

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

objects, which is basic to writing any program in an object-oriented language such as Java. We use objects to work with character strings, get information from the user, do difficult 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 work with color and draw shapes.

chapter objectives

- ▶ Define the difference between primitive data and objects.
- ▶ Declare and use variables.
- ▶ Perform mathematical computations.
- ▶ Create objects and use them.
- ▶ 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. Object-oriented programming ultimately requires a solid understanding of the following terms:

- object
- attribute
- method
- class
- encapsulation
- inheritance
- polymorphism

This section introduces these terms, giving an overview of object-oriented principles in order to show you the big picture.

key concept

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

An *object* is a basic part of a Java program. A software object often represents a real object in our problem area, such as a bank account. Every object has a *state* and a set of *behaviors*. By “state” we mean state of being—basic characteristics that currently define the object.

For example, part of a bank account’s state is its current balance. The behaviors of an object are the activities associated with the object. Behaviors associated with a bank account probably include the ability to make deposits and withdrawals. This book focuses on developing software by defining objects that interact with us and with each other.

In addition to objects, a Java program also manages primitive data. *Primitive data* include common values such as numbers and characters. The different kinds of primitive data are distinguished by their data type. A *data type* defines a set of values and operations—what we can do with 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.

In contrast to primitive data, objects usually represent something more complicated, and may contain primitive values as attributes. An object’s *attributes* are the values it stores internally, representing its state. These values may be primitive data or other objects. For example, an object that represents a bank account may store a primitive numeric value representing the account balance. It may contain other attributes, such as the name of the account owner.

As we discussed in Chapter 1, a *method* is a group of programming statements that is given a name so that we can use the method when we need it. When a method is called, its statements are executed. A set of methods is associated with an object. The methods of an object define its behaviors. To define the ability to make a deposit into a bank account, we define a method containing programming statements that will update the account balance accordingly.

An object is defined by a *class*, which is like the data type of the object. A class is the model or blueprint from which an object is created. It establishes the kind of data an object of that type will hold and defines the methods that represent the behavior of such objects or the operations that can be performed on them. However, a class is not an object any more than a blueprint is a house. In general, a class contains no space to store data. Each object has space for its own data, which is why each object can have its own state.

Once a class has been defined, objects can be created from that class. For example, once we define a class to represent the idea of a bank account, we can create objects that represent 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 only changes made to the state of the object should be done by that object's methods. We should design objects so that other objects cannot "reach in" and change its state. The methods defined in the bank account class would let us perform operations on individual bank account objects, such as withdrawing money from a particular account.

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*. We are taking advantage of the ways some kinds of classes are alike. One class can be used to create several new classes. These classes can then be used to create even more classes. This creates a family 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 family of classes for different types of bank accounts. Common characteristics are defined in high-level classes, and specific differences are defined in child, or *derived classes*.

Polymorphism is the idea that we can refer to objects of different but related types in the same way. It is a big help with designing solutions to problems that deal with multiple objects.

Classes, objects, encapsulation, and inheritance are the ideas that make up the world of object-oriented software. They are shown in Figure 2.1. We don't expect you to understand these ideas fully yet. This overview is

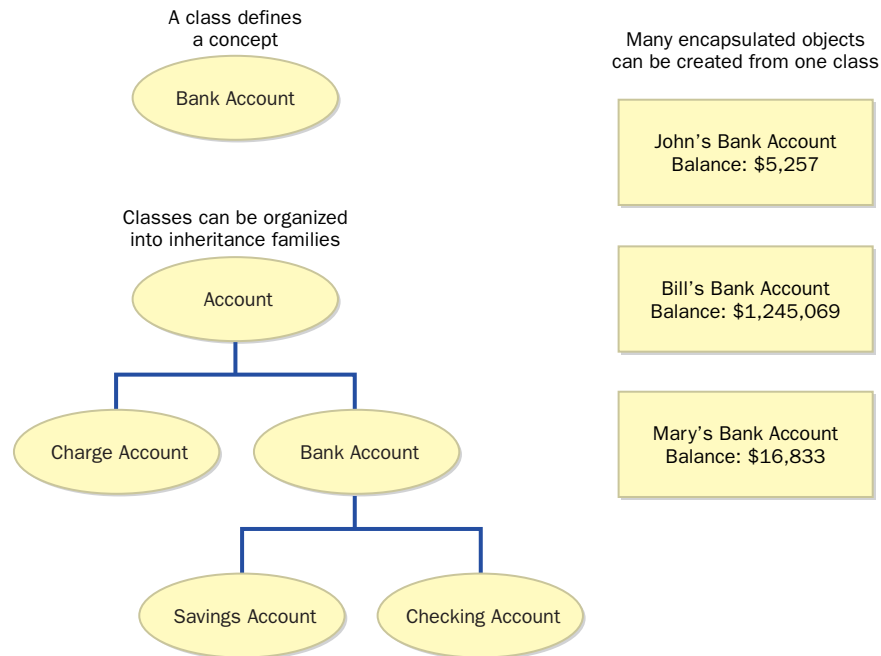


figure 2.1 Aspects of object-oriented software

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

2.1 using objects

In the `Lincoln` program in Chapter 1 (Listing 1.1), 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. The object's name is `out` and it is stored in the `System` class. We explore that relationship in more detail later in the text.

The `println` method is a behavior of the `System.out` object, or a service that the `System.out` object performs for us. Whenever we ask it to, 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 ask 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 methods 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 like `println`, but does not go 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.

listing 2.1

```
//*****  
//  Countdown.java          Author: Lewis/Loftus/Cocking  
//  
//  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.
```

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 what happens in the Countdown program. The Countdown class, with its `main` method, is shown invoking the `println` method of the `System.out` object.

The method name on the arrow in the diagram can be thought of as a message sent to the `System.out` object. 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 in this book, we will use these types of diagrams to explain object-oriented programs. The more complex our programs get, the more helpful such diagrams become.

abstraction

An object is an *abstraction*, meaning that the details of how it works don't matter to 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.

Sometimes it is important to hide or ignore certain details. People can manage 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, we can manage those pieces as one “chunk” in our minds. We don't deal with all of the details in the chunk, just the chunk itself. This way, we can deal with large quantities of information by organizing them into chunks. An object organizes information and lets us 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 need to know how a four-cycle combustion engine works in order to drive a

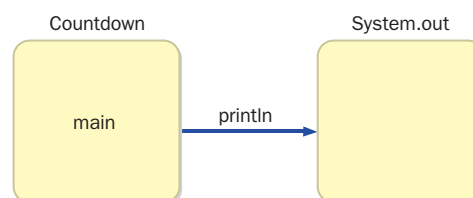


figure 2.2 Invoking a method

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 details, you'd probably never be able to operate something as complicated as a car.

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

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, but for now, we can use objects that have been defined for us already. Abstraction makes that possible.

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

key
concept

2.2 string literals

A character string is an object in Java, defined by the class `String`. Because strings are such an important part of computer programming, Java provides something called a *string literal*, which appears inside double quotation marks, 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 in double quotation marks cannot be split between two lines of code. One way to get around this

listing
2.2

```

//*****
// Facts.java      Author: Lewis/Loftus/Cocking
//
// 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

```

problem is to use the *string concatenation* operator, the plus sign (+). String concatenation adds one string to another. The string concatenation operation in the first `println` statement results in one large string that is passed to the method and printed.

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 characters that you can see, but it does move to the next line of output. In this case, the call to `println` passing in no parameters makes it “print” 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 how 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. 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 shows 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 together, 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 learn the rules that define the order in which operators get evaluated.

escape sequences

Because the double quotation mark (") is used in the Java language to indicate the beginning and end of a string, we need a special way to print a

listing
2.3

```

//*****
//  Addition.java      Author: Lewis/Loftus/Cocking
//
//  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

```

quotation mark. If we simply put it in a string ("\""), the compiler gets 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 quotation mark is treated as part of the string, not the end of it, so it can be printed as part of the output.

	Escape Sequence	Meaning
AP*→	<code>\n</code>	newline
AP*→	<code>\"</code>	double quote
AP*→	<code>\\</code>	backslash
	<code>\b</code>	backspace
	<code>\t</code>	tab
	<code>\r</code>	carriage return
	<code>\'</code>	single quote

figure 2.3 Java escape sequences

listing 2.4

```

//*****
//  Roses.java      Author: Lewis/Loftus/Cocking
//
//  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 in a program is represented by variables. Let's look at how we declare and use them in a program.

variables

key concept

A variable is a name for a memory location used to hold a value.

A *variable* is a name for a location in memory used to hold a data value. A variable declaration tells the compiler to reserve a portion of main memory space large enough to hold the value. It also tells the compiler what name to call the location.

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 a number, or an integer (`int`), value. The declaration also gives `keys` an initial value of 88. If you don't give an initial value for a variable, the value is undefined. Most Java compilers give errors or warnings if you try to use a variable before you've given it a value.

The `keys` variable, with its value, could be pictured as follows:

keys	88
------	----

In the `PianoKeys` program, the string passed to the `println` method is formed from three pieces. The first and third are string literals, and the second is the variable `keys`. When the program gets to the variable it uses the currently stored value. Because the value of `keys` is an integer, it is automatically converted to a string so it can be concatenated with the first string. Then the concatenated string is passed to `println` and printed.

Note that a variable declaration can have many variables of the same type on one line. Each variable on the line can be declared with or without an initializing value. For example, the following declaration declares two variables, `weight` and `total`, and gives `total` a beginning value of 0.

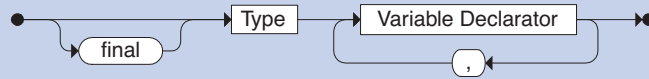
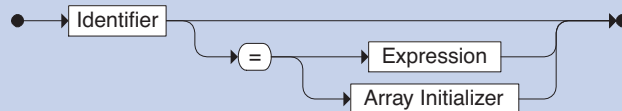
```
int weight, total = 0;
```

the assignment statement

Let's look at 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`.

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

```
sides = 10;
```

Local Variable Declaration**Variable Declarator**

A variable declaration is a Type followed by a list of variables. Each variable can be given a value computed from the Expression. If the `final` modifier comes before the declaration, the variables are declared as named constants whose values cannot be changed.

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/Cocking
//
//  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.
```

Basic Assignment

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

Examples:

```

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

**listing
2.6**

```

//*****
//  Geometry.java      Author: Lewis/Loftus/Cocking
//
//  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.
  
```

This is called an *assignment statement* because it gives or *assigns* a value to a variable. When the statement is 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.

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, as follows:

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

key
concept

After initialization: `sides` 7

After first assignment: `sides` 10

When a reference is made to a variable, such as when it is printed, the value of the variable is not changed. This is the nature of computer memory: Accessing (reading) data leaves the values in memory intact, but writing data replaces the old data with the new.

The Java language is *strongly typed*, meaning that we can't assign a value to a variable that is inconsistent with its declared type. Trying to combine incompatible types will cause an error when you attempt to compile the program. Therefore, the expression on the right-hand side of an assignment statement must have the same type as the variable on the left-hand side.

Java is a strongly typed language. Each variable has a declared type and we cannot assign a value of one type to a variable of another type.

key
concept

constants

Sometimes we use data that never changes—it 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 like variables except that they always have the same value. In Java, if you write reserved word `final` before a declaration, the identifier is made a constant. Uppercase letters are used for constant names to help us tell them apart from regular variables, and words are separated by the underscore character. For example, the constant describing the maximum occupancy of a theater could be:

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

The compiler will give you an error message if you try 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 accidental coding errors because the only place you can change their value is in the initial assignment.

**key
concept**

Constants are like variables, but they have the same value throughout the program.

There is a third good reason to use constants. If a constant is used throughout a program and its value needs to be changed, then you only have to change it in 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 you had used the literal 427 throughout the code, you would have had to find and change each use. 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 kinds of integers, two kinds of floating point numbers, a character data type, and a boolean data type. Everything else is represented using objects. Of the eight primitive types, three are a part of the AP* subset. We look at these three (`int`, `double`, and `boolean`) plus a fourth (`char`), in more detail. A discussion of the other primitive types can be found on the Web site.

integers and floating points

Java has two basic kinds of numeric values: integers, which have no fractional part, and floating points, which do. The primitive type `int` is an integer data type and `double` is a floating point data type. The numeric types are *signed*, meaning that both positive and negative values can be stored in them.

**key
concept**

Java has two kinds of numeric values: integers and floating point. The primitive type `int` is an integer data type and the type `double` is a floating point data type.

The `int` data type can be used to represent numbers in the range $-2,147,483,648$ to $2,147,483,647$. The `double` data type can represent numbers from approximately $-1.7\text{E}+308$ to $1.7\text{E}+308$ with 15 significant digits.

A *literal* is a specific data value used in a program. The numbers used in programs such as `Facts` (Listing 2.2) and `Addition` (Listing 2.3) and `PianoKeys` (Listing 2.5) are all *integer literals*. Java assumes all integer literals are of type `int`. Likewise, Java assumes that all *floating point literals*

are of type `double`. The following are examples of numeric variable declarations in Java:

```
int answer = 42;
int number1, number2;
double delta = 453.523311903;
```

The specific numbers used, 42 and 453.523311903, are literals.

booleans

A boolean value, defined in Java using the reserved word `boolean`, has only two values: `true` and `false`. A boolean variable usually tells us whether a condition is true, but it can also represent any situation that has two states, such as a lightbulb being on or off.

A boolean value cannot be changed to any other data type, nor can any other data type be changed to a boolean value. The words `true` and `false` are called *boolean literals* and cannot be used for anything else.

Here are some examples of boolean variable declarations in Java:

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

characters

Characters are another type of data. Note, however, that they are not part of the AP* subset. Individual characters can be treated as separate data items, or, 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 `';'`. Remember that *string literals* come in 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 are defined by a *character set*, which is just a list of characters in a particular order. Each programming language has its own particular character set. 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, which leaves enough 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 symbol that represents them. Yet they can be stored and used in the same way as any other character. Many control characters have special meanings to certain software applications.

As computers became more popular all over the world, users needed character sets that included other language alphabets. ASCII was changed to use eight bits per character, and the number of characters in the set doubled to 256. The new ASCII has many characters not used in English.

But even with 256 characters, the ASCII character set can't represent all the world's alphabets, especially the Asian alphabets, which have many thousands of characters, called ideograms. So 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 B 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 = ' ';
```

2.5 arithmetic expressions

An *expression* is a combination of operators and operands, like a mathematical expression. Expressions usually do a calculation such as addition or division. The answer does not have to be a number, but it often is. The operands might be literals, constants, variables, or other sources of data. The way expressions are used is basic to programming.

For now we will focus on mathematical expressions. The usual arithmetic operations include addition (+), subtraction (−), multiplication (*), and division (/). Java also has another arithmetic operation: the *remainder operator* (%). The remainder operator returns the remainder after dividing the second operand into the first. For example, $17\%4$ equals 1 because 17 divided by 4 equals 4 with one remaining. The remainder operator returns the 1. The sign of the result is the sign of the numerator. So because 17 and 4 are both positive, the remainder is positive. Likewise, $-20\%3$ equals −2, and $10\%-5$ equals 0.

Expressions are combinations of one or more operands and the operators used to perform a calculation.

key
concept

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 somewhat more complicated. 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, in the expression $10/4$ both 10 and 4 are integers so integer division is performed. 4 goes into 10 2.5 times but the fractional part (the .5) is discarded, so the answer is 2, an integer. On the other hand, the results of $10.0/4$ and $10/4.0$ and $10.0/4.0$ are all 2.5, because in these cases floating point division is performed.

operator precedence

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

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

The entire right-hand side of the assignment is solved, and then the answer is stored in the variable. But what is the answer? It is 11 if the addition is done first, or it is 18 if the division is done first. The order makes a big difference. In this case, the division is done before the addition, so the answer is 18. You should note that in this and other examples we have used literal values rather than variables to keep the expression simple. The order of operation is the same no matter what the operands are.

key
concept

Java follows a set of rules that govern the order in which operators will be evaluated in an expression. These rules are called an operator precedence hierarchy.

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

You can change the order, however, by using parentheses. For instance, if we really wanted the addition to be performed first, we could write the expression as follows:

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

Any expression in parentheses is done first. In complicated expressions, it is a good idea to use parentheses even when it is not strictly necessary.

Parentheses can be placed one inside another, and the innermost expressions are done first. Consider the following expression:

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

In this example, the result is 21. First, the subtraction is done, because it is inside the inner parentheses. Then, even though multiplication and division usually would be done left to right, the division is done next because of the outer parentheses. Finally, the multiplication is done.

After the arithmetic operations are complete, the answer is stored in the variable on the left-hand side of the assignment operator (=), in this case the variable `result`.

Figure 2.4 shows a table with the order of the arithmetic operators, parentheses, and the assignment operator. Appendix C 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 are for addition and subtraction, and the unary versions show positive and negative numbers. For example, –1 has a unary negation operator.

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 inside one another. 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 Celsius to Fahrenheit. Note that the operands to the division operation are `double` to ensure that the fractional part of the number is kept. The precedence rules dictate that the multiplication happens before the addition, which is what we want.

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 2.4 Precedence among some of the Java operators

listing 2.7

```
//*****
// TempConverter.java      Author: Lewis/Loftus/Cocking
//
// 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
```

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 `double` variable that holds the number 23.45 is converted to an `int` value. Because an `int` cannot store the fractional part of a number, some information would be lost in the conversion, and the number represented in the `int` 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. Converting from an `int` to a `double` is a widening conversion.

key concept

Avoid narrowing conversions because they can lose information.

Narrowing conversions are more likely to lose information than widening conversions are. Therefore, in general, they should be avoided. Converting from a `double` to an `int` is a narrowing conversion.

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

In Java, conversions can occur in three ways:

- assignment conversion
- arithmetic promotion
- casting

Assignment conversion happens when a value of one type is assigned to a variable of another type and the value is converted to the new type. Only widening conversions can be done this way. For example, if `money` is a `double` variable and `dollars` is an `int` variable, then the following assignment statement automatically converts the value in `dollars` to a `double`:

```
money = dollars;
```

So if `dollars` contains the value 25, after the assignment, `money` contains the value 25.0. However, if we try to go the other way around and assign `money` to `dollars`, the compiler will send us an error message telling us that we are trying to do a narrowing conversion that could lose information. If we really want to do this assignment, we have to do something called *casting*, which we'll get to in a minute.

Arithmetic promotion happens automatically when certain arithmetic operators need to change 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` becomes 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 done at all in a Java program, it can be done using a cast. A cast is a type name in parentheses, 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`, cutting off 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`.

We can use casting to round a floating point number to the nearest integer. Since casting to an `int` cuts off the fractional part of a number, we can add `0.5` to a positive floating point value, cast it to an `int`, and get the effect of rounding the value. If `number` is a `double` variable with a positive value, then the expression `(int)(number+0.5)` is the nearest integer.

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

```
result = (double) 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 by arithmetic promotion. Now the division operator will do floating point division. If the cast had not been included, the operation would have done integer division and cut the fraction off 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 enumerated types

Sometimes in a program we deal with values that come from a small, fixed set, such as the days of the week or the seasons of the year. Java allows us to create our own types to represent these values; these are called *enumerated types*. An enumerated type can be used as the type of a variable when the variable is declared. An enumerated type establishes all possible values of a variable of that type by listing, or enumerating, them. The values may be any valid identifiers.

For example, the following declaration defines an enumerated type called `Season` whose possible values are `winter`, `spring`, `summer`, and `fall`:

```
enum Season {winter, spring, summer, fall}
```

There is no limit to the number of values that you can list for an enumerated type. Once the type is defined, a variable can be declared of that type:

```
Season time;
```

key
concept

Enumerated types are type-safe, ensuring that invalid values will not be used.

The variable `time` is now restricted in the values it can take on. It can hold one of the four `Season` values, but nothing else. Java enumerated types are considered to be *type-safe*, meaning that any attempt to use a value other than one of the enumerated values will result in a compile-time error.

The values are accessed through the name of the type. For example:

```
time = Season.spring;
```

Enumerated types can be quite helpful in some situations. For example, suppose we wanted to represent the various letter grades a student could earn. We might declare the following enumerated type:

```
enum Grade {A, B, C, D, F}
```

Any initialized variable that holds a `Grade` is guaranteed to have one of those valid grades. That's better than using a simple character or string variable to represent the grade, which could take on any value.

Suppose we also wanted to represent plus and minus grades, such as A- and B+. We couldn't use A- or B+ as values, because they are not valid identifiers (the characters '-' and '+' cannot be part of an identifier in Java). However, the same values could be represented using the identifiers `Aminus`, `Bplus`, etc.

Listing 2.8 shows a program called `IceCream` that declares an enumerated type and uses it. Enumerated types are actually a special kind of class,

listing
2.8

```
//*****
//  IceCream.java      Author: Lewis/Loftus/Cocking
//
//  Demonstrates the use of enumerated types.
//*****

public class IceCream
{
    enum Flavor {vanilla, chocolate, strawberry, fudgeRipple, coffee,
                 rockyRoad, mintChocolateChip, cookieDough}

    //-----
    //  Creates and uses variables of the Flavor type.
    //-----

    public static void main (String[] args)
    {
        Flavor cone1, cone2, cone3;

        cone1 = Flavor.rockyRoad;
        cone2 = Flavor.chocolate;

        System.out.println ("cone1: " + cone1);
        System.out.println ("cone2: " + cone2);

        cone3 = cone1;

        System.out.println ("cone3: " + cone3);
    }
}
```

output

```
cone1: rockyRoad
cone2: chocolate
cone3: rockyRoad
```

and this means they may not be defined inside a method. They can be defined either at the class level (within the class but outside a method), as in this example, or at the outermost level.

2.7 creating objects

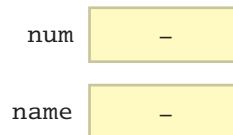
A variable can hold either a primitive value or a *reference to an object*. Like variables that hold primitive types, a variable that holds an object reference must be declared. A class is used to define an object, and the class name can be thought of as the type of an object. The declarations of object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

Consider the following two declarations:

```
int num;
String name;
```

The first declaration creates a variable that holds an integer value, as we've seen many times before. The second declaration creates a `String` variable that holds a reference to a `String` object. An object variable doesn't hold an object itself, it holds the address of an object.

Initially, the two variables declared above don't contain any data. We say they are *uninitialized*, which can be depicted as follows:



It is always important to make sure a variable is initialized before using it. For an object variable, that means we must make sure it refers to a valid object prior to using it. In most situations the compiler will issue an error if you attempt to use a variable before initializing it.

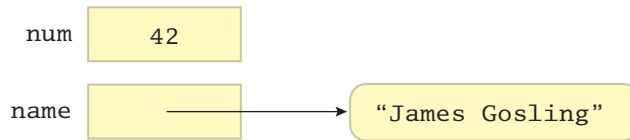
Note that, although we've declared a `String` reference variable, no `String` object actually exists yet. We create an object with the `new` operator, and this is called *instantiation*. An object is an *instance* of a particular class. The following two assignment statements give values to the two variables declared above:

```
num = 42;
name = new String("James Gosling");
```

key concept

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

After the `new` operator creates the object, a *constructor* helps set it up. A constructor has the same name as the class and is like a method. In this example, the parameter to the constructor is a string literal ("James Gosling"), which spells out the characters that the string object will hold. After these assignments are executed, the variables can be depicted as:



We can declare the object reference variable and create the object itself in 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 get 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 added right after the object reference and is followed by the method being invoked. For example, to invoke 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 we need the parentheses to show that `length` is a method. Some methods produce a value that is *returned*. The `length` method will 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`) stores the address where the object is stored in memory. We learn more about object references, instantiation, and constructors in later chapters.

the String class

Let's look at the `String` class in more detail. Strings in Java are objects represented by the `String` class. Figure 2.5 lists some of the more useful methods of the `String` class. The method headers 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. This is the type of information that will be returned, if anything. A return type of `void` means 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 or shortened, nor can any of its characters change. Thus we say that a `String` object is *immutable*. We can, however, create new `String` objects that have the new version of the original string's value.

	<code>String (String str)</code> Constructor: creates a new string object with the same characters as str.
	<code>char charAt (int index)</code> Returns the character at the specified index.
AP*→	<code>int compareTo (String str)</code> Returns a number indicating whether this string comes before (a negative return value), is equal to (a zero return value), or comes after (a positive return value), the string str.
	<code>String concat (String str)</code> Returns a new string made up of this string added to (concatenated with) str.
AP*→	<code>boolean equals (String str)</code> Returns true if this string contains the same characters as str (including upper or lowercase) and false if it does not.
	<code>boolean equalsIgnoreCase (String str)</code> Returns true if this string contains the same characters as str (ignoring upper and lowercase) and false if it does not.
AP*→	<code>int indexOf (String str)</code> Returns the position of the first character in the first occurrence of str in this string.
AP*→	<code>int length ()</code> Returns the number of characters in this string.
	<code>String replace (char oldChar, char newChar)</code> Returns a new string that is identical with this string except that every oldChar is replaced by newChar.
AP*→	<code>String substring (int offset, int endIndex)</code> Returns a new string that is a subset of this string starting at index offset and ending with the character at position endIndex-1.
AP*→	<code>String substring (int offset)</code> Returns a new string that starts at index offset and extends to the end of the string.
	<code>String toLowerCase ()</code> Returns a new string that is the same as this string except all uppercase letters are changed to lowercase.
	<code>String toUpperCase ()</code> Returns a new string that is the same as this string except all lowercase letters are changed to uppercase.

figure 2.5 Some methods of the String class

Notice that some of the String methods refer to the *index* of a particular character. The index is a character's position 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, 'e' is one, and so on.

Several `String` methods are used in the program called `StringMutation`, shown in Listing 2.9.

Figure 2.6 shows the `String` objects that are created in Listing 2.9, the `StringMutation` program. Compare this diagram to the program code and the output. Keep in mind this program creates five separate `String` objects using various methods of the `String` class.

Even though they are not primitive types, strings are so basic and so often used that Java defines string literals in 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, we don’t need the `new` operator and the call to the constructor. In most cases, we will use this simplified syntax.

wrapper classes

As we’ve been discussing, there are two categories of data in Java: primitive values and objects. Sometimes we’ll find ourselves in a situation where we’ve got primitive data but objects are required. For example, as we will see in later chapters, there are container objects that hold other objects, and they cannot hold primitive types. In cases like this we need to “wrap” a primitive value into an object.

All of the primitive types in Java have *wrapper classes*. These are classes that let you create objects representing primitive data. The `Integer` class

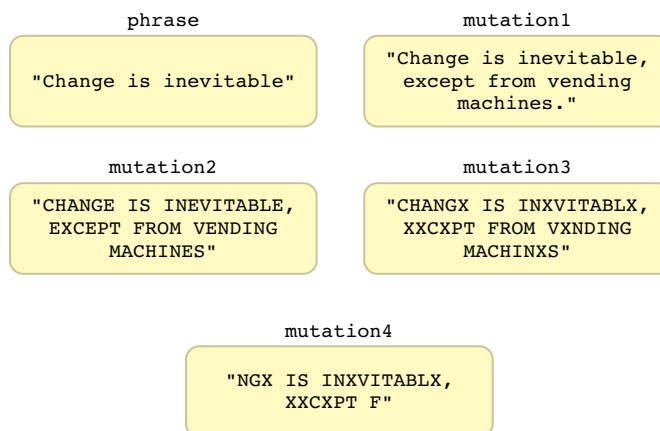


figure 2.6 The `String` objects created in the `StringMutation` program

Listing 2.9

```

//*****
//  StringMutation.java          Author: Lewis/Loftus/Cocking
//
//  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

```

**key
concept**

A wrapper class allows a primitive value to be used as an object.

wraps (represents) an `int` and the `Double` class wraps (represents) a `double`. We can create `Integer` and `Double` objects from primitive data by passing an `int` or `double` value to the constructor of

`Integer` or `Double`, respectively. For example, the following declaration creates an `Integer` object representing the integer 45:

```
Integer number = new Integer (45);
```

Once this statement is executed, the `number` object represents the integer 45 as an object. It can be used wherever an object is needed in a program rather than a primitive type.

Like strings, `Integer` and `Double` objects are immutable. Once an `Integer` or `Double` object is created, its value cannot be changed. Figures 2.7 and 2.8 list the methods on the `Integer` and `Double` classes that are part of the AP* subset.

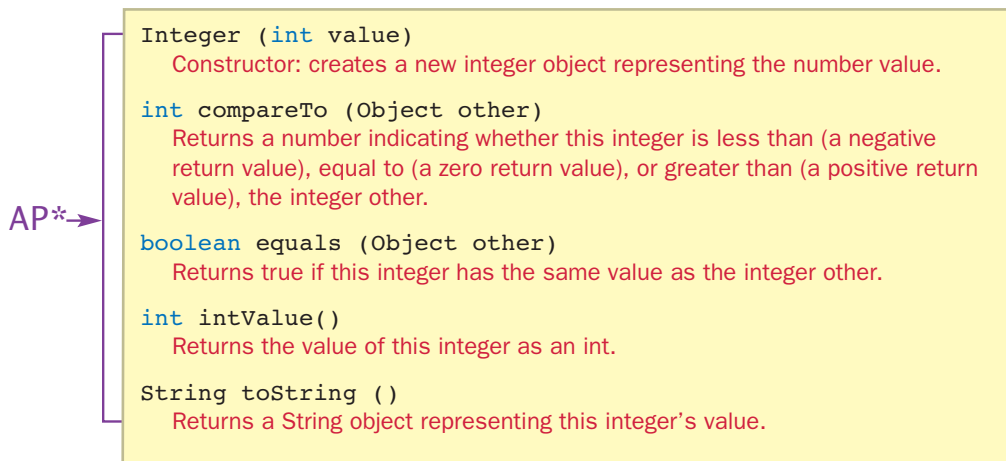


figure 2.7 Some methods of the `Integer` class

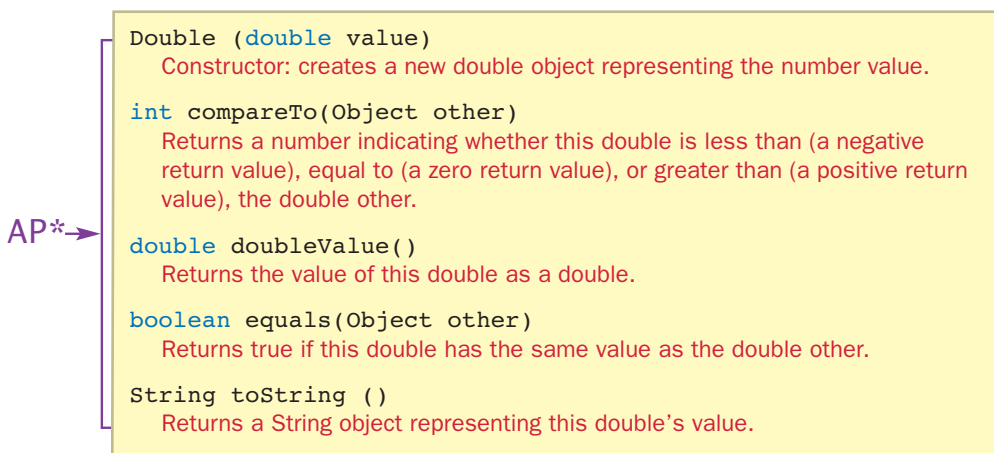


figure 2.8 Some methods of the `Double` class

autoboxing

key concept

Autoboxing provides automatic conversions between primitive values and corresponding wrapper objects.

Autoboxing is the automatic conversion between a primitive value and a corresponding wrapper object. For example, in the following code, an `int` value is assigned to an `Integer` object reference variable:

```
Integer obj1;
int num1 = 69;
obj1 = num1; // automatically creates an Integer object
```

The reverse conversion, called *unboxing*, also occurs automatically when needed. For example:

```
Integer obj2 = new Integer(69);
int num2;
num2 = obj2; // automatically extracts the int value
```

Assignments between primitive types and object types are generally incompatible. The ability to autobox occurs only between primitive types and corresponding wrapper classes. In any other case, attempting to assign a primitive value to an object reference variable, or vice versa, will cause a compile-time error. Autoboxing is not part of the AP* subset.

2.8 class libraries and packages

key concept

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

A *class library* is a set of classes that supports the development of programs. A compiler often comes with a class library. You can also get class libraries separately through third-party vendors. The classes in a class library have methods that are often valuable to a programmer because of their special functions. In fact, programmers often depend on the methods in a class library and begin to think of them as part of the language, even though technically, they are not in the language definition.

The `String` class, for instance, is not part of the Java language. It is part of the Java *standard class library*. 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 sets of related classes, which are sometimes called Java APIs, or *Application Programmer Interfaces*. 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 is a set of classes used in a graphical user interface (GUI). Sometimes the entire standard library is referred to as the Java API.

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 2.9 shows how the library is organized into packages.

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

key
concept

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

Figure 2.10 describes some of the packages that are part of the Java standard class library. These packages are available on any type of computer system that supports Java software development. Many of these packages are very sophisticated and are not used in the development of basic programs.

Many classes of the Java standard class library are discussed throughout this book. The classes that are part of the AP* subset are found in the `java.lang` and `java.util` packages. Appendix D serves as a general reference for all of the Java classes in the AP* subset.

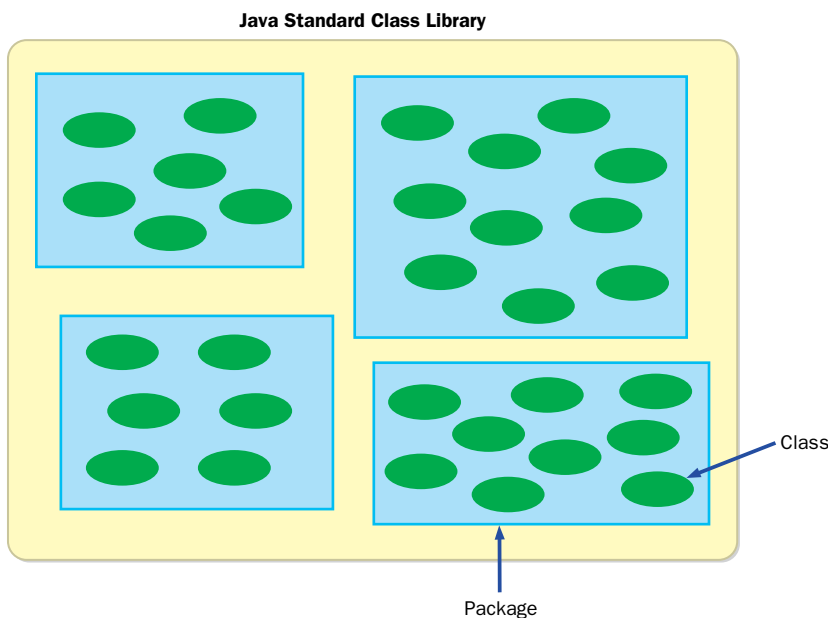


figure 2.9 Classes organized into packages in the Java standard class library

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 many kinds 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.
<code>java.net</code>	Communicate across a network.
<code>java.rmi</code>	Create programs that can be distributed across many 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 that extend the AWT capabilities.
<code>javax.xml.parsers</code>	Process XML documents; XML stands for eXtensible Markup Language.

figure 2.10 Some packages in the Java standard class library

the import declaration

We can use the classes of the package `java.lang` when we write a program. To use classes from any other package, however, we must either *fully qualify* the reference, or use an *import declaration*.

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, typing the whole package and class name every time it is needed quickly gets tiring. An import declaration makes this easier.

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 says that the `Random` class of the `java.util` package may be used in the program. Once you make this import declaration you only need 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 in the package might be used in the program. For example, the following declaration lets you use all classes in the `java.util` package in the program without having to type in the package name:

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

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 like basic extensions to the language. Therefore, any class in the `java.lang` package, such as `String`, can be used without an explicit `import` statement. It's as if all programs automatically contain the following statement:

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

Let's take a look at a few classes from the `java.util` package which we will find quite useful.

the `Random` class

You will often need random numbers when you are 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 decide 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 uses a *pseudorandom number generator*. A random number generator picks a number at random out of a range of values. A program that does this is called *pseudorandom*, because a program can't really pick a number randomly. A pseudorandom number generator might do a series of complicated calculations, starting with an initial *seed value*, and produce a number. Though they are technically not random (because they are calculated), the numbers produced by a pseudorandom number generator usually seem to be random, at least random enough for most situations. Figure 2.11 lists the methods of the `Random` class that are part of the AP* subset.

AP* →

```

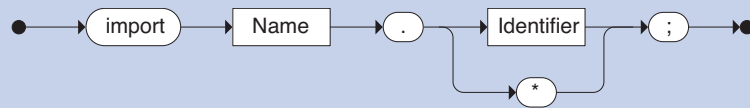
Random ( )
    Constructor: creates a new pseudorandom number generator.

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

int nextInt (int num)
    Returns a random number in the range 0 to num-1.

```

figure 2.11 Some methods of the Random class

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;

```

The `nextInt` method is called with a single integer value as a parameter. 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 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 `nextDouble` method of the `Random` class returns a `double` value that is greater than or equal to 0.0 and less than 1.0. If we want, we can use multiplication to scale the result, cast it into an `int` value to cut off the fractional part, then shift the range as we do with integers.

The program shown in Listing 2.10 produces several random numbers.

listing
2.10

```
//*****
// RandomNumbers.java      Author: Lewis/Loftus/Cocking
//
// 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;
        double num2;

        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.nextDouble();
        System.out.println ("A random double [between 0-1]: " + num2);

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

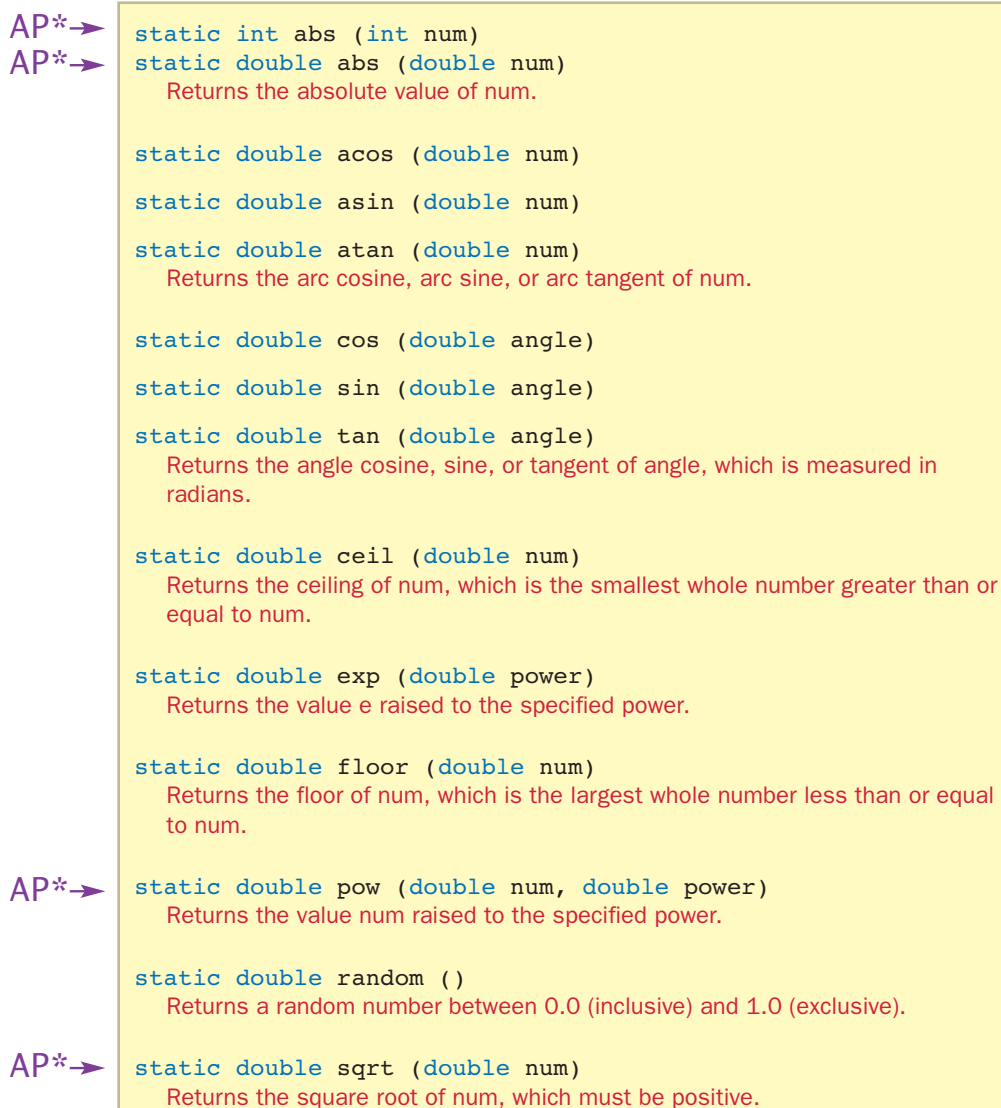
output

```
From 0 to 9: 6
From 1 to 10: 4
From 20 to 34: 30
From -10 to 9: -4
A random double [between 0-1]: 0.052495003
From 1 to 6: 6
```

the Math class

Some methods can be invoked through their class name, without having to instantiate an object of the class first. These are called *class methods* or *static methods*. Let's look at an example.

The Math class lets us do 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 2.12 lists several of its methods.



AP*→ `static int abs (int num)`
 AP*→ `static double abs (double 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.

AP*→ `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).

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

figure 2.12 Some methods of the Math class

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 in the next section.

2.9 interactive programs

It is often useful to design a program to read data from the user interactively during execution. That way, new results can be computed each time the program is run, depending on the data that is entered.

the Scanner class

The `Scanner` class, which is part of the standard Java class library in the `java.util` package, provides convenient methods for reading input values of various types. The input could come from various sources, including data typed interactively by the user or data stored in a file. The `Scanner` class can also be used to parse a character string into separate pieces. Figure 2.13 lists some of the methods provided by the `Scanner` class.

The `Scanner` class provides methods for reading input of various types from various sources.

key
concept

We must first create a `Scanner` object in order to invoke its methods. The following declaration creates a `Scanner` object that reads input from the keyboard:

```
Scanner scan = new Scanner (System.in);
```

This declaration creates a variable called `scan` that represents a `Scanner` object. The object itself is created by the `new` operator and a call to the constructor to set up the object. The `Scanner` constructor accepts a parameter that indicates the source of the input. The `System.in` object represents the *standard input stream*, which by default is the keyboard.

Unless specified otherwise, a `Scanner` object assumes that white space characters (space characters, tabs, and new lines) are used to separate the elements of the input, called *tokens*, from each other. These characters are called the input *delimiters*. The set of delimiters can be changed if the input tokens are separated by characters other than white space.

The `next` method of the `Scanner` class reads the next input token as a string and returns it. Therefore, if the input consisted of a series of words

```

Scanner (InputStream source)
Scanner (File source)
Scanner (String source)
    Constructors: sets up the new scanner to scan values from the specified source.

String next()
    Returns the next input token as a character string.

String nextLine()
    Returns all input remaining on the current line as a character string.

boolean nextBoolean()
byte nextByte()
double nextDouble()
float nextFloat()
int nextInt()
long nextLong()
short nextShort()
    Returns the next input token as the indicated type. Throws
    InputMismatchException if the next token is inconsistent with the type.

boolean hasNext()
    Returns true if the scanner has another token in its input.

Scanner useDelimiter (String pattern)
    Sets the scanner's delimiting pattern.

```

figure 2.13 Some methods of the Scanner class

separated by spaces, each call to `next` would return the next word. The `nextLine` method reads all of the input until the end of the line is found, and returns it as one string.

The program `Echo`, shown in Listing 2.11, simply reads a line of text typed by the user, stores it in a variable that holds a character string, then echoes it back to the screen.

A `Scanner` object processes the input one token at a time, based on the methods used to read the data and the delimiters used to separate the input values. Therefore, multiple values can be put on the same line of input or can be separated over multiple lines, as appropriate for the situation.

listing
2.11

```

//*****
//  Echo.java      Author: Lewis/Loftus/Cocking
//
//  Demonstrates the use of the nextLine method of the Scanner class
//  to read a string from the user.
//*****

import java.util.Scanner;

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

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

        message = scan.nextLine();

        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!"

```

Various `Scanner` methods such as `nextInt` and `nextDouble` are provided to read data of particular types. The `Quadratic` program, shown in Listing 2.12, uses the `Scanner` 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:

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

listing
2.12

```

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

import java.util.Scanner;

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

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

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

        System.out.print ("Enter the constant: ");
        c = scan.nextInt();

        // 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);

        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

```

In Chapter 5 we use the `Scanner` class to read input from a data file and modify the delimiters it uses to parse the data.

2.10 formatting output

The `NumberFormat` class and the `DecimalFormat` class are used to format information so that it looks right when printed or displayed. They are both part of the Java standard class library and are defined in the `java.text` package. These classes are not part of the AP* subset.

the `NumberFormat` class

The `NumberFormat` class lets you format numbers. You don't instantiate a `NumberFormat` object using the `new` operator. Instead, you ask for an object from one of the methods that you can invoke through the class itself. We haven't covered the reasons for this yet, but we will explain them later. Figure 2.14 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 money values. The `getPercentInstance` method returns an object that formats a percentage. The `format` method is called through a formatter object and returns a `String` that contains the formatted number.

The `Price` program shown in Listing 2.13 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 2.14 Some methods of the `NumberFormat` class

listing
2.13

```

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

import java.util.Scanner;
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;
        Scanner scan = new Scanner(System.in);

        System.out.print ("Enter the quantity: ");
        quantity = scan.nextInt();

        System.out.print ("Enter the unit price: ");
        unitPrice = scan.nextDouble();

        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));
    }
}

```

listing
2.13 continued**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 usual way using the `new` operator. Its constructor takes a string that represents the formatting pattern. We can then use the `format` method to format a particular value. Later on, if we want to change the formatting pattern, we can call the `applyPattern` method. Figure 2.15 describes these methods.

The pattern defined by the string that is passed to the `DecimalFormat` constructor gets pretty complicated. Different symbols are used to represent different formatting guidelines. The pattern defined by the string `"0.###"`, for example, tells us that at least one digit should be printed to the left of the decimal point and should be a zero if that part of the number is zero. It also indicates that the value to the right of the decimal point should be rounded to three digits. This pattern is used in the `CircleStats` program shown in Listing 2.14, which reads the radius of a circle from the user and computes its area and circumference. The final zero, such as in 78.540, is not printed.

```
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 the current pattern.

figure 2.15 Some methods of the `DecimalFormat` class

listing
2.14

```

//*****
//  CircleStats.java      Author: Lewis/Loftus/Cocking
//
//  Demonstrates the formatting of decimal values using the
//  DecimalFormat class.
//*****

import java.util.Scanner;
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;
        Scanner scan = new Scanner(System.in);

        System.out.print ("Enter the circle's radius: ");
        radius = scan.nextInt();

        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 `printf` method

In addition to `print` and `println`, the `System` class has another output method called `printf`, which allows the user to print a formatted string containing data values. The first parameter to the method represents the format string, and the remaining parameters specify the values that are inserted into the format string.

For example, the following line of code prints an ID number and a name:

```
System.out.printf ("ID: %5d\tName: %s", id, name);
```

The first parameter specifies the format of the output and includes literal characters that label the output values (`ID:` and `Name:`) as well as escape characters such as `\t`. The pattern `%5d` indicates that the corresponding numeric value (`id`) should be printed in a field of five characters. The pattern `%s` matches the string parameter `name`. The values of `id` and `name` are inserted into the string, producing a result such as:

```
ID: 24036      Name: Larry Flagelhopper
```

The `printf` method was added to Java to mirror a similar function used in programs written in the C programming language. It makes it easier for a programmer to translate an existing C program into Java. However, using the `printf` method is not a particularly clean object-oriented solution to the problem of formatting output, so we avoid its use in this book.

AP* case study

One of the things that you will want to study for the AP* Exam is the AP* Case Study. The AP* Case Study helps you learn by analyzing a relatively large program written by experienced programmers. Some of the questions on the AP* Exam will be about this AP* Case Study.

In special online sections for each chapter of this book, starting here with Chapter 2 and extending through Chapter 7, we tie in the AP* Case Study with the topics you learned in the chapter. These sections help you learn the AP* Case Study as well as providing an opportunity for you to see the material from the chapter in action.

To work with the AP* Case Study section for this chapter, go to www.aw.com/cssupport and look under author: Lewis/Loftus/Cocking.



2.11 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 embedded in 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 shown so far in this book have been Java applications.

The Web lets users send and receive different 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 just another type of media that can be exchanged across the Web.

Though Java applets are meant 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 so it can 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 that is part of a Web browser executes 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 a Java applet and 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. That means they do not have a `main` method where execution starts. For example, the `paint` method in an applet is automatically invoked by the applet. Consider the program in Listing 2.15, 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 tell which packages are used in the program. In this example, we need the `Applet` class, which is part of the `java.applet` package, and the graphics capabilities defined in the `java.awt` package.

A class that defines an applet extends the `Applet` class, as shown in the header line of the class declaration. This makes 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`.

listing 2.15

```

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

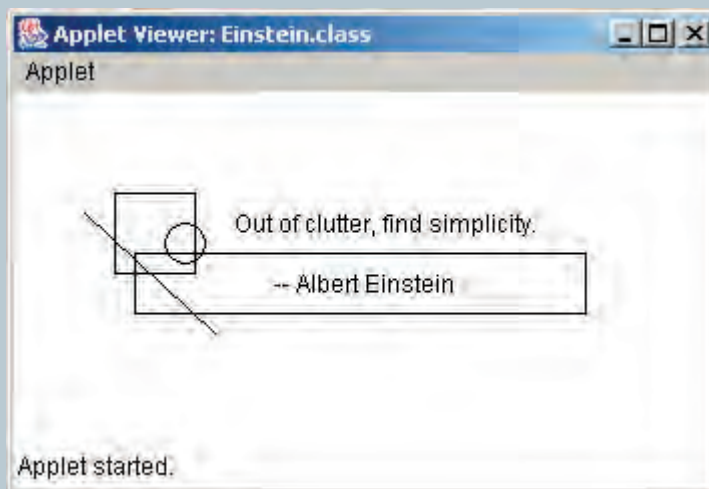
import java.applet.Applet;
import java.awt.*;

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



The `paint` method is one of several special applet methods. 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.

Note that the `paint` method accepts a `Graphics` object as a parameter. A `Graphics` object defines a particular *graphics context*, a part of the screen we can use. 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 graphics context.

A `Graphics` object lets us draw shapes using methods such as `drawRect`, `drawOval`, `drawLine`, and `drawString`. The parameters passed to the drawing methods list 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 travel over the Web and be executed by a browser, it must be referenced in a HyperText Markup Language (HTML) document. An HTML document contains *tags* that spell out 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:

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

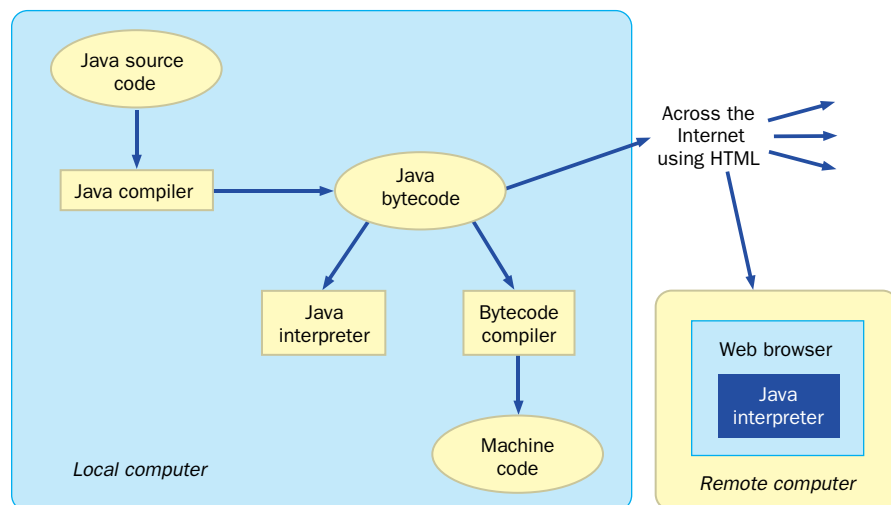


figure 2.16 The Java translation and execution process, including applets

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

There are other tags that can be used to reference an applet in an HTML file, including the `<object>` tag and the `<embed>` tag. The `<object>` tag is actually the tag that should be used, according to the World Wide Web Consortium (W3C). However, browser support for the `<object>` tag is not consistent. For now, the most reliable solution is the `<applet>` tag.

Note that the applet tag refers to the bytecode file of the `Einstein` applet, not to the source code file. Before an applet can travel over the Web, it must be compiled into its bytecode format. Then, as shown in Figure 2.16, the document can be loaded using a Web browser, which will automatically interpret and execute the applet.

2.12 drawing shapes

The Java standard class library provides many classes that let us use graphics. The `Graphics` class is the basic tool for presenting and using graphics.

the `Graphics` class

The `Graphics` class is defined in the `java.awt` package. Its methods let us draw shapes, including lines, rectangles, and ovals. Figure 2.17 lists some of the drawing methods of the `Graphics` class. Note that these methods also let us draw circles and squares, which are types of ovals and rectangles. We discuss more drawing methods of the `Graphics` class later in this book.

The methods of the `Graphics` class let us fill, or color in, a shape if we want to. An unfilled shape is only an outline and is transparent (you can see any underlying graphics). A filled shape is solid and covers any underlying graphics.

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

key
concept

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 the `drawOval` method draws an oval inside an imaginary rectangle, called the *bounding*

A bounding rectangle defines the position and size of curved shapes such as ovals.

key
concept

```

void drawArc (int x, int y, int width, int height, int
startAngle, int arcAngle)
    Paints part of an oval in the rectangle defined by x, y, width, and
    height. The oval starts at startAngle and continues 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 in 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)
    Draws a shape and fills it 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 2.17 Some methods of the Graphics class

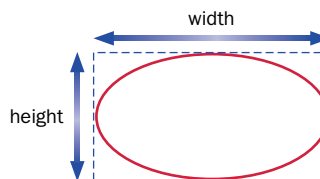


figure 2.18 An oval and its bounding rectangle

rectangle. Shapes with curves such as ovals are often drawn inside a rectangle as a way of giving their perimeters. Figure 2.18 shows a bounding rectangle for an oval.

An arc is a segment of an oval. To draw an arc, we describe the oval and the part of the oval we're interested in. The starting point of the arc is the *start angle* and the ending point is the *arc angle*. The arc angle does not say where the arc ends, but rather its range. The start angle and the arc angle are measured in degrees. The beginning of the start angle is an imaginary horizontal line passing through the center of the oval and can be referred to as 0° , as shown in Figure 2.19.

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

key
concept

the Color class

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 provides a basic set of predefined colors. Figure 2.20 lists the colors of the `Color` class.

A `Color` class contains several common colors.

key
concept

The `Color` class also contains methods you can use to define and manage many other colors. Recall from Chapter 1 that you can create colors using the RGB technique by mixing the primary colors: red, green, and blue.

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 2.16 shows an applet called `Snowman`. It uses drawing and color methods to draw a snowman. Look at the code carefully to see how each shape is drawn to create the picture.

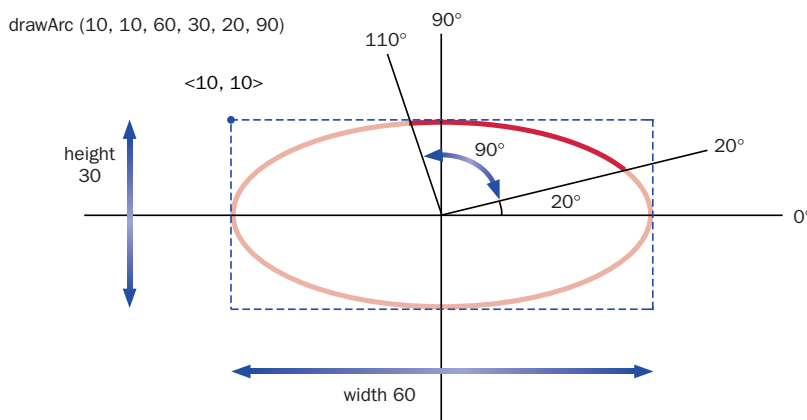


figure 2.19 An arc defined by an oval, a start angle, and an arc angle

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

figure 2.20 Predefined colors in the Color class

listing
2.16

```

//*****
//  Snowman.java      Author: Lewis/Loftus/Cocking
//
//  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
    }
}

```

listing
2.16 **continued**

```

page.setColor (Color.white);
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 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 is drawn relative to these values. Using constants like these makes it easier to create the snowman and to make changes later. For example, to shift the snowman to the right or left in our picture, we only have to change one constant declaration.

summary of key concepts

- The information we manage in a Java program is either primitive data or objects.
- An abstraction hides details. A good abstraction hides the right details at the right time.
- A variable is a name for a memory location used to hold a value.
- A variable can store only one value of its declared type.
- Java is a strongly typed language. Each variable has a specific type, and we cannot assign a value of one type to a variable of another type.
- Constants are like variables, but they have the same value throughout the program.
- Java has two kinds of numeric values: integers and floating point. The primitive type `int` is an integer data type and `double` is a floating point data type.
- Expressions are combinations of one or more operands and the operators used to perform a calculation.
- Java has rules that govern the order in which operators will be evaluated in an expression. These rules are called operator precedence rules.
- Avoid narrowing conversions because they can lose information.
- Enumerated types are type-safe, ensuring that invalid values will not be used.
- The `new` operator returns a reference to a newly created object.
- A wrapper class allows a primitive value to be used as an object.
- Autoboxing provides automatic conversions between primitive values and corresponding wrapper objects.
- 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 `Scanner` class provides methods for reading input of various types from various sources.
- Applets are Java programs that can travel across a network and be 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 in or left unfilled.
- 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 is a character set?
- 2.11 What is operator precedence?
- 2.12 What is the result of `19*5` when evaluated in a Java expression? Explain.
- 2.13 What is the result of `13/4` when evaluated in a Java expression? Explain.
- 2.14 Why are widening conversions safer than narrowing conversions?
- 2.15 What is an enumerated type?
- 2.16 What does the `new` operator do?
- 2.17 How can we represent a primitive value as an object?
- 2.18 What is a Java package?
- 2.19 Why doesn't the `String` class have to be imported into our programs?
- 2.20 What is a class method (also called a static method)?

2.21 What is the difference between a Java application and a Java applet?

multiple choice

2.1 What will be printed by the following statement?

```
System.out.println("Use a \\\"\\\"");
```

- a. Use a "\\\"\\\""
- b. "Use a "\\\""
- c. Use a "\\\""
- d. Use a "\\\"\\\""
- e. Use a "\\\""

2.2 Which keyword is used to declare a constant?

- a. `int`
- b. `double`
- c. `MAX`
- d. `constant`
- e. `final`

2.3 The expression "number" + 6 + 4 * 5 produces which of the following string literals?

- a. "number645"
- b. "number105"
- c. "number50"
- d. "number620"
- e. "number26"

2.4 Which of the following is a character literal?

- a. `b`
- b. `'b'`
- c. `"b"`
- d. `2`
- e. `2.0`

- 2.5 What is the result of the operation $30 \% 4$?
- a. 2
 - b. 3
 - c. 4
 - d. 7
 - e. 7.5
- 2.6 The expression $1 / 4$ is equal to which of the following?
- a. $1.0 / 4.0$
 - b. $(\text{double})2 / 8$
 - c. 0.25
 - d. $(\text{int})1.0 / 4.0$
 - e. $1 / (\text{int})4.0$
- 2.7 Which of the following instantiates a `String` object?
- a. `String word;`
 - b. `word = new String("the");`
 - c. `word.length();`
 - d. `word = name;`
 - e. `String word = name;`
- 2.8 Assuming `g` is an instance of the `Random` class, which statement will generate a random number between 10 and 100 inclusive?
- a. `num = g.nextInt(101);`
 - b. `num = 10 + g.nextInt(101);`
 - c. `num = 10 + g.nextInt(91);`
 - d. `num = 10 + g.nextInt(90);`
 - e. `num = g.nextInt(110) - 10;`
- 2.9 Which statement would we use to create an object from a class called `Thing`?
- a. `Thing something;`
 - b. `Thing something = Thing();`
 - c. `Thing something = new Thing;`
 - d. `Thing something = new Thing();`
 - e. `new Thing() = something;`

- 2.10 Suppose we have a variable `something` that is a reference to a `Thing` object. How would we call the method `doIt` on our `Thing` object?
- a. `doIt()`
 - b. `something.doIt()`
 - c. `doIt(something)`
 - d. `something/doIt`
 - e. `something(doIt)`

true/false

- 2.1 An object is an abstraction, meaning that the user doesn't need to know the details of how it works.
- 2.2 A string literal appears inside single quotation marks.
- 2.3 In order to include a double quotation mark (") or a backslash (\) in a string literal, we must use an escape sequence.
- 2.4 The operators `*`, `/`, and `%` have precedence over `+` and `-`.
- 2.5 Widening conversions can happen automatically, such as in the expression `1 + 2.5` where `1` is converted to a `double`.
- 2.6 In the declaration `int num = 2.4;` the `2.4` will automatically be converted to an `int` and `num` will get the value `2`.
- 2.7 In Java, `Integer` is a class, whereas `int` is a primitive type.
- 2.8 Assuming `generator` is an object of the `Random` class, the call `generator.nextInt(8)` will generate a random number between `0` and `7` inclusive.

short answer

- 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;
```

Example: `iResult = num2%num1;`

The result that gets stored is 15 because 40%25 equals 15 (25 goes into 40 once, with remainder 15).

- a. `iResult = num1 / num4;`
- b. `fResult = num1 / num4;`
- c. `iResult = num3 / num4;`
- d. `fResult = num3 / num4;`
- e. `fResult = val1 / num4;`
- f. `fResult = val1 / val2;`
- g. `iResult = num1 / num2;`
- h. `fResult = (double) num1 / num2;`
- i. `fResult = num1 / (double) num2;`
- j. `fResult = (double) (num1 / num2);`
- k. `iResult = (int) (val1 / num4);`
- l. `fResult = (int) (val1 / num4);`
- m. `fResult = (int) ((double) num1 / num2);`

```

n. iResult = num3 % num4;
o. iResult = num2 % num3;
p. iResult = num3 % num2;
q. 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.

Example: $a + b * c - d$

2 1 3

- a. $a - b - c - d$
- b. $a - b + c - d$
- c. $a + b / c / d$
- d. $a + b / c * d$
- e. $a / b * c * d$
- f. $a \% b / c * d$
- g. $a \% b \% c \% d$
- h. $a - (b - c) - d$
- i. $(a - (b - c)) - d$
- j. $a - ((b - c) - d)$
- k. $a \% (b \% c) * d * e$
- l. $a + (b - c) * d - e$
- m. $(a + b) * c + d * e$
- n. $(a + b) * (c / d) \% e$

- 2.8 Write code to create an enumerated type for the days of the week. Declare a variable of the type you created and set it equal to Sunday.
- 2.9 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'));

```

- 2.10 Write an assignment statement that computes the square root of the sum of num1 and num2 and assigns the result to num3.

- 2.11 Write a single statement that computes and prints the absolute value of `total`.
- 2.12 Assuming that a `Random` object called `generator` has been created, what is the range of the result of each of the following expressions?
- a. `generator.nextInt(20)`
 - b. `generator.nextInt(8) + 1`
 - c. `generator.nextInt(45) + 10`
 - d. `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 ranges, including the endpoints.
- a. 0 to 10
 - b. 0 to 500
 - c. 1 to 10
 - d. 1 to 500
 - e. 25 to 50
 - f. -10 to 15

programming projects

- 2.1 Create a new version of the `Lincoln` application from Chapter 1 with quotation marks around the quotation.
- 2.2 Write an application that reads three numbers 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 in hours, minutes, and seconds. Then print the same time in seconds. (For example, 1 hour, 28 minutes, and 42 seconds is equal to 5322 seconds.)
- 2.7 Create a new version of Programming Project 2.6 that works in reverse. That is, read a value representing a number of seconds, then print the same amount of time in hours, minutes, and seconds. (For example, 9999 seconds is equal 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 is half of the perimeter of the triangle, and a , b , and c are 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. The total amount of gas used should be a floating point number. Also accept two numbers representing the odometer readings at the start and end of the trip.
- 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 revised version of the Snowman applet (Listing 2.15) 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.

2.15 Write an applet that draws a smiling face. Give the face a nose, ears, a mouth, and eyes with pupils.

AP*-style multiple choice

2.1 Consider the following variable declarations.

```
int a = 5, b = 4;  
double x = 5.0, y = 4.0;
```

Which of the following expressions have the value 1?

- I. `(int) x / y`
- II. `a / b`
- III. `a % b`

- (A) II only
- (B) I and II only
- (C) I and III only
- (D) II and III only
- (E) I, II, and III

2.2 Consider a program that converts hours, minutes, and seconds into the total number of seconds. Assume the number of hours, minutes, and seconds have been read into the variables `hours`, `minutes`, and `seconds`, respectively, and that the variable `totalSeconds` has been properly declared. Which of the following correctly calculates the total number of seconds?

- (A) `totalSeconds = hours * minutes * 60 + minutes * 60 + seconds;`
- (B) `totalSeconds = hours * 360 + minutes * 60 + seconds;`
- (C) `totalSeconds = hours + minutes * 60 + seconds * 360;`
- (D) `totalSeconds = hours % 360 + minutes % 60 + seconds;`
- (E) `totalSeconds = hours / 360 + minutes / 60 + seconds;`

2.3 Consider the following variable declarations.

```
String s = "crunch";  
int a = 3, b = 1;
```

What is printed by the following statements?

```
System.out.print(s + a + b);  
System.out.println(b + a + s);
```

- (A) crunch44crunch
- (B) crunch413crunch
- (C) crunch314crunch
- (D) crunch3113crunch
- (E) Nothing is printed due to a runtime error.

answers to self-review questions

- 2.1 The main 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 store and manage their own data. Inheritance is a technique in which one class can be created from another.
- 2.2 An object is abstract because the details of the object are hidden from, and largely unimportant 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 and usually contain primitive data that help define them.
- 2.4 A string literal is a sequence of characters that appear in double quotation marks.
- 2.5 Both the `print` and `println` methods of the `System.out` object write a string of characters to the computer screen. The difference is that, after printing the characters, the `println` does a carriage return so that whatever's printed next appears on the next line. The `print` method lets new output appear on the same line.

- 2.6 A parameter is data that is passed into a method. The method usually uses that data. For example, the parameter to the `println` method is the string of characters to be printed. As another example, the two numeric parameters 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 (`\`). The characters that follow should be treated in some special way. Examples: `\n` represents the newline character and `\"` represents the quotation character (as opposed to using it to terminate a string).
- 2.8 A variable declaration gives the name of a variable and the type of data that it can contain. A declaration may also have an 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 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 recognize. Java uses the Unicode character set.
- 2.11 Operator precedence is the set of rules that dictates the order in which operators are evaluated in an expression.
- 2.12 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.
- 2.13 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 cut off.
- 2.14 A widening conversion does not cause information to be lost. 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.
- 2.15 An enumerated type is a user-defined type with a small, fixed set of possible values.
- 2.16 The `new` operator creates a new instance (an object) of a class. The constructor of the class helps set up the newly created object.

- 2.17 A wrapper class is defined in the Java standard class library for each primitive type. In situations where objects are called for, an object created from a wrapper class may suffice.
- 2.18 A Java package is a set of classes that have something in common. The Java standard class library is a group of packages that support common programming tasks.
- 2.19 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.
- 2.20 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.
- 2.21 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. A Java application is a Java program that can stand on its own. It does not need a Web browser in order to execute.