that the value that we pass to the `nextInt` method is also the number of possible values we can get in return. We can shift the range as needed by adding or subtracting the proper amount. To get a random number in the range 1 to 6, we can call `nextInt(6)` to get a value from 0 to 5, and then add 1.

The `nextFloat` method of the `Random` class returns a `float` value that is greater than or equal to 0.0 and less than 1.0. If desired, we can use multiplication to scale the result, cast it into an `int` value to truncate the fractional part, then shift the range as we do with integers.

The program shown in Listing 2.9 produces several random numbers in various ranges.

```
Random ()  
    Constructor: creates a new pseudorandom number generator.  
  
float nextFloat ()  
    Returns a random number between 0.0 (inclusive) and 1.0 (exclusive).  
  
int nextInt ()  
    Returns a random number that ranges over all possible int values (positive and  
    negative).  
  
int nextInt (int num)  
    Returns a random number in the range 0 to num-1.
```

**figure 2.12** Some methods of the `Random` class

**listing**  
**2.9**

```
//*****
// RandomNumbers.java      Author: Lewis/Loftus
//
// Demonstrates the import statement, and the creation of pseudo-
// random numbers using the Random class.
//*****

import java.util.Random;

public class RandomNumbers
{
    //-----
    // Generates random numbers in various ranges.
    //-----
    public static void main (String[] args)
    {
        Random generator = new Random();
        int num1;
        float num2;

        num1 = generator.nextInt();
        System.out.println ("A random integer: " + num1);

        num1 = generator.nextInt(10);
        System.out.println ("From 0 to 9: " + num1);

        num1 = generator.nextInt(10) + 1;
        System.out.println ("From 1 to 10: " + num1);

        num1 = generator.nextInt(15) + 20;
        System.out.println ("From 20 to 34: " + num1);

        num1 = generator.nextInt(20) - 10;
        System.out.println ("From -10 to 9: " + num1);

        num2 = generator.nextFloat();
        System.out.println ("A random float [between 0-1]: " + num2);

        num2 = generator.nextFloat() * 6; // 0.0 to 5.999999
        num1 = (int) num2 + 1;
        System.out.println ("From 1 to 6: " + num1);
    }
}
```

**listing  
2.9**

continued

**output**

```
A random integer: -889285970
0 to 9: 6
1 to 10: 9
10 to 29: 18
A random float [between 0-1] : 0.8815305
1 to 6: 2
```

## 2.8 invoking class methods

Some methods can be invoked through the class name in which they are defined, without having to instantiate an object of the class first. These are called *class methods* or *static methods*. Let's look at some examples.

### the math class

The `Math` class provides a large number of basic mathematical functions. The `Math` class is part of the Java standard class library and is defined in the `java.lang` package. Figure 2.13 lists several of its methods.

The reserved word `static` indicates that the method can be invoked through the name of the class. For example, a call to `Math.abs(total)` will return the absolute value of the number stored in `total`. A call to `Math.pow(7, 4)` will return 7 raised to the fourth power. Note that you can pass integer values to a method that accepts a `double` parameter. This is a form of assignment conversion, which we discussed earlier in this chapter.

We'll make use of some `Math` methods in examples after examining the `Keyboard` class.

### the keyboard class

The `Keyboard` class contains methods that help us obtain input data that the user types on the keyboard. The methods of the `Keyboard` class are static and are therefore invoked through the `Keyboard` class name.

```
static int abs (int num)
    Returns the absolute value of num.

static double acos (double num)

static double asin (double num)

static double atan (double num)
    Returns the arc cosine, arc sine, or arc tangent of num.

static double cos (double angle)

static double sin (double angle)

static double tan (double angle)
    Returns the angle cosine, sine, or tangent of angle, which is measured
    in radians.

static double ceil (double num)
    Returns the ceiling of num, which is the smallest whole number greater
    than or equal to num.

static double exp (double power)
    Returns the value e raised to the specified power.

static double floor (double num)
    Returns the floor of num, which is the largest whole number less than
    or equal to num.

static double pow (double num, double power)
    Returns the value num raised to the specified power.

static double random ()
    Returns a random number between 0.0 (inclusive) and 1.0 (exclusive).

static double sqrt (double num)
    Returns the square root of num, which must be positive.
```

**figure 2.13** Some methods of the Math class

One very important characteristic of the `Keyboard` class must be made clear: The `Keyboard` class is *not* part of the Java standard class library. It has been written by the authors of this book to help you read user input. It is defined as part of a package called `cs1` (that's cs-one, not cs-el). Because it is not part of the Java standard class library, it will not be found on generic Java development environments.

**key  
concept**

The `Keyboard` class is not part of the Java standard library. It is therefore not available on all Java development platforms.

You may have to configure your environment so that it knows where to find the `Keyboard` class.

The process of reading input from the user in Java can get somewhat involved. The `Keyboard` class allows you to ignore those details for now. We explore these issues later in the book, at which point we fully explain the details currently hidden by the `Keyboard` class.

For now we will use the `Keyboard` class for the services it provides, just as we do any other class. In that sense, the `Keyboard` class is a good example of object abstraction. We rely on classes and objects for the services they provide. It doesn't matter if they are part of a library, if a third party writes them, or if we write them ourselves. We use and interact with them in the same way. Figure 2.14 lists the input methods of the `Keyboard` class.

**web  
bonus**

For each example in this book that uses the `Keyboard` class, the Web site contains a version of the program that does not use it (for comparison purposes).

Let's look at some examples that use the `Keyboard` class. The program shown in Listing 2.10, called `Echo`, simply reads a string that is typed by the user and echoes it back to the screen.

```
static boolean readBoolean ()
static byte readByte ()
static char readChar ()
static double readDouble ()
static float readFloat ()
static int readInt ()
static long readLong ()
static short readShort ()
static String readString ()
    Returns a value of the indicated type obtained from user keyboard input.
```

**figure 2.14** Some methods of the `Keyboard` class



**listing**  
**2.10**

```
//*****
//  Echo.java          Author: Lewis/Loftus
//
//  Demonstrates the use of the readString method of the Keyboard
//  class.
//*****

import cs1.Keyboard;

public class Echo
{
    //-----
    //  Reads a character string from the user and prints it.
    //-----
    public static void main (String[] args)
    {
        String message;

        System.out.println ("Enter a line of text:");

        message = Keyboard.readString();

        System.out.println ("You entered: \"" + message + "\"");
    }
}
```

**output**

```
Enter a line of text:
Set your laser printer on stun!
You entered: "Set your laser printer on stun!"
```

The Quadratic program, shown in Listing 2.11 uses the Keyboard and Math classes. Recall that a quadratic equation has the following general form:

$$ax^2 + bx + c$$

The `Quadratic` program reads values that represent the coefficients in a quadratic equation ( $a$ ,  $b$ , and  $c$ ), and then evaluates the quadratic formula to determine the roots of the equation. The quadratic formula is:

$$\text{roots} = -b \pm \sqrt{\frac{b^2 - 4 \times a \times c}{2 \times a}}$$

### listing 2.11



```
//*****
// Quadratic.java      Author: Lewis/Loftus
//
// Demonstrates a calculation based on user input.
//*****

import cs1.Keyboard;

public class Quadratic
{
    //-----
    // Determines the roots of a quadratic equation.
    //-----
    public static void main (String[] args)
    {
        int a, b, c; // ax^2 + bx + c

        System.out.print ("Enter the coefficient of x squared: ");
        a = Keyboard.readInt();

        System.out.print ("Enter the coefficient of x: ");
        b = Keyboard.readInt();

        System.out.print ("Enter the constant: ");
        c = Keyboard.readInt();

        // Use the quadratic formula to compute the roots.
        // Assumes a positive discriminant.

        double discriminant = Math.pow(b, 2) - (4 * a * c);
        double root1 = ((-1 * b) + Math.sqrt(discriminant)) / (2 * a);
        double root2 = ((-1 * b) - Math.sqrt(discriminant)) / (2 * a);
    }
}
```

**listing**  
**2.11** continued

```
        System.out.println ("Root #1: " + root1);
        System.out.println ("Root #2: " + root2);
    }
}
```

**output**

```
Enter the coefficient of x squared: 3
Enter the coefficient of x: 8
Enter the constant: 4
Root #1: -0.6666666666666666
Root #2: -2.0
```

## 2.9 formatting output

The `NumberFormat` class and the `DecimalFormat` class are used to format information so that it looks appropriate when printed or displayed. They are both part of the Java standard class library and are defined in the `java.text` package.

### the `NumberFormat` class

The `NumberFormat` class provides generic formatting capabilities for numbers. You don't instantiate a `NumberFormat` object using the `new` operator. Instead, you request an object from one of the methods that you can invoke through the class itself. The reasons for this approach involve issues that we haven't covered yet, but we explain them in due course. Figure 2.15 lists some of the methods of the `NumberFormat` class.

Two of the methods in the `NumberFormat` class, `getCurrencyInstance` and `getPercentInstance`, return an object that is used to format numbers. The `getCurrencyInstance` method returns a formatter for monetary values whereas the `getPercentInstance` method returns an object that formats a percentage. The `format` method is invoked through a formatter object and returns a `String` that contains the number formatted in the appropriate manner.

The `Price` program shown in Listing 2.12 uses both types of formatters. It reads in a sales transaction and computes the final price, including tax.

```
String format (double number)
    Returns a string containing the specified number formatted according to
    this object's pattern.

static NumberFormat getCurrencyInstance()
    Returns a NumberFormat object that represents a currency format for the
    current locale.

static NumberFormat getPercentInstance()
    Returns a NumberFormat object that represents a percentage format for
    the current locale.
```

figure 2.15 Some methods of the NumberFormat class

b listing  
2.12


```

//*****
// Price.java          Author: Lewis/Loftus
//
// Demonstrates the use of various Keyboard and NumberFormat
// methods.
//*****

import cs1.Keyboard;
import java.text.NumberFormat;

public class Price
{
    //-----
    // Calculates the final price of a purchased item using values
    // entered by the user.
    //-----
    public static void main (String[] args)
    {
        final double TAX_RATE = 0.06;  // 6% sales tax

        int quantity;
        double subtotal, tax, totalCost, unitPrice;

        System.out.print ("Enter the quantity: ");
        quantity = Keyboard.readInt();
    }
}

```

**listing**  
**2.12** continued

```
System.out.print ("Enter the unit price: ");
unitPrice = Keyboard.readDouble();

subtotal = quantity * unitPrice;
tax = subtotal * TAX_RATE;
totalCost = subtotal + tax;

// Print output with appropriate formatting
NumberFormat money = NumberFormat.getCurrencyInstance();
NumberFormat percent = NumberFormat.getPercentInstance();

System.out.println ("Subtotal: " + money.format(subtotal));
System.out.println ("Tax: " + money.format(tax) + " at "
                    + percent.format(TAX_RATE));
System.out.println ("Total: " + money.format(totalCost));
}
```

**output**

```
Enter the quantity: 5
Enter the unit price: 3.87
Subtotal: $19.35
Tax: $1.16 at 6%
Total: $20.51
```

## the DecimalFormat class

Unlike the `NumberFormat` class, the `DecimalFormat` class is instantiated in the traditional way using the `new` operator. Its constructor takes a string that represents the pattern that will guide the formatting process. We can then use the `format` method to format a particular value. At a later point, if we want to change the pattern that the formatter object uses, we can invoke the `applyPattern` method. Figure 2.16 describes these methods.

The pattern defined by the string that is passed to the `DecimalFormat` constructor gets fairly elaborate. Various symbols are used to represent particular formatting guidelines.

```
DecimalFormat (String pattern)
    Constructor: creates a new DecimalFormat object with the specified
    pattern.

void applyPattern (String pattern)
    Applies the specified pattern to this DecimalFormat object.

String format (double number)
    Returns a string containing the specified number formatted according to
```

figure 2.16 Some methods of the DecimalFormat class

#### web bonus

The book's Web site contains additional information about techniques for formatting information, including a discussion of the various patterns that can be defined for the DecimalFormat class.

The pattern defined by the string "0.###", for example, indicates that at least one digit should be printed to the left of the decimal point and should be a zero if the integer portion of the value is zero. It also indicates that the fractional portion of the value should be rounded to three digits. This pattern is used in the CircleStats program shown in Listing 2.13, which reads the radius of a circle from the user and computes its area and circumference. Trailing zeros, such as in the circle's area of 78.540, are not printed.

## 2.10 an introduction to applets

### key concept

Applets are Java programs that are usually transported across a network and executed using a Web browser. Java applications are stand-alone programs that can be executed using the Java interpreter.

There are two kinds of Java programs: Java applets and Java applications. A Java *applet* is a Java program that is intended to be embedded into an HTML document, transported across a network, and executed using a Web browser. A Java *application* is a stand-alone program that can be executed using the Java interpreter. All programs presented thus far in this book have been Java applications.

**listing**  
**2.13**

```
//*****  
// CircleStats.java      Author: Lewis/Loftus  
//  
// Demonstrates the formatting of decimal values using the  
// DecimalFormat class.  
//*****  
  
import cs1.Keyboard;  
import java.text.DecimalFormat;  
  
public class CircleStats  
{  
    //-----  
    // Calculates the area and circumference of a circle given its  
    // radius.  
    //-----  
    public static void main (String[] args)  
    {  
        int radius;  
        double area, circumference;  
  
        System.out.print ("Enter the circle's radius: ");  
        radius = Keyboard.readInt();  
  
        area = Math.PI * Math.pow(radius, 2);  
        circumference = 2 * Math.PI * radius;  
  
        // Round the output to three decimal places  
        DecimalFormat fmt = new DecimalFormat ("0.###");  
  
        System.out.println ("The circle's area: " + fmt.format(area));  
        System.out.println ("The circle's circumference: "  
                             + fmt.format(circumference));  
    }  
}
```

**output**

```
Enter the circle's radius: 5  
The circle's area: 78.54  
The circle's circumference: 31.416
```

The Web enables users to send and receive various types of media, such as text, graphics, and sound, using a point-and-click interface that is extremely convenient and easy to use. A Java applet was the first kind of executable program that could be retrieved using Web software. Java applets are considered just another type of media that can be exchanged across the Web.

Though Java applets are generally intended to be transported across a network, they don't have to be. They can be viewed locally using a Web browser. For that matter, they don't even have to be executed through a Web browser at all. A tool in Sun's Java Software Development Kit called `appletviewer` can be used to interpret and execute an applet. We use `appletviewer` to display most of the applets in the book. However, usually the point of making a Java applet is to provide a link to it on a Web page and allow it to be retrieved and executed by Web users anywhere in the world.

Java bytecode (not Java source code) is linked to an HTML document and sent across the Web. A version of the Java interpreter embedded in a Web browser is used to execute the applet once it reaches its destination. A Java applet must be compiled into bytecode format before it can be used with the Web.

There are some important differences between the structure of a Java applet and the structure of a Java application. Because the Web browser that executes an applet is already running, applets can be thought of as a part of a larger program. As such they do not have a `main` method where execution starts. The `paint` method in an applet is automatically invoked by the applet. Consider the program in Listing 2.14, in which the `paint` method is used to draw a few shapes and write a quotation by Albert Einstein to the screen.

The two `import` statements at the beginning of the program explicitly indicate the packages that are used in the program. In this example, we need the `Applet` class, which is part of the `java.applet` package, and various graphics capabilities defined in the `java.awt` package.

A class that defines an applet extends the `Applet` class, as indicated in the header line of the class declaration. This process is making use of the object-oriented concept of inheritance, which we explore in more detail in Chapter 7. Applet classes must also be declared as `public`.

The `paint` method is one of several applet methods that have particular significance. It is invoked automatically whenever the graphic elements of the applet need to be painted to the screen, such as when the applet is first run or when another window that was covering it is moved.



**listing**  
**2.14**

```

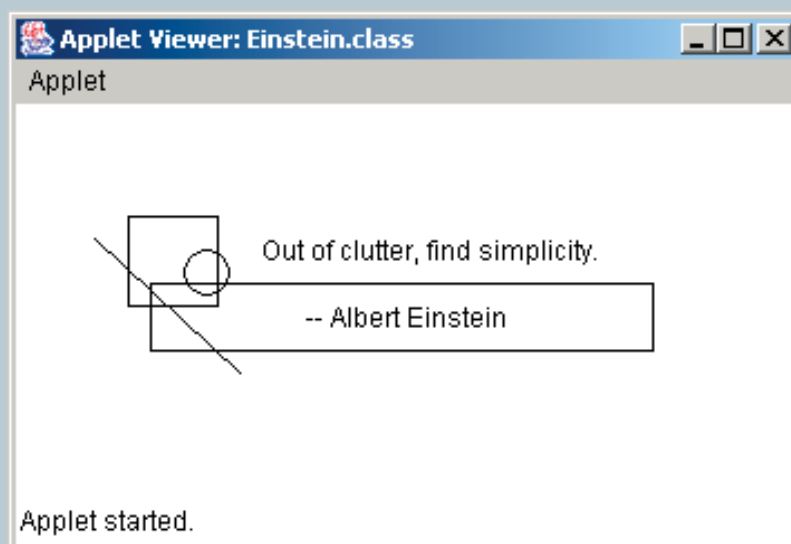
//*****
//  Einstein.java      Author: Lewis/Loftus
//
//  Demonstrates a basic applet.
//*****

import java.applet.Applet;
import java.awt.*;

public class Einstein extends Applet
{
    //-----
    //  Draws a quotation by Albert Einstein among some shapes.
    //-----
    public void paint (Graphics page)
    {
        page.drawRect (50, 50, 40, 40);    // square
        page.drawRect (60, 80, 225, 30);   // rectangle
        page.drawOval (75, 65, 20, 20);    // circle
        page.drawLine (35, 60, 100, 120);  // line

        page.drawString ("Out of clutter, find simplicity.", 110, 70);
        page.drawString ("-- Albert Einstein", 130, 100);
    }
}

```

**display**

Note that the `paint` method accepts a `Graphics` object as a parameter. A `Graphics` object defines a particular *graphics context* with which we can interact. The graphics context passed into an applet's `paint` method represents the entire applet window. Each graphics context has its own coordinate system. In later examples, we will have multiple components, each with its own graphic context.

A `Graphics` object allows us to draw various shapes using methods such as `drawRect`, `drawOval`, `drawLine`, and `drawString`. The parameters passed to the drawing methods specify the coordinates and sizes of the shapes to be drawn. We explore these and other methods that draw shapes in the next section.

## executing applets using the Web

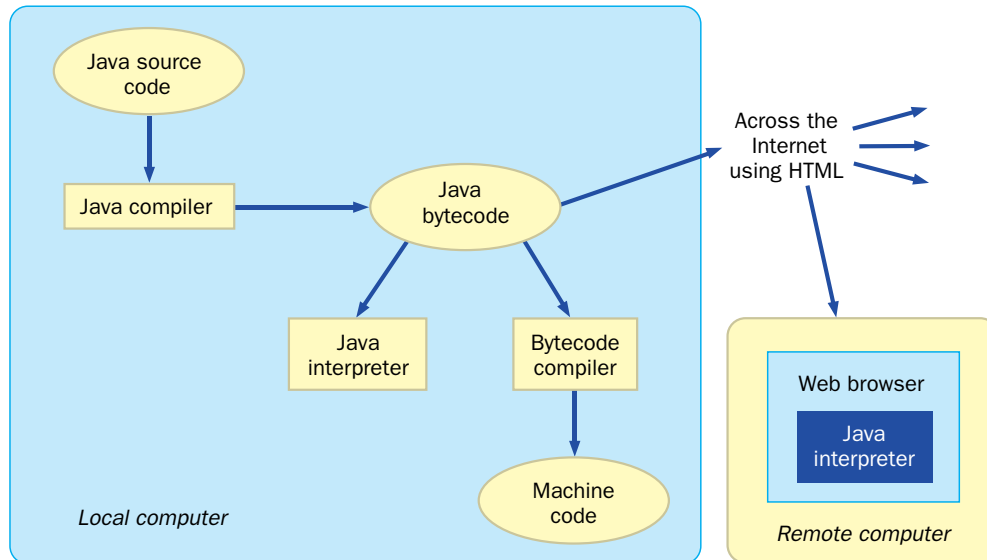
In order for the applet to be transmitted over the Web and executed by a browser, it must be referenced in a HyperText Markup Language (HTML) document. An HTML document contains *tags* that specify formatting instructions and identify the special types of media that are to be included in a document. A Java program is considered a specific media type, just as text, graphics, and sound are.

An HTML tag is enclosed in angle brackets. Appendix J contains a tutorial on HTML that explores various tag types. The following is an example of an applet tag:

```
<applet code="Einstein.class" width=350 height=175>
</applet>
```

This tag dictates that the bytecode stored in the file `Einstein.class` should be transported over the network and executed on the machine that wants to view this particular HTML document. The applet tag also indicates the width and height of the applet.

Note that the applet tag refers to the bytecode file of the `Einstein` applet, not to the source code file. Before an applet can be transported using the Web, it must be compiled into its bytecode format. Then, as shown in Fig. 2.17, the document can be loaded using a Web browser, which will automatically interpret and execute the applet.



**figure 2.17** The Java translation and execution process, including applets

## 2.11 drawing shapes

The Java standard class library provides many classes that let us present and manipulate graphical information. The `Graphics` class is fundamental to all such processing.

### the `Graphics` class

The `Graphics` class is defined in the `java.awt` package. It contains various methods that allow us to draw shapes, including lines, rectangles, and ovals. Figure 2.18 lists some of the fundamental drawing methods of the `Graphics` class. Note that these methods also let us draw circles and squares, which are just specific types of ovals and rectangles, respectively. We discuss additional drawing methods of the `Graphics` class later in the book at appropriate points.

The methods of the `Graphics` class allow us to specify whether we want a shape filled or unfilled. An unfilled shape shows only the outline of the shape and is otherwise transparent (you can see any underlying graphics). A filled shape is solid between its boundaries and covers any underlying graphics.

Most shapes can be drawn filled (opaque) or unfilled (as an outline).

key  
concept

```
void drawArc (int x, int y, int width, int height, int
startAngle, int arcAngle)
    Paints an arc along the oval bounded by the rectangle defined by x, y, width,
    and height. The arc starts at startAngle and extends for a distance defined by
    arcAngle.

void drawLine (int x1, int y1, int x2, int y2)
    Paints a line from point (x1, y1) to point (x2, y2).

void drawOval (int x, int y, int width, int height)
    Paints an oval bounded by the rectangle with an upper left corner of (x, y) and
    dimensions width and height.

void drawRect (int x, int y, int width, int height)
    Paints a rectangle with upper left corner (x, y) and dimensions width and
    height.

void drawString (String str, int x, int y)
    Paints the character string str at point (x, y), extending to the right.

void fillArc (int x, int y, int width, int height,
int startAngle, int arcAngle)

void fillOval (int x, int y, int width, int height)

void fillRect (int x, int y, int width, int height)
    Same as their draw counterparts, but filled with the current foreground color.

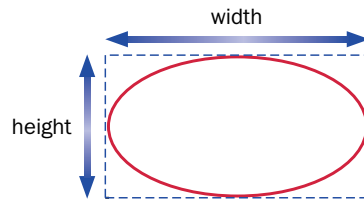
Color getColor ()
    Returns this graphics context's foreground color.

void setColor (Color color)
    Sets this graphics context's foreground color to the specified color.
```

**figure 2.18** Some methods of the Graphics class

All of these methods rely on the Java coordinate system, which we discussed in Chapter 1. Recall that point (0,0) is in the upper-left corner, such that *x* values get larger as we move to the right, and *y* values get larger as we move down. Any shapes drawn at coordinates that are outside the visible area will not be seen.

Many of the Graphics drawing methods are self-explanatory, but some require a little more discussion. Note, for instance, that an oval drawn by the



**figure 2.19** An oval and its bounding rectangle

`drawOval` method is defined by the coordinate of the upper-left corner and dimensions that specify the width and height of a *bounding rectangle*. Shapes with curves such as ovals are often defined by a rectangle that encompasses their perimeters. Figure 2.19 depicts a bounding rectangle for an oval.

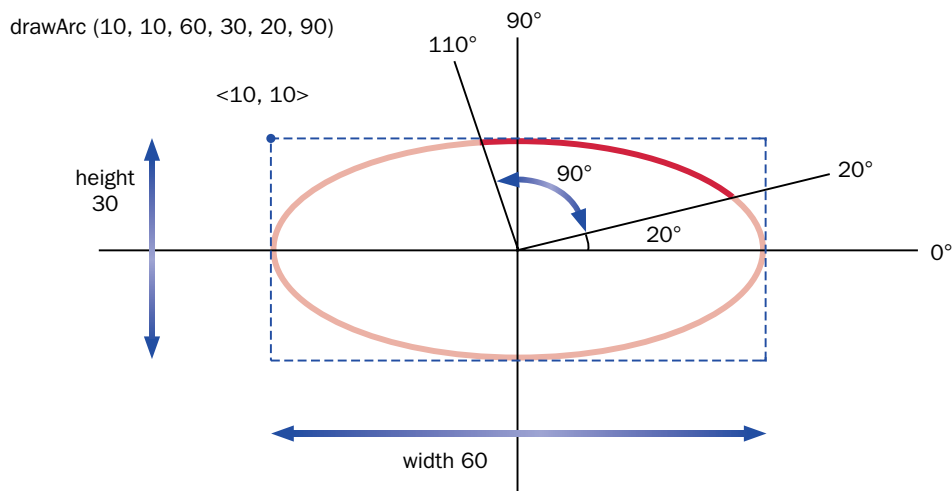
A bounding rectangle is often used to define the position and size of curved shapes such as ovals.

key  
concept

An arc can be thought of as a segment of an oval. To draw an arc, we specify the oval of which the arc is a part and the portion of the oval in which we're interested. The starting point of the arc is defined by the *start angle* and the ending point of the arc is defined by the *arc angle*. The arc angle does not indicate where the arc ends, but rather its range. The start angle and the arc angle are measured in degrees. The origin for the start angle is an imaginary horizontal line passing through the center of the oval and can be referred to as  $0^\circ$ ; as shown in Fig. 2.20.

An arc is a segment of an oval; the segment begins at a specific start angle and extends for a distance specified by the arc angle.

key  
concept



**figure 2.20** An arc defined by an oval, a start angle, and an arc angle

**web  
bonus**

The book's Web site contains additional information and examples about drawing shapes.

**the color class****key  
concept**

A `Color` class contains several common predefined colors.

In Java, a programmer uses the `Color` class, which is part of the `java.awt` package, to define and manage colors. Each object of the `Color` class represents a single color. The class contains several instances of itself to provide a basic set of predefined colors. Figure 2.21 lists the predefined colors of the `Color` class.

The `Color` class also contains methods to define and manage many other colors. Recall from Chapter 1 that colors can be defined using the RGB technique for specifying the contributions of three additive primary colors: red, green, and blue.

| Color      | Object                       | RGB Value     |
|------------|------------------------------|---------------|
| black      | <code>Color.black</code>     | 0, 0, 0       |
| blue       | <code>Color.blue</code>      | 0, 0, 255     |
| cyan       | <code>Color.cyan</code>      | 0, 255, 255   |
| gray       | <code>Color.gray</code>      | 128, 128, 128 |
| dark gray  | <code>Color.darkGray</code>  | 64, 64, 64    |
| light gray | <code>Color.lightGray</code> | 192, 192, 192 |
| green      | <code>Color.green</code>     | 0, 255, 0     |
| magenta    | <code>Color.magenta</code>   | 255, 0, 255   |
| orange     | <code>Color.orange</code>    | 255, 200, 0   |
| pink       | <code>Color.pink</code>      | 255, 175, 175 |
| red        | <code>Color.red</code>       | 255, 0, 0     |
| white      | <code>Color.white</code>     | 255, 255, 255 |
| yellow     | <code>Color.yellow</code>    | 255, 255, 0   |

**figure 2.21** Predefined colors in the `Color` class



Every graphics context has a current *foreground color* that is used whenever shapes or strings are drawn. Every surface that can be drawn on has a *background color*. The foreground color is set using the `setColor` method of the `Graphics` class, and the background color is set using the `setBackground` method of the component on which we are drawing, such as the applet.

Listing 2.15 shows an applet called `Snowman`. It uses various drawing and color methods to draw a winter scene featuring a snowman. Review the code carefully to note how each shape is drawn to create the overall picture.

### listing 2.15

```

//*****
//  Snowman.java      Author: Lewis/Loftus
//
//  Demonstrates basic drawing methods and the use of color.
//*****

import java.applet.Applet;
import java.awt.*;

public class Snowman extends Applet
{
    //-----
    //  Draws a snowman.
    //-----
    public void paint (Graphics page)
    {
        final int MID = 150;
        final int TOP = 50;

        setBackground (Color.cyan);

        page.setColor (Color.blue);
        page.fillRect (0, 175, 300, 50);  // ground

        page.setColor (Color.yellow);
        page.fillOval (-40, -40, 80, 80);  // sun

        page.setColor (Color.white);
    }
}
```

**listing**  
**2.15** continued

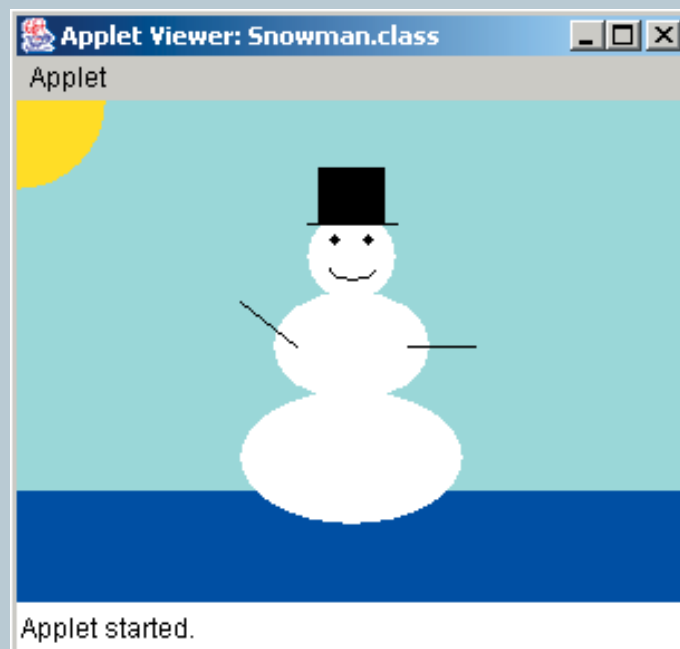
```
page.fillOval (MID-20, TOP, 40, 40);      // head
page.fillOval (MID-35, TOP+35, 70, 50);    // upper torso
page.fillOval (MID-50, TOP+80, 100, 60);   // lower torso

page.setColor (Color.black);
page.fillOval (MID-10, TOP+10, 5, 5);      // left eye
page.fillOval (MID+5, TOP+10, 5, 5);      // right eye

page.drawArc (MID-10, TOP+20, 20, 10, 190, 160); // smile

page.drawLine (MID-25, TOP+60, MID-50, TOP+40); // left arm
page.drawLine (MID+25, TOP+60, MID+55, TOP+60); // right arm


page.drawLine (MID-20, TOP+5, MID+20, TOP+5); // brim of hat
page.fillRect (MID-15, TOP-20, 30, 25);      // top of hat
}
```

**display**



Note that the snowman figure is based on two constant values called `MID` and `TOP`, which define the midpoint of the snowman (left to right) and the top of the snowman's head. The entire snowman figure is drawn relative to these values. Using constants like these makes it easier to create the snowman and to make modifications later. For example, to shift the snowman to the right or left in our picture, only one constant declaration would have to change.





### summary of key concepts

- The information we manage in a Java program is either represented as primitive data or as objects.
- An abstraction hides details. A good abstraction hides the right details at the right time so that we can manage complexity.
- A variable is a name for a memory location used to hold a value of a particular data type.
- A variable can store only one value of its declared type.
- Java is a strongly typed language. Each variable is associated with a specific type for the duration of its existence, and we cannot assign a value of one type to a variable of an incompatible type.
- Constants are similar to variables, but they hold a particular value for the duration of their existence.
- Java has two kinds of numeric values: integers and floating point. There are four integer data types (`byte`, `short`, `int`, and `long`) and two floating point data types (`float` and `double`).
- Many programming statements involve expressions. Expressions are combinations of one or more operands and the operators used to perform a calculation.
- Java follows a well-defined set of rules that govern the order in which operators will be evaluated in an expression. These rules form an operator precedence hierarchy.
- Avoid narrowing conversions because they can lose information.
- The `new` operator returns a reference to a newly created object.
- The Java standard class library is a useful set of classes that anyone can use when writing Java programs.
- A package is a Java language element used to group related classes under a common name.
- The `Keyboard` class is not part of the Java standard library. It is therefore not available on all Java development platforms.
- Applets are Java programs that are usually transported across a network and executed using a Web browser. Java applications are stand-alone programs that can be executed using the Java interpreter.
- Most shapes can be drawn filled (opaque) or unfilled (as an outline).

- ▶ A bounding rectangle is often used to define the position and size of curved shapes such as ovals.
- ▶ An arc is a segment of an oval; the segment begins at a specific start angle and extends for a distance specified by the arc angle.
- ▶ The `Color` class contains several common predefined colors.

## self-review questions

- 2.1 What are the primary concepts that support object-oriented programming?
- 2.2 Why is an object an example of abstraction?
- 2.3 What is primitive data? How are primitive data types different from objects?
- 2.4 What is a string literal?
- 2.5 What is the difference between the `print` and `println` methods?
- 2.6 What is a parameter?
- 2.7 What is an escape sequence? Give some examples.
- 2.8 What is a variable declaration?
- 2.9 How many values can be stored in an integer variable?
- 2.10 What are the four integer data types in Java? How are they different?
- 2.11 What is a character set?
- 2.12 What is operator precedence?
- 2.13 What is the result of `19%5` when evaluated in a Java expression? Explain.
- 2.14 What is the result of `13/4` when evaluated in a Java expression? Explain.
- 2.15 Why are widening conversions safer than narrowing conversions?
- 2.16 What does the `new` operator accomplish?
- 2.17 What is a Java package?
- 2.18 Why doesn't the `String` class have to be specifically imported into our programs?
- 2.19 What is a class method (also called a static method)?
- 2.20 What is the difference between a Java application and a Java applet?

## exercises

- 2.1 Explain the following programming statement in terms of objects and the services they provide:

```
System.out.println ("I gotta be me!");
```

- 2.2 What output is produced by the following code fragment? Explain.

```
System.out.print ("Here we go!");
System.out.println ("12345");
System.out.print ("Test this if you are not sure.");
System.out.print ("Another.");
System.out.println ();
System.out.println ("All done.");
```

- 2.3 What is wrong with the following program statement? How can it be fixed?

```
System.out.println ("To be or not to be, that
is the question.");
```

- 2.4 What output is produced by the following statement? Explain.

```
System.out.println ("50 plus 25 is " + 50 + 25);
```

- 2.5 What is the output produced by the following statement? Explain.

```
System.out.println ("He thrusts his fists\n\tagainst" +
" the post\nand still insists\n\tthe sees the \"ghost\"");
```

- 2.6 Given the following declarations, what result is stored in each of the listed assignment statements?

```
int iResult, num1 = 25, num2 = 40, num3 = 17, num4 = 5;
double fResult, val1 = 17.0, val2 = 12.78;
• iResult = num1 / num4;
• fResult = num1 / num4;
• iResult = num3 / num4;
• fResult = num3 / num4;
• fResult = val1 / num4;
• fResult = val1 / val2;
• iResult = num1 / num2;
• fResult = (double) num1 / num2;
• fResult = num1 / (double) num2;
```

```
    ▶ fResult = (double) (num1 / num2);
    ▶ iResult = (int) (val1 / num4);
    ▶ fResult = (int) (val1 / num4);
    ▶ fResult = (int) ((double) num1 / num2);
    ▶ iResult = num3 % num4;
    ▶ iResult = num 2 % num3;
    ▶ iResult = num3 % num2;
    ▶ iResult = num2 % num4;
```

- 2.7 For each of the following expressions, indicate the order in which the operators will be evaluated by writing a number beneath each operator.

```
    ▶ a - b - c - d
    ▶ a - b + c - d
    ▶ a + b / c / d
    ▶ a + b / c * d
    ▶ a / b * c * d
    ▶ a % b / c * d
    ▶ a % b % c % d
    ▶ a - (b - c) - d
    ▶ (a - (b - c)) - d
    ▶ a - ((b - c) - d)
    ▶ a % (b % c) * d * e
    ▶ a + (b - c) * d - e
    ▶ (a + b) * c + d * e
    ▶ (a + b) * (c / d) % e
```

- 2.8 What output is produced by the following code fragment?

```
String m1, m2, m3;
m1 = "Quest for the Holy Grail";
m2 = m1.toLowerCase();
m3 = m1 + " " + m2;
System.out.println (m3.replace('h', 'z'));
```

- 2.9 Write an assignment statement that computes the square root of the sum of num1 and num2 and assigns the result to num3.
- 2.10 Write a single statement that computes and prints the absolute value of total.

2.11 What is the effect of the following import statement?

```
import java.awt.*;
```

2.12 Assuming that a `Random` object has been created called `generator`, what is the range of the result of each of the following expressions?

```
generator.nextInt(20)
generator.nextInt(8) + 1
generator.nextInt(45) + 10
generator.nextInt(100) - 50
```

2.13 Write code to declare and instantiate an object of the `Random` class (call the object reference variable `rand`). Then write a list of expressions using the `nextInt` method that generates random numbers in the following specified ranges, including the endpoints. Use the version of the `nextInt` method that accepts a single integer parameter.

- 0 to 10
- 0 to 500
- 1 to 10
- 1 to 500
- 25 to 50
- -10 to 15

2.14 Write code statements to create a `DecimalFormat` object that will round a formatted value to 4 decimal places. Then write a statement that uses that object to print the value of `result`, properly formatted.

2.15 Explain the role played by the Web in the translation and execution of some Java programs.

2.16 Assuming you have a `Graphics` object called `page`, write a statement that will draw a line from point (20, 30) to point (50, 60).

2.17 Assuming you have a `Graphics` object called `page`, write a statement that will draw a rectangle with length 70 and width 35, such that its upper-left corner is at point (10, 15).

2.18 Assuming you have a `Graphics` object called `page`, write a statement that will draw a circle *centered* on point (50, 50) with a radius of 20 pixels.

- 2.19 The following lines of code draw the eyes of the snowman in the Snowman applet. The eyes seem centered on the face when drawn, but the first parameters of each call are not equally offset from the midpoint. Explain.

```
page.fillOval (MID-10, TOP+10, 5, 5);  
page.fillOval (MID+5, TOP+10, 5, 5);
```

## programming projects

- 2.1 Create a revised version of the `Lincoln` application from Chapter 1 such that quotes appear around the quotation.
- 2.2 Write an application that reads three integers and prints their average.
- 2.3 Write an application that reads two floating point numbers and prints their sum, difference, and product.
- 2.4 Create a revised version of the `TempConverter` application to convert from Fahrenheit to Celsius. Read the Fahrenheit temperature from the user.
- 2.5 Write an application that converts miles to kilometers. (One mile equals 1.60935 kilometers.) Read the miles value from the user as a floating point value.
- 2.6 Write an application that reads values representing a time duration in hours, minutes, and seconds, and then print the equivalent total number of seconds. (For example, 1 hour, 28 minutes, and 42 seconds is equivalent to 5322 seconds.)
- 2.7 Create a revised version of the previous project that reverses the computation. That is, read a value representing a number of seconds, then print the equivalent amount of time as a combination of hours, minutes, and seconds. (For example, 9999 seconds is equivalent to 2 hours, 46 minutes, and 39 seconds.)
- 2.8 Write an application that reads the  $(x, y)$  coordinates for two points. Compute the distance between the two points using the following formula:

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



- 2.9 Write an application that reads the radius of a sphere and prints its volume and surface area. Use the following formulas. Print the output to four decimal places.  $r$  represents the radius.

$$\text{Volume} = \frac{4}{3}\pi r^3$$

$$\text{Surface area} = 4\pi r^2$$



- 2.10 Write an application that reads the lengths of the sides of a triangle from the user. Compute the area of the triangle using Heron's formula (below), in which  $s$  represents half of the perimeter of the triangle, and  $a$ ,  $b$ , and  $c$  represent the lengths of the three sides. Print the area to three decimal places.

$$\text{Area} = \sqrt{s(s-a)(s-b)(s-c)}$$

- 2.11 Write an application that computes the number of miles per gallon (MPG) of gas for a trip. Accept as input a floating point number that represents the total amount of gas used. Also accept two integers representing the odometer readings at the start and end of the trip. Compute the number of kilometers per liter if you prefer.



- 2.12 Write an application that determines the value of the coins in a jar and prints the total in dollars and cents. Read integer values that represent the number of quarters, dimes, nickels, and pennies. Use a currency formatter to print the output.
- 2.13 Write an application that creates and prints a random phone number of the form xxx-xxx-xxxx. Include the dashes in the output. Do not let the first three digits contain an 8 or 9 (but don't be more restrictive than that), and make sure that the second set of three digits is not greater than 742. *Hint:* Think through the easiest way to construct the phone number. Each digit does not have to be determined separately.
- 2.14 Create a personal Web page using HTML (see Appendix J).
- 2.15 Create a revised version of the Snowman applet with the following modifications:
- ▀ Add two red buttons to the upper torso.
  - ▀ Make the snowman frown instead of smile.
  - ▀ Move the sun to the upper-right corner of the picture.
  - ▀ Display your name in the upper-left corner of the picture.
  - ▀ Shift the entire snowman 20 pixels to the right.



- 2.16 Write an applet that writes your name using the `drawString` method. Embed a link to your applet in an HTML document and view it using a Web browser.
- 2.17 Write an applet that draws a smiling face. Give the face a nose, ears, a mouth, and eyes with pupils.
- 2.18 Write an applet that draws the Big Dipper. Add some extra stars in the night sky.
- 2.19 Write an applet that draws some balloons tied to strings. Make the balloons various colors.
- 2.20 Write an applet that draws the Olympic logo. The circles in the logo should be colored, from left to right, blue, yellow, black, green, and red.
- 2.21 Write an applet that draws a house with a door (and doorknob), windows, and a chimney. Add some smoke coming out of the chimney and some clouds in the sky.
- 2.22 Write an applet that displays a business card of your own design. Include both graphics and text.
- 2.23 Write an applet that displays your name in shadow text by drawing your name in black, then drawing it again slightly offset in a lighter color.
- 2.24 Write an applet the shows a pie chart with eight equal slices, all colored differently.

## answers to self-review questions

- 2.1 The primary elements that support object-oriented programming are objects, classes, encapsulation, and inheritance. An object is defined by a class, which contains methods that define the operations on those objects (the services that they perform). Objects are encapsulated such that they store and manage their own data. Inheritance is a reuse technique in which one class can be derived from another.
- 2.2 An object is considered to be abstract because the details of the object are hidden from, and largely irrelevant to, the user of the object. Hidden details help us manage the complexity of software.

- 2.3 Primitive data are basic values such as numbers or characters. Objects are more complex entities that usually contain primitive data that help define them.
- 2.4 A string literal is a sequence of characters delimited by double quotes.
- 2.5 Both the `print` and `println` methods of the `System.out` object write a string of characters to the monitor screen. The difference is that, after printing the characters, the `println` performs a carriage return so that whatever's printed next appears on the next line. The `print` method allows subsequent output to appear on the same line.
- 2.6 A parameter is data that is passed into a method when it is invoked. The method usually uses that data to accomplish the service that it provides. For example, the parameter to the `println` method indicate what characters should be printed. The two numeric operands to the `Math.pow` method are the operands to the power function that is computed and returned.
- 2.7 An escape sequence is a series of characters that begins with the backslash (`\`) and that implies that the following characters should be treated in some special way. Examples: `\n` represents the newline character, `\t` represents the tab character, and `\"` represents the quotation character (as opposed to using it to terminate a string).
- 2.8 A variable declaration establishes the name of a variable and the type of data that it can contain. A declaration may also have an optional initialization, which gives the variable an initial value.
- 2.9 An integer variable can store only one value at a time. When a new value is assigned to it, the old one is overwritten and lost.
- 2.10 The four integer data types in Java are `byte`, `short`, `int`, and `long`. They differ in how much memory space is allocated for each and therefore how large a number they can hold.
- 2.11 A character set is a list of characters in a particular order. A character set defines the valid characters that a particular type of computer or programming language will support. Java uses the Unicode character set.
- 2.12 Operator precedence is the set of rules that dictates the order in which operators are evaluated in an expression.

- 2.13 The result of `19%5` in a Java expression is 4. The remainder operator `%` returns the remainder after dividing the second operand into the first. Five goes into 19 three times, with 4 left over.
- 2.14 The result of `13/4` in a Java expression is 3 (not 3.25). The result is an integer because both operands are integers. Therefore the `/` operator performs integer division, and the fractional part of the result is truncated.
- 2.15 A widening conversion tends to go from a small data value, in terms of the amount of space used to store it, to a larger one. A narrowing conversion does the opposite. Information is more likely to be lost in a narrowing conversion, which is why narrowing conversions are considered to be less safe than widening ones.
- 2.16 The `new` operator creates a new instance (an object) of the specified class. The constructor of the class is then invoked to help set up the newly created object.
- 2.17 A Java package is a collection of related classes. The Java standard class library is a group of packages that support common programming tasks.
- 2.18 The `String` class is part of the `java.lang` package, which is automatically imported into any Java program. Therefore, no separate import declaration is needed.
- 2.19 A class or static method can be invoked through the name of the class that contains it, such as `Math.abs`. If a method is not static, it can be executed only through an instance (an object) of the class.
- 2.20 A Java applet is a Java program that can be executed using a Web browser. Usually, the bytecode form of the Java applet is pulled across the Internet from another computer and executed locally. A Java application is a Java program that can stand on its own. It does not require a Web browser in order to execute.