

10

Strings and Characters

Objectives

- To be able to create and manipulate nonmodifiable character string objects of class **String**.
- To be able to create and manipulate modifiable character string objects of class **StringBuffer**.
- To be able to create and manipulate objects of class **Character**.
- To be able to use a **StringTokenizer** object to break a **String** object into individual components called tokens.

*The chief defect of Henry King
Was chewing little bits of string.*
Hilaire Belloc

*Vigorous writing is concise. A sentence should contain no
unnecessary words, a paragraph no unnecessary sentences.*
William Strunk, Jr.

*I have made this letter longer than usual, because I lack the
time to make it short.*
Blaise Pascal

*The difference between the almost-right word & the right
word is really a large matter—it's the difference between the
lightning bug and the lightning.*
Mark Twain

Mum's the word.
Miguel de Cervantes, *Don Quixote de la Mancha*



Outline

- 10.1 Introduction
- 10.2 Fundamentals of Characters and Strings
- 10.3 `String` Constructors
- 10.4 `String` Methods `length`, `charAt` and `getChars`
- 10.5 Comparing Strings
- 10.6 `String` Method `hashCode`
- 10.7 Locating Characters and Substrings in Strings
- 10.8 Extracting Substrings from Strings
- 10.9 Concatenating Strings
- 10.10 Miscellaneous `String` Methods
- 10.11 Using `String` Method `valueOf`
- 10.12 `String` Method `intern`
- 10.13 `StringBuffer` Class
- 10.14 `StringBuffer` Constructors
- 10.15 `StringBuffer` Methods `length`, `capacity`, `setLength` and `ensureCapacity`
- 10.16 `StringBuffer` Methods `charAt`, `setCharAt`, `getChars` and `reverse`
- 10.17 `StringBuffer` `append` Methods
- 10.18 `StringBuffer` Insertion and Deletion Methods
- 10.19 `Character` Class Examples
- 10.20 Class `StringTokenizer`
- 10.21 Card Shuffling and Dealing Simulation
- 10.22 (Optional Case Study) Thinking About Objects: Event Handling

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises • Special Section: Advanced String Manipulation Exercises • Special Section: Challenging String Manipulation Projects

10.1 Introduction

In this chapter, we introduce Java's string and character-processing capabilities. The techniques discussed here are appropriate for validating program input, displaying information to users and other text-based manipulations. The techniques also are appropriate for developing text editors, word processors, page-layout software, computerized typesetting systems and other kinds of text-processing software. We have already presented several string-processing capabilities in the text. This chapter discusses in detail the capabilities of class ***String***, class ***StringBuffer*** and class ***Character*** from the `java.lang` package and class ***StringTokenizer*** from the `java.util` package. These classes provide the foundation for string and character manipulation in Java.

10.2 Fundamentals of Characters and Strings

Characters are the fundamental building blocks of Java source programs. Every program is composed of a sequence of characters that—when grouped together meaningfully—is interpreted by the computer as a series of instructions used to accomplish a task. A program might contain *character constants*. A character constant is an integer value represented as a character in single quotes. As we stated previously, the value of a character constant is the integer value of the character in the *Unicode character set*. For example, `'z'` represents the integer value of `z`, and `'\n'` represents the integer value of newline. See Appendix D for the integer equivalents of these characters.

A string is a series of characters treated as a single unit. A string may include letters, digits and various *special characters*, such as `+`, `-`, `*`, `/`, `$` and others. A string is an object of class **String**. *String literals* or *string constants* (often called *anonymous String objects*) are written as a sequence of characters in double quotation marks as follows:

<code>"John Q. Doe"</code>	(a name)
<code>"9999 Main Street"</code>	(a street address)
<code>"Waltham, Massachusetts"</code>	(a city and state)
<code>"(201) 555-1212"</code>	(a telephone number)

A **String** may be assigned in a declaration to a **String** reference. The declaration

```
String color = "blue";
```

initializes **String** reference `color` to refer to the anonymous **String** object `"blue"`.



Performance Tip 10.1

Java treats all anonymous **Strings** with the same contents as one anonymous **String** object that has many references. This conserves memory.

10.3 String Constructors

Class **String** provides nine constructors for initializing **String** objects in a variety of ways. Seven of the constructors are demonstrated in Fig. 10.1. All the constructors are used in the **StringConstructors** application's **main** method.

Line 25 instantiates a new **String** object and assigns it to reference `s1`, using class **String**'s default constructor. The new **String** object contains no characters (the *empty string*) and has a length of 0.

Line 26 instantiates a new **String** object and assigns it to reference `s2`, using class **String**'s copy constructor. The new **String** object contains a copy of the characters in the **String** object `s` that is passed as an argument to the constructor.

```
1 // Fig. 10.1: StringConstructors.java
2 // This program demonstrates the String class constructors.
3
4 // Java extension packages
5 import javax.swing.*;
6
```

Fig. 10.1 Demonstrating the **String** class constructors (part 1 of 2).

```

7  public class StringConstructors {
8
9      // test String constructors
10     public static void main( String args[] )
11     {
12         char charArray[] = { 'b', 'i', 'r', 't', 'h', ' ',
13                               'd', 'a', 'y' };
14         byte byteArray[] = { ( byte ) 'n', ( byte ) 'e',
15                               ( byte ) 'w', ( byte ) ' ', ( byte ) 'y',
16                               ( byte ) 'e', ( byte ) 'a', ( byte ) 'r' };
17
18         StringBuffer buffer;
19         String s, s1, s2, s3, s4, s5, s6, s7, output;
20
21         s = new String( "hello" );
22         buffer = new StringBuffer( "Welcome to Java Programming!" );
23
24         // use String constructors
25         s1 = new String();
26         s2 = new String( s );
27         s3 = new String( charArray );
28         s4 = new String( charArray, 6, 3 );
29         s5 = new String( byteArray, 4, 4 );
30         s6 = new String( byteArray );
31         s7 = new String( buffer );
32
33         // append Strings to output
34         output = "s1 = " + s1 + "\ns2 = " + s2 + "\ns3 = " + s3 +
35                 "\ns4 = " + s4 + "\ns5 = " + s5 + "\ns6 = " + s6 +
36                 "\ns7 = " + s7;
37
38         JOptionPane.showMessageDialog( null, output,
39                                       "Demonstrating String Class Constructors",
40                                       JOptionPane.INFORMATION_MESSAGE );
41
42         System.exit( 0 );
43     }
44 }
45 // end class StringConstructors

```

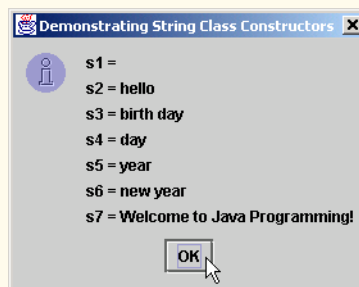


Fig. 10.1 Demonstrating the **String** class constructors (part 2 of 2).

**Software Engineering Observation 10.1**

*In most cases, it is not necessary to make a copy of an existing **String** object. **String** objects are immutable—their character contents cannot be changed after they are created. Also, if there are one or more references to a **String** object (or any object for that matter), the object cannot be reclaimed by the garbage collector. Thus, a **String** reference cannot be used to modify a **String** object or to delete a **String** object from memory as in other programming languages, such as C or C++.*

Line 27 instantiates a new **String** object and assigns it to reference **s3** using class **String**'s constructor that takes a character array as an argument. The new **String** object contains a copy of the characters in the array.

Line 28 instantiates a new **String** object and assigns it to reference **s4** using class **String**'s constructor that takes a **char** array and two integers as arguments. The second argument specifies the starting position (the **offset**) from which characters in the array are copied. The third argument specifies the number of characters (the **count**) to be copied from the array. The new **String** object contains a copy of the specified characters in the array. If the **offset** or the **count** specified as arguments result in accessing an element outside the bounds of the character array, a **StringIndexOutOfBoundsException** is thrown. We discuss exceptions in detail in Chapter 14.

Line 29 instantiates a new **String** object and assigns it to reference **s5** using class **String**'s constructor that receives a **byte** array and two integers as arguments. The second and third arguments specify the **offset** and **count**, respectively. The new **String** object contains a copy of the specified **bytes** in the array. If the **offset** or the **count** specified as arguments result in accessing an element outside the bounds of the character array, a **StringIndexOutOfBoundsException** is thrown.

Line 30 instantiates a new **String** object and assigns it to reference **s6** using class **String**'s constructor that takes a **byte** array as an argument. The new **String** object contains a copy of the bytes in the array.

Line 31 instantiates a new **String** object and assigns it to reference **s7** using class **String**'s constructor that receives a **StringBuffer** as an argument. A **StringBuffer** is a dynamically resizable and modifiable string. The new **String** object contains a copy of the characters in the **StringBuffer**. Line 22 creates a new object of class **StringBuffer** using the constructor that receives a **String** argument (in this case "Welcome to Java Programming") and assign the new object to reference **buffer**. We discuss **StringBuffers** in detail later in this chapter. The screen capture for the program displays the contents of each **String**.

10.4 String Methods length, charAt and getChars

The application of Fig. 10.2 presents **String** methods **length**, **charAt** and **getChars**, which determine the length of a **String**, get the character at a specific location in a **String** and get the entire set of characters in a **String**, respectively.

Line 28 uses **String** method **length** to determine the number of characters in **String s1**. Like arrays, **Strings** always know their own size. However, unlike arrays, **Strings** do not have a **length** instance variable that specifies the number of elements in a **String**.

```
1 // Fig. 10.2: StringMiscellaneous.java
2 // This program demonstrates the length, charAt and getChars
3 // methods of the String class.
4 //
5 // Note: Method getChars requires a starting point
6 // and ending point in the String. The starting point is the
7 // actual subscript from which copying starts. The ending point
8 // is one past the subscript at which the copying ends.
9
10 // Java extension packages
11 import javax.swing.*;
12
13 public class StringMiscellaneous {
14
15     // test miscellaneous String methods
16     public static void main( String args[] )
17     {
18         String s1, output;
19         char charArray[];
20
21         s1 = new String( "hello there" );
22         charArray = new char[ 5 ];
23
24         // output the string
25         output = "s1: " + s1;
26
27         // test length method
28         output += "\nLength of s1: " + s1.length();
29
30         // loop through characters in s1 and display reversed
31         output += "\nThe string reversed is: ";
32
33         for ( int count = s1.length() - 1; count >= 0; count-- )
34             output += s1.charAt( count ) + " ";
35
36         // copy characters from string into char array
37         s1.getChars( 0, 5, charArray, 0 );
38         output += "\nThe character array is: ";
39
40         for ( int count = 0; count < charArray.length; count++ )
41             output += charArray[ count ];
42
43         JOptionPane.showMessageDialog( null, output,
44             "Demonstrating String Class Constructors",
45             JOptionPane.INFORMATION_MESSAGE );
46
47         System.exit( 0 );
48     }
49
50 } // end class StringMiscellaneous
```

Fig. 10.2 The **String** class character manipulation methods (part 1 of 2).

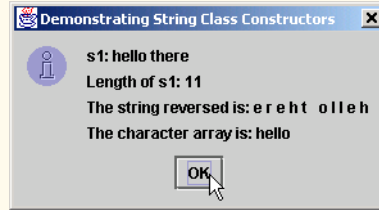


Fig. 10.2 The **String** class character manipulation methods (part 2 of 2).



Common Programming Error 10.1

Attempting to determine the length of a **String** via an instance variable called **length** (e.g., **s1.length**) is a syntax error. The **String** method **length** must be used. (e.g., **s1.length()**).

The **for** structure at lines 33–34 appends to **output** the characters of the **String** **s1** in reverse order. The **String** method **charAt** returns the character at a specific position in the **String**. Method **charAt** receives an integer argument that is used as the *position number* (or *index*) and returns the character at that position. Like arrays, the first element of a **String** is considered to be at position 0.



Common Programming Error 10.2

Attempting to access a character that is outside the bounds of a **String** (i.e., an index less than 0 or an index greater than or equal to the **String**'s length) results in a **StringIndexOutOfBoundsException**.

Line 37 uses **String** method **getChars** to copy the characters of a **String** into a character array. The first argument is the starting index from which characters are copied in the **String**. The second argument is the index that is one past the last character to be copied from the **String**. The third argument is the character array into which the characters are copied. The last argument is the starting index where the copied characters are placed in the character array. Next, the **char** array contents are appended one character at a time to **String output** with the **for** structure at lines 40–41 for display purposes.

10.5 Comparing Strings

Java provides a variety of methods for comparing **String** objects; these are demonstrated in the next two examples. To understand just what it means for one string to be “greater than” or “less than” another string, consider the process of alphabetizing a series of last names. The reader would, no doubt, place “Jones” before “Smith” because the first letter of “Jones” comes before the first letter of “Smith” in the alphabet. But the alphabet is more than just a list of 26 letters—it is an ordered list of characters. Each letter occurs in a specific position within the list. “Z” is more than just a letter of the alphabet; “Z” is specifically the twenty-sixth letter of the alphabet.

How does the computer know that one letter comes before another? All characters are represented inside the computer as numeric codes (see Appendix D). When the computer compares two strings, it actually compares the numeric codes of the characters in the strings.

Figure 10.3 demonstrates the **String** methods *equals*, *equalsIgnoreCase*, *compareTo* and *regionMatches* and demonstrates using the equality operator `==` to compare **String** objects.

```

1  // Fig. 10.3: StringCompare.java
2  // This program demonstrates the methods equals,
3  // equalsIgnoreCase, compareTo, and regionMatches
4  // of the String class.
5
6  // Java extension packages
7  import javax.swing.JOptionPane;
8
9  public class StringCompare {
10
11     // test String class comparison methods
12     public static void main( String args[] )
13     {
14         String s1, s2, s3, s4, output;
15
16         s1 = new String( "hello" );
17         s2 = new String( "good bye" );
18         s3 = new String( "Happy Birthday" );
19         s4 = new String( "happy birthday" );
20
21         output = "s1 = " + s1 + "\ns2 = " + s2 +
22                "\ns3 = " + s3 + "\ns4 = " + s4 + "\n\n";
23
24         // test for equality
25         if ( s1.equals( "hello" ) )
26             output += "s1 equals \"hello\"\n";
27         else
28             output += "s1 does not equal \"hello\"\n";
29
30         // test for equality with ==
31         if ( s1 == "hello" )
32             output += "s1 equals \"hello\"\n";
33         else
34             output += "s1 does not equal \"hello\"\n";
35
36         // test for equality (ignore case)
37         if ( s3.equalsIgnoreCase( s4 ) )
38             output += "s3 equals s4\n";
39         else
40             output += "s3 does not equal s4\n";
41
42         // test compareTo
43         output +=
44             "\ns1.compareTo( s2 ) is " + s1.compareTo( s2 ) +
45             "\ns2.compareTo( s1 ) is " + s2.compareTo( s1 ) +
46             "\ns1.compareTo( s1 ) is " + s1.compareTo( s1 ) +
47             "\ns3.compareTo( s4 ) is " + s3.compareTo( s4 ) +
48             "\ns4.compareTo( s3 ) is " + s4.compareTo( s3 ) +
49             "\n\n";

```

Fig. 10.3 Demonstrating **String** comparisons (part 1 of 2).

```

50
51     // test regionMatches (case sensitive)
52     if ( s3.regionMatches( 0, s4, 0, 5 ) )
53         output += "First 5 characters of s3 and s4 match\n";
54     else
55         output +=
56             "First 5 characters of s3 and s4 do not match\n";
57
58     // test regionMatches (ignore case)
59     if ( s3.regionMatches( true, 0, s4, 0, 5 ) )
60         output += "First 5 characters of s3 and s4 match\n";
61     else
62         output +=
63             "First 5 characters of s3 and s4 do not match\n";
64
65     JOptionPane.showMessageDialog( null, output,
66         "Demonstrating String Class Constructors",
67         JOptionPane.INFORMATION_MESSAGE );
68
69     System.exit( 0 );
70 }
71
72 } // end class StringCompare

```

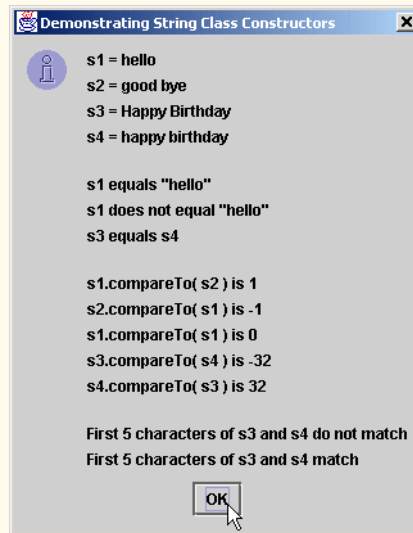


Fig. 10.3 Demonstrating **String** comparisons (part 2 of 2).

The condition in the **if** structure at line 25 uses method **equals** to compare **String** **s1** and anonymous **String** **"hello"** for equality. Method **equals** (inherited into **String** from its superclass **Object**) tests any two objects for equality—the contents of the two objects are identical. The method returns **true** if the objects are equal and **false** otherwise. The preceding condition is **true** because **String** **s1** was initialized with a copy of the anonymous **String** **"hello"**. Method **equals** uses a *lexicographical comparison*—the integer Unicode values that represent each character in each **String** are

compared. Thus, if the **String** "hello" is compared with the **String** "HELLO", the result is **false**, because the integer representation of a lowercase letter is different from that of the corresponding uppercase letter.

The condition in the **if** structure at line 31 uses the equality operator **==** to compare **String** **s1** for equality with the anonymous **String** "hello". *Operator **==** has different functionality when it is used to compare references and when it is used to compare values of primitive data types.* When primitive data type values are compared with **==**, the result is **true** if both values are identical. When references are compared with **==**, the result is **true** if both references *refer to the same object in memory*. To compare the actual contents (or state information) of objects for equality, methods (such as **equals**) must be invoked. The preceding condition evaluates to **false** in this program because the reference **s1** was initialized with the statement

```
s1 = new String( "hello" );
```

which creates a new **String** object with a copy of anonymous **String** "hello" and assigns the new object to reference **s1**. If **s1** had been initialized with the statement

```
s1 = "hello";
```

which directly assigns the anonymous **String** "hello" to the reference **s1**, the condition would be **true**. Remember that Java treats all anonymous **String** objects with the same contents as one anonymous **String** object that has many references. Thus, lines 16, 25 and 31 all refer to the same anonymous **String** object "hello" in memory.



Common Programming Error 10.3

*Comparing references with **==** can lead to logic errors, because **==** compares the references to determine whether they refer to the same object, not whether two objects have the same contents. When two identical (but separate) objects are compared with **==**, the result will be **false**. When comparing objects to determine whether they have the same contents, use method **equals**.*

If you are sorting **Strings**, you may compare them for equality with method **equalsIgnoreCase**, which ignores the case of the letters in each **String** when performing the comparison. Thus, the **String** "hello" and the **String** "HELLO" compare as equal. The **if** structure at line 37 uses **String** method **equalsIgnoreCase** to compare **String** **s3**—Happy Birthday—for equality with **String** **s4**—happy birthday. The result of this comparison is **true**, because the comparison ignores case sensitivity.

Lines 44–48 use **String** method **compareTo** to compare **String** objects. For example, line 44 compares **String** **s1** to **String** **s2**. Method **compareTo** returns 0 if the **Strings** are equal, a negative number if the **String** that invokes **compareTo** is less than the **String** that is passed as an argument and a positive number if the **String** that invokes **compareTo** is greater than the **String** that is passed as an argument. Method **compareTo** uses a *lexicographical comparison*—it compares the numeric values of corresponding characters in each **String**.

The condition in the **if** structure at line 52 uses **String** method **regionMatches** to compare portions of two **String** objects for equality. The first argument is the starting index in the **String** that invokes the method. The second argument is a comparison

String. The third argument is the starting index in the comparison **String**. The last argument is the number of characters to compare between the two **Strings**. The method returns **true** only if the specified number of characters are lexicographically equal.

Finally, the condition in the **if** structure at line 59 uses a second version of **String** method **regionMatches** to compare portions of two **String** objects for equality. If the first argument is **true**, the method ignores the case of the characters being compared. The remaining arguments are identical to those described for the for-argument **regionMatches** method.

The second example of this section (Fig. 10.4) demonstrates the **startsWith** and **endsWith** methods of class **String**. Application **StringStartEnd**'s **main** method defines an array of **Strings** called **strings** containing "started", "starting", "ended" and "ending". The remainder of method **main** consists of three **for** structures that test the elements of the array to determine whether they start with or end with a particular set of characters.

```

1  // Fig. 10.4: StringStartEnd.java
2  // This program demonstrates the methods startsWith and
3  // endsWith of the String class.
4
5  // Java extension packages
6  import javax.swing.*;
7
8  public class StringStartEnd {
9
10     // test String comparison methods for beginning and end
11     // of a String
12     public static void main( String args[] )
13     {
14         String strings[] =
15             { "started", "starting", "ended", "ending" };
16         String output = "";
17
18         // test method startsWith
19         for ( int count = 0; count < strings.length; count++ )
20
21             if ( strings[ count ].startsWith( "st" ) )
22                 output += "\"" + strings[ count ] +
23                     "\" starts with \"st\"\\n";
24
25         output += "\\n";
26
27         // test method startsWith starting from position
28         // 2 of the string
29         for ( int count = 0; count < strings.length; count++ )
30
31             if ( strings[ count ].startsWith( "art", 2 ) )
32                 output += "\"" + strings[ count ] +
33                     "\" starts with \"art\" at position 2\\n";
34
35         output += "\\n";

```

Fig. 10.4 **String** class **startsWith** and **endsWith** methods (part 1 of 2).

```

36
37     // test method endsWith
38     for ( int count = 0; count < strings.length; count++ )
39
40         if ( strings[ count ].endsWith( "ed" ) )
41             output += "\"" + strings[ count ] +
42                 "\" ends with \"ed\"\\n\";
43
44     JOptionPane.showMessageDialog( null, output,
45     "Demonstrating String Class Comparisons",
46     JOptionPane.INFORMATION_MESSAGE );
47
48     System.exit( 0 );
49 }
50
51 } // end class StringStartEnd

```

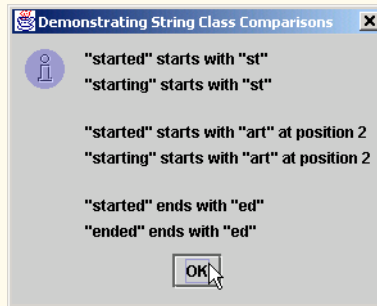


Fig. 10.4 String class `startsWith` and `endsWith` methods (part 2 of 2).

The first **for** structure (lines 19–23) uses the version of method `startsWith` that takes a **String** argument. The condition in the **if** structure (line 21) determines whether the **String** at location **count** of the array starts with the characters "st". If so, the method returns **true** and the program appends `strings[ count ]` to **output** for display purposes.

The second **for** structure (lines 29–33) uses the version of method `startsWith` that takes a **String** and an integer as arguments. The integer argument specifies the index at which the comparison should begin in the **String**. The condition in the **if** structure (line 31) determines whether the **String** at location **count** of the array starts with the characters "art" beginning with the character at index 2 in each **String**. If so, the method returns **true** and the program appends `strings[ count ]` to **output** for display purposes.

The third **for** structure (line 38–42) uses method `endsWith`, which takes a **String** argument. The condition in the **if** structure (40) determines whether the **String** at location **count** of the array ends with the characters "ed". If so, the method returns **true** and the program appends `strings[ count ]` to **output** for display purposes.

10.6 String Method `hashCode`

Often, it is necessary to store **Strings** and other data types in a manner that allows the information to be found quickly. One of the best ways to store information for fast lookup

is a *hash table*. A hash table stores information using a special calculation on the object to be stored that produces a *hash code*. The hash code is used to choose the location in the table at which to store the object. When the information needs to be retrieved, the same calculation is performed, the hash code is determined and a lookup of that location in the table results in the value that was stored there previously. Every object has the ability to be stored in a hash table. Class **Object** defines method **hashCode** to perform the hash code calculation. This method is inherited by all subclasses of **Object**. Method **hashCode** is overridden by **String** to provide a good hash code distribution based on the contents of the **String**. We will say more about hashing in Chapter 20.

The example in Fig. 10.5 demonstrates the **hashCode** method for two **Strings** containing "hello" and "Hello". Note that the hash code value for each **String** is different. That is because the **Strings** themselves are lexicographically different.

```

1  // Fig. 10.5: StringHashCode.java1
2  // This program demonstrates the method
3  // hashCode of the String class.
4
5  // Java extension packages
6  import javax.swing.*;
7
8  public class StringHashCode {
9
10     // test String hashCode method
11     public static void main( String args[] )
12     {
13         String s1 = "hello", s2 = "Hello";
14
15         String output =
16             "The hash code for \"" + s1 + "\" is " +
17             s1.hashCode() +
18             "\nThe hash code for \"" + s2 + "\" is " +
19             s2.hashCode();
20
21         JOptionPane.showMessageDialog( null, output,
22             "Demonstrating String Method hashCode",
23             JOptionPane.INFORMATION_MESSAGE );
24
25         System.exit( 0 );
26     }
27
28 } // end class StringHashCode

```

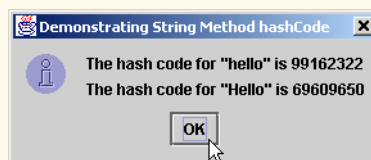


Fig. 10.5 **String** class **hashCode** method.

10.7 Locating Characters and Substrings in Strings

Often it is useful to search for a character or set of characters in a **String**. For example, if you are creating your own word processor, you might want to provide a capability for searching through the document. The application of Fig. 10.6 demonstrates the many versions of **String** methods **indexOf** and **lastIndexOf** that search for a specified character or substring in a **String**. All the searches in this example are performed on the **String** **letters** (initialized with "abcdefghijklmabcdefghijklm") in method **main** of class **StringIndexMethods**.

```

1  // Fig. 10.6: StringIndexMethods.java
2  // This program demonstrates the String
3  // class index methods.
4
5  // Java extension packages
6  import javax.swing.*;
7
8  public class StringIndexMethods {
9
10     // String searching methods
11     public static void main( String args[] )
12     {
13         String letters = "abcdefghijklmabcdefghijklm";
14
15         // test indexOf to locate a character in a string
16         String output = "'c' is located at index " +
17             letters.indexOf( 'c' );
18
19         output += "\n'a' is located at index " +
20             letters.indexOf( 'a', 1 );
21
22         output += "\n'$' is located at index " +
23             letters.indexOf( '$' );
24
25         // test lastIndexOf to find a character in a string
26         output += "\n\nLast 'c' is located at index " +
27             letters.lastIndexOf( 'c' );
28
29         output += "\n\nLast 'a' is located at index " +
30             letters.lastIndexOf( 'a', 25 );
31
32         output += "\n\nLast '$' is located at index " +
33             letters.lastIndexOf( '$' );
34
35         // test indexOf to locate a substring in a string
36         output += "\n\n\"def\" is located at index " +
37             letters.indexOf( "def" );
38
39         output += "\n\n\"def\" is located at index " +
40             letters.indexOf( "def", 7 );
41     }

```

Fig. 10.6 The **String** class searching methods (part 1 of 2).

```

42     output += "\n\"hello\" is located at index " +
43           letters.indexOf( "hello" );
44
45     // test lastIndexOf to find a substring in a string
46     output += "\n\nLast \"def\" is located at index " +
47           letters.lastIndexOf( "def" );
48
49     output += "\nLast \"def\" is located at index " +
50           letters.lastIndexOf( "def", 25 );
51
52     output += "\nLast \"hello\" is located at index " +
53           letters.lastIndexOf( "hello" );
54
55     JOptionPane.showMessageDialog( null, output,
56           "Demonstrating String Class \"index\" Methods",
57           JOptionPane.INFORMATION_MESSAGE );
58
59     System.exit( 0 );
60 }
61
62 } // end class StringIndexMethods

```

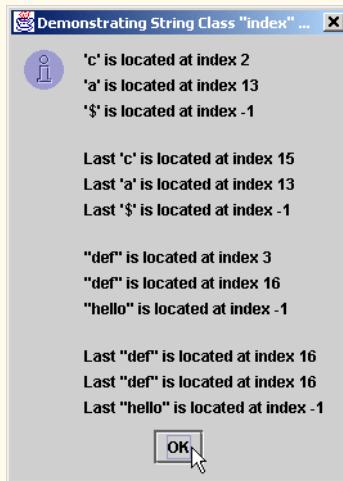


Fig. 10.6 The **String** class searching methods (part 2 of 2).

The statements at lines 16–23 use method **indexOf** to locate the first occurrence of a character in a **String**. If **indexOf** finds the character, **indexOf** returns the index of that character in the **String**; otherwise, **indexOf** returns **-1**. There are two versions of **indexOf** that search for characters in a **String**. The expression on line 17 uses method **indexOf** that takes one integer argument, which is the integer representation of a character. Remember that a character constant in single quotes is of type **char** and specifies the integer representation of the character in the Unicode character set. The expression at line 20 uses the second version of method **indexOf**, which takes two integer arguments—the integer representation of a character and the starting index at which the search of the **String** should begin.

The statements at lines 26–33 use method `lastIndexOf` to locate the last occurrence of a character in a `String`. Method `lastIndexOf` performs the search from the end of the `String` toward the beginning of the `String`. If method `lastIndexOf` finds the character, `lastIndexOf` returns the index of that character in the `String`; otherwise, `lastIndexOf` returns `-1`. There are two versions of `lastIndexOf` that search for characters in a `String`. The expression at line 27 uses the version of method `lastIndexOf` that takes one integer argument that is the integer representation of a character. The expression at line 30 uses the version of method `lastIndexOf` that takes two integer arguments—the integer representation of a character and the highest index from which to begin searching backward for the character.

Lines 36–53 of the program demonstrate versions of methods `indexOf` and `lastIndexOf` that each take a `String` as the first argument. These versions of the methods perform identically to those described above except that they search for sequences of characters (or substrings) that are specified by their `String` arguments.

10.8 Extracting Substrings from Strings

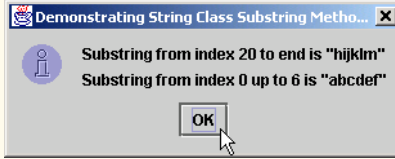
Class `String` provides two *substring* methods to enable a new `String` object to be created by copying part of an existing `String` object. Each method returns a new `String` object. Both methods are demonstrated in Fig. 10.7.

```

1  // Fig. 10.7: SubString.java
2  // This program demonstrates the
3  // String class substring methods.
4
5  // Java extension packages
6  import javax.swing.*;
7
8  public class SubString {
9
10     // test String substring methods
11     public static void main( String args[] )
12     {
13         String letters = "abcdefghijklmabcdefghijklm";
14
15         // test substring methods
16         String output = "Substring from index 20 to end is " +
17             "\"" + letters.substring( 20 ) + "\"\n";
18
19         output += "Substring from index 0 up to 6 is " +
20             "\"" + letters.substring( 0, 6 ) + "\"";
21
22         JOptionPane.showMessageDialog( null, output,
23             "Demonstrating String Class Substring Methods",
24             JOptionPane.INFORMATION_MESSAGE );
25
26         System.exit( 0 );
27     }
28
29 } // end class SubString

```

Fig. 10.7 `String` class *substring* methods (part 1 of 2).

Fig. 10.7 **String** class **substring** methods (part 2 of 2).

The expression **letters.substring(20)** from line 17 uses the **substring** method that takes one integer argument. The argument specifies the starting index from which characters are copied in the original **String**. The substring returned contains a copy of the characters from the starting index to the end of the **String**. If the index specified as an argument is outside the bounds of the **String**, the program generates a **StringIndexOutOfBoundsException**.

The expression **letters.substring(0, 6)** from line 17 uses the **substring** method that takes two integer arguments. The first argument specifies the starting index from which characters are copied in the original **String**. The second argument specifies the index one beyond the last character to be copied (i.e., copy up to, but not including, that index in the **String**). The substring returned contains copies of the specified characters from the original **String**. If the arguments are outside the bounds of the **String**, the program generates a **StringIndexOutOfBoundsException**.

10.9 Concatenating Strings

The **String** method **concat** (Fig. 10.8) concatenates two **String** objects and returns a new **String** object containing the characters from both original **Strings**. If the argument **String** has no characters in it, the original **String** is returned. The expression **s1.concat(s2)** at line 20 appends the characters from the **String** **s2** to the end of the **String** **s1**. The original **Strings** to which **s1** and **s2** refer are not modified.



Performance Tip 10.2

*In programs that frequently perform **String** concatenation, or other **String** modifications, it is more efficient to implement those modifications with class **StringBuffer** (covered in Section 10.13–Section 10.18).*

```

1  // Fig. 10.8: StringConcatenation.java
2  // This program demonstrates the String class concat method.
3  // Note that the concat method returns a new String object. It
4  // does not modify the object that invoked the concat method.
5
6  // Java extension packages
7  import javax.swing.*;
8
9  public class StringConcatenation {
10

```

Fig. 10.8 **String** method **concat** (part 1 of 2).

```

11 // test String method concat
12 public static void main( String args[] )
13 {
14     String s1 = new String( "Happy " ),
15         s2 = new String( "Birthday" );
16
17     String output = "s1 = " + s1 + "\ns2 = " + s2;
18
19     output += "\n\nResult of s1.concat( s2 ) = " +
20         s1.concat( s2 );
21
22     output += "\ns1 after concatenation = " + s1;
23
24     JOptionPane.showMessageDialog( null, output,
25         "Demonstrating String Method concat",
26         JOptionPane.INFORMATION_MESSAGE );
27
28     System.exit( 0 );
29 }
30
31 } // end class StringConcatenation

```

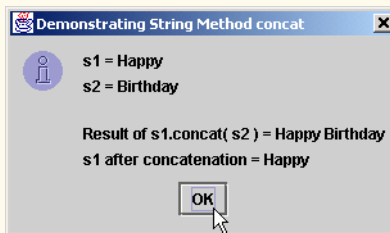


Fig. 10.8 **String** method **concat** (part 2 of 2).

10.10 Miscellaneous String Methods

Class **String** provides several methods that return modified copies of **Strings** or that return a character array. These methods are demonstrated in the application of Fig. 10.9.

Line 22 uses **String** method **replace** to return a new **String** object in which the method replaces every occurrence in **String** **s1** of character '**l**' (el) with character '**L**'. Method **replace** leaves the original **String** unchanged. If there are no occurrences of the first argument in the **String**, Method **replace** returns the original **String**.

Line 26 uses **String** method **toUpperCase** to generate a new **String** object with uppercase letters where corresponding lowercase letters reside in **s1**. The method returns a new **String** object containing the converted **String** and leaves the original **String** unchanged. If there are no characters to convert to uppercase letters, method **toUpperCase** returns the original **String**.

Line 27 uses **String** method **toLowerCase** to return a new **String** object with lowercase letters where corresponding uppercase letters reside in **s1**. The original **String** remains unchanged. If there are no characters to convert to lowercase letters, method **toLowerCase** returns the original **String**.

```
1 // Fig. 10.9: StringMiscellaneous2.java
2 // This program demonstrates the String methods replace,
3 // toLowerCase, toUpperCase, trim, toString and toCharArray
4
5 // Java extension packages
6 import javax.swing.*;
7
8 public class StringMiscellaneous2 {
9
10     // test miscellaneous String methods
11     public static void main( String args[] )
12     {
13         String s1 = new String( "hello" ),
14             s2 = new String( "GOOD BYE" ),
15             s3 = new String( "   spaces   " );
16
17         String output = "s1 = " + s1 + "\ns2 = " + s2 +
18             "\ns3 = " + s3;
19
20         // test method replace
21         output += "\n\nReplace 'l' with 'L' in s1: " +
22             s1.replace( 'l', 'L' );
23
24         // test toLowerCase and toUpperCase
25         output +=
26             "\n\ns1.toUpperCase() = " + s1.toUpperCase() +
27             "\ns2.toLowerCase() = " + s2.toLowerCase();
28
29         // test trim method
30         output += "\n\ns3 after trim = \"" + s3.trim() + "\"";
31
32         // test toString method
33         output += "\n\ns1 = " + s1.toString();
34
35         // test toCharArray method
36         char charArray[] = s1.toCharArray();
37
38         output += "\n\ns1 as a character array = ";
39
40         for ( int count = 0; count < charArray.length; ++count )
41             output += charArray[ count ];
42
43         JOptionPane.showMessageDialog( null, output,
44             "Demonstrating Miscellaneous String Methods",
45             JOptionPane.INFORMATION_MESSAGE );
46
47         System.exit( 0 );
48     }
49
50 } // end class StringMiscellaneous2
```

Fig. 10.9 Miscellaneous **String** methods (part 1 of 2).

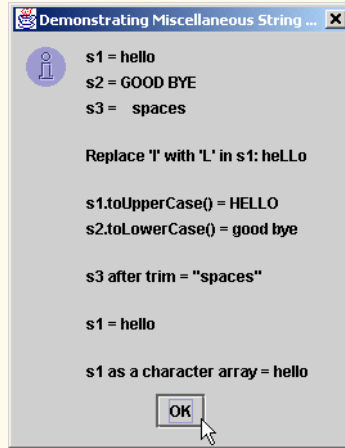


Fig. 10.9 Miscellaneous **String** methods (part 2 of 2).

Line 30 uses **String** method **trim** to generate a new **String** object that removes all white-space characters that appear at the beginning or end of the **String** to which the **trim** message is sent. The method returns a new **String** object containing the **String** without leading or trailing white-space characters. The original **String** remains unchanged.

Line 33 uses **String** method **toString** to return the **String** **s1**. Why is the **toString** method provided for class **String**? All objects can be converted to **Strings** in Java by using method **toString**, which originates in class **Object**. If a class that inherits from **Object** (such as **String**) does not override method **toString**, the default version from class **Object** is used. The default version in **Object** creates a **String** consisting of the object's class name and the hash code for the object. The **toString** method normally is used to express the contents of an object as text. Method **toString** is provided in class **String** to ensure that the proper **String** value is returned.

Line 36 creates a new character array containing a copy of the characters in **String** **s1** and assigns it to **charArray**.

10.11 Using **String** Method **valueOf**

Class **String** provides a set of **static** class methods that take arguments of various types, convert those arguments to **Strings** and return them as **String** objects. Class **String.valueOf** (Fig. 10.10) demonstrates the **String** class **valueOf** methods.

```

1  // Fig. 10.10: String.valueOf.java
2  // This program demonstrates the String class valueOf methods.
3
4  // Java extension packages
5  import javax.swing.*;
6

```

Fig. 10.10 **String** class **valueOf** methods (part 1 of 2).

```

7  public class StringValueOf {
8
9      // test String valueOf methods
10     public static void main( String args[] )
11     {
12         char charArray[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
13         boolean b = true;
14         char c = 'Z';
15         int i = 7;
16         long l = 10000000;
17         float f = 2.5f;
18         double d = 33.333;
19
20         Object o = "hello"; // assign to an Object reference
21         String output;
22
23         output = "char array = " + String.valueOf( charArray ) +
24             "\npart of char array = " +
25             String.valueOf( charArray, 3, 3 ) +
26             "\nboolean = " + String.valueOf( b ) +
27             "\nchar = " + String.valueOf( c ) +
28             "\nint = " + String.valueOf( i ) +
29             "\nlong = " + String.valueOf( l ) +
30             "\nfloat = " + String.valueOf( f ) +
31             "\ndouble = " + String.valueOf( d ) +
32             "\nObject = " + String.valueOf( o );
33
34         JOptionPane.showMessageDialog( null, output,
35             "Demonstrating String Class valueOf Methods",
36             JOptionPane.INFORMATION_MESSAGE );
37
38         System.exit( 0 );
39     }
40
41 } // end class StringValueOf

```

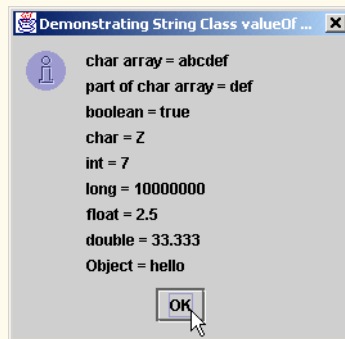


Fig. 10.10 **String** class **valueOf** methods (part 2 of 2).

The expression **String.valueOf(charArray)** from line 23 copies the contents of the character array **charArray** into a new **String** object and returns the new **String**. The expression **String.valueOf(charArray, 3, 3)** from line 25

copies a portion of the contents of the character array **charArray** into a new **String** object and returns the new **String**. The second argument specifies the starting index from which the characters are copied. The third argument specifies the number of characters to copy.

There are seven other versions of method **valueOf**, which take arguments of type **boolean**, **char**, **int**, **long**, **float**, **double** and **Object**, respectively. These are demonstrated in lines 26–32 of the program. Note that the version of **valueOf** that takes an **Object** as an argument can do so because all **Objects** can be converted to **Strings** with method **toString**.

10.12 String Method **intern**

Comparing large **String** objects is a relatively slow operation. **String** method **intern** can improve **String** comparison performance. When **String** method **intern** is invoked on a **String** object, it returns a reference to a **String** object that is guaranteed to have the same contents as the original **String**. Class **String** maintains the resulting **String** during program execution. Subsequently, if the program invokes method **intern** on other **String** objects with contents identical to the original **String**, method **intern** returns a reference to the copy of the **String** maintained in memory by class **String**. If a program uses **intern** on extremely large **Strings**, the program can compare those **Strings** faster by using the **==** operator, which simply compares two references—a fast operation. Using standard **String** methods such as **equals** and **equalsIgnoreCase** can be slower, because they compare corresponding characters in each **String**. For large **Strings**, this is a time-consuming, iterative operation. The program of Fig. 10.11 demonstrates the **intern** method.

The program declares five **String** references—**s1**, **s2**, **s3**, **s4** and **output**. **Strings** **s1** and **s2** are initialized with new **String** objects that each contain a copy of "hello". The first **if** structure (lines 20–23) uses operator **==** to determine that **Strings** **s1** and **s2** are the same object. References **s1** and **s2** refer to different objects, because they were initialized with new **String** objects.

The second **if** structure (lines 26–29) uses method **equals** to determine that the contents of **Strings** **s1** and **s2** are equal. They were each initialized with copies of "hello", so they have the same contents.

```

1  // Fig. 10.11: StringIntern.java
2  // This program demonstrates the intern method
3  // of the String class.
4
5  // Java extension packages
6  import javax.swing.*;
7
8  public class StringIntern {
9
10     // test String method intern
11     public static void main( String args[] )
12     {
13         String s1, s2, s3, s4, output;

```

Fig. 10.11 **String** class **intern** method (part 1 of 3).

```
14
15     s1 = new String( "hello" );
16     s2 = new String( "hello" );
17
18     // test strings to determine if they are same
19     // String object in memory
20     if ( s1 == s2 )
21         output = "s1 and s2 are the same object in memory";
22     else
23         output = "s1 and s2 are not the same object in memory";
24
25     // test strings for equality of contents
26     if ( s1.equals( s2 ) )
27         output += "\ns1 and s2 are equal";
28     else
29         output += "\ns1 and s2 are not equal";
30
31     // use String intern method to get a unique copy of
32     // "hello" referred to by both s3 and s4
33     s3 = s1.intern();
34     s4 = s2.intern();
35
36     // test strings to determine if they are same
37     // String object in memory
38     if ( s3 == s4 )
39         output += "\ns3 and s4 are the same object in memory";
40     else
41         output +=
42             "\ns3 and s4 are not the same object in memory";
43
44     // determine if s1 and s3 refer to same object
45     if ( s1 == s3 )
46         output +=
47             "\ns1 and s3 are the same object in memory";
48     else
49         output +=
50             "\ns1 and s3 are not the same object in memory";
51
52     // determine if s2 and s4 refer to same object
53     if ( s2 == s4 )
54         output += "\ns2 and s4 are the same object in memory";
55     else
56         output +=
57             "\ns2 and s4 are not the same object in memory";
58
59     // determine if s1 and s4 refer to same object
60     if ( s1 == s4 )
61         output += "\ns1 and s4 are the same object in memory";
62     else
63         output +=
64             "\ns1 and s4 are not the same object in memory";
65
```

Fig. 10.11 **String** class **intern** method (part 2 of 3).

```

66     JOptionPane.showMessageDialog( null, output,
67         "Demonstrating String Method intern",
68         JOptionPane.INFORMATION_MESSAGE );
69
70     System.exit( 0 );
71 }
72
73 } // end class StringIntern

```

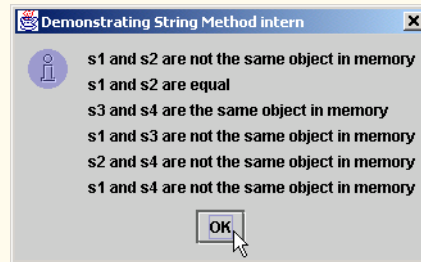


Fig. 10.11 **String** class **intern** method (part 3 of 3).

Line 33 uses method **intern** to get a reference to a **String** with the same contents as object **s1** and assigns the reference to **s3**. The **String** to which **s3** refers is maintained in memory by class **String**. Line 34 also uses method **intern** to get a reference to a **String** object. However, because **String s1** and **String s2** have the same contents, the reference returned by this call to **intern** is a reference to the same **String** object returned by **s1.intern()**. The third **if** structure (lines 38–42) uses operator **==** to determine that **Strings s3** and **s4** refer to the same object.

The fourth **if** structure (lines 45–50) uses operator **==** to determine that **Strings s1** and **s3** are *not* the same object. Technically, they could refer to the same object, but they are not guaranteed to refer to the same object unless the objects they refer to were returned by calls to **intern** on **Strings** with the same contents. In this case, **s1** refers to the **String** it was assigned in method **main** and **s3** refers to the **String** with the same contents maintained by class **String**.

The fifth **if** structure (lines 53–57) uses operator **==** to determine that **Strings s2** and **s4** are *not* the same object, because the second **intern** call results in a reference to the same object returned by **s1.intern()**, not **s2**. Similarly, the sixth **if** structure (lines 60–64) uses operator **==** to determine that **Strings s1** and **s4** are not the same object, because the second **intern** call results in a reference to the same object returned by **s1.intern()**, not **s1**.

10.13 StringBuffer Class

The **String** class provides many capabilities for processing **Strings**. However, once a **String** object is created, its contents can never change. The next several sections discuss the features of class **StringBuffer** for creating and manipulating dynamic string information—i.e., modifiable **Strings**. Every **StringBuffer** is capable of storing a number of characters specified by its capacity. If the capacity of a **StringBuffer** is exceeded, the capacity is automatically expanded to accommodate the additional charac-

ters. As we will see, class **StringBuffer** is also used to implement operators **+** and **+=** for **String** concatenation.



Performance Tip 10.3

String objects are constant strings and **StringBuffer** objects are modifiable strings. Java distinguishes constant strings from modifiable strings for optimization purposes; in particular, Java can perform certain optimizations involving **String** objects (such as sharing one **String** object among multiple references) because it knows these objects will not change.



Performance Tip 10.4

When given the choice between using a **String** object to represent a string versus a **StringBuffer** object to represent that string, always use a **String** object if indeed the object will not change; this improves performance.



Common Programming Error 10.4

Invoking **StringBuffer** methods that are not methods of class **String** on **String** objects is a syntax error.

10.14 StringBuffer Constructors

Class **StringBuffer** provides three constructors (demonstrated in Fig. 10.12). Line 14 uses the default **StringBuffer** constructor to create a **StringBuffer** with no characters in it and an initial capacity of 16 characters. Line 15 uses **StringBuffer** constructor that takes an integer argument to create a **StringBuffer** with no characters in it and the initial capacity specified in the integer argument (i.e., 10). Line 16 uses the **StringBuffer** constructor that takes a **String** argument to create a **StringBuffer** containing the characters of the **String** argument. The initial capacity is the number of characters in the **String** argument plus 16.

The statement on lines 18–21 uses **StringBuffer** method **toString** to convert the **StringBuffers** into **String** objects that can be displayed with **drawString**. Note the use of operator **+** to concatenate **Strings** for output. In Section 10.17, we discuss how Java uses **StringBuffers** to implement the **+** and **+=** operators for **String** concatenation.

```

1  // Fig. 10.12: StringBufferConstructors.java
2  // This program demonstrates the StringBuffer constructors.
3
4  // Java extension packages
5  import javax.swing.*;
6
7  public class StringBufferConstructors {
8
9      // test StringBuffer constructors
10     public static void main( String args[] )
11     {
12         StringBuffer buffer1, buffer2, buffer3;
13     }

```

Fig. 10.12 **StringBuffer** class constructors (part 1 of 2).

```

14     buffer1 = new StringBuffer();
15     buffer2 = new StringBuffer( 10 );
16     buffer3 = new StringBuffer( "hello" );
17
18     String output =
19         "buffer1 = \"" + buffer1.toString() + "\"" +
20         "\nbuffer2 = \"" + buffer2.toString() + "\"" +
21         "\nbuffer3 = \"" + buffer3.toString() + "\"";
22
23     JOptionPane.showMessageDialog( null, output,
24         "Demonstrating StringBuffer Class Constructors",
25         JOptionPane.INFORMATION_MESSAGE );
26
27     System.exit( 0 );
28 }
29
30 } // end class StringBufferConstructors

```

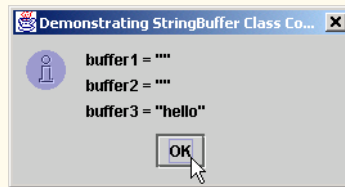


Fig. 10.12 **StringBuffer** class constructors (part 2 of 2).

10.15 **StringBuffer** Methods **length**, **capacity**, **setLength** and **ensureCapacity**

Class **StringBuffer** provides the **length** and **capacity** methods to return the number of characters currently in a **StringBuffer** and the number of characters that can be stored in a **StringBuffer** without allocating more memory, respectively. Method **ensureCapacity** is provided to allow the programmer to guarantee that a **StringBuffer** has a minimum capacity. Method **setLength** is provided to enable the programmer to increase or decrease the length of a **StringBuffer**. The program of Fig. 10.13 demonstrates these methods.

```

1  // Fig. 10.13: StringBufferCapLen.java
2  // This program demonstrates the length and
3  // capacity methods of the StringBuffer class.
4
5  // Java extension packages
6  import javax.swing.*;
7
8  public class StringBufferCapLen {
9
10     // test StringBuffer methods for capacity and length
11     public static void main( String args[] )
12     {

```

Fig. 10.13 **StringBuffer** **length** and **capacity** methods (part 1 of 2).

```

13     StringBuffer buffer =
14         new StringBuffer( "Hello, how are you?" );
15
16     String output = "buffer = " + buffer.toString() +
17         "\nlength = " + buffer.length() +
18         "\ncapacity = " + buffer.capacity();
19
20     buffer.ensureCapacity( 75 );
21     output += "\n\nNew capacity = " + buffer.capacity();
22
23     buffer.setLength( 10 );
24     output += "\n\nNew length = " + buffer.length() +
25         "\nbuf = " + buffer.toString();
26
27     JOptionPane.showMessageDialog( null, output,
28         "StringBuffer length and capacity Methods",
29         JOptionPane.INFORMATION_MESSAGE );
30
31     System.exit( 0 );
32 }
33
34 } // end class StringBufferCapLen

```

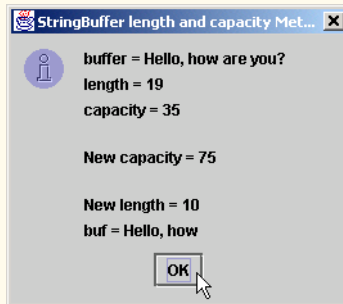


Fig. 10.13 **StringBuffer** **length** and **capacity** methods (part 2 of 2).

The program contains one **StringBuffer** called **buffer**. Lines 13–14 of the program use the **StringBuffer** constructor that takes a **String** argument to instantiate and initialize the **StringBuffer** with "Hello, how are you?". Lines 16–18 append to **output** the contents, the length and the capacity of the **StringBuffer**. Notice in the output window that the capacity of the **StringBuffer** is initially 35. Remember that the **StringBuffer** constructor that takes a **String** argument creates a **StringBuffer** object with an initial capacity that is the length of the **String** passed as an argument plus 16.

Line 20 expands the capacity of the **StringBuffer** to a minimum of 75 characters. Actually, if the original capacity is less than the argument, the method ensures a capacity that is the greater of the number specified as an argument or twice the original capacity plus 2. If the **StringBuffer**'s current capacity is more than the specified capacity, the **StringBuffer**'s capacity remains unchanged.

Line 23 uses method **setLength** to set the length of the **StringBuffer** to 10. If the specified length is less than the current number of characters in the **StringBuffer**,

the characters are truncated to the specified length (i.e., the characters in the **StringBuffer** after the specified length are discarded). If the specified length is greater than the number of characters currently in the **StringBuffer**, null characters (characters with the numeric representation 0) are appended to the **StringBuffer** until the total number of characters in the **StringBuffer** is equal to the specified length.

10.16 StringBuffer Methods `charAt`, `setCharAt`, `getChars` and `reverse`

Class **StringBuffer** provides the `charAt`, `setCharAt`, `getChars` and `reverse` methods to manipulate the characters in a **StringBuffer**. Method `charAt` takes an integer argument and returns the character in the **StringBuffer** at that index. Method `setCharAt` takes an integer and a character argument and sets the character at the specified position to the character argument. The index specified in the `charAt` and `setCharAt` methods must be greater than or equal to 0 and less than the **StringBuffer** length; otherwise, a **StringIndexOutOfBoundsException** is generated.



Common Programming Error 10.5

*Attempting to access a character that is outside the bounds of a **StringBuffer** (i.e., an index less than 0 or an index greater than or equal to the **StringBuffer**'s length) results in a **StringIndexOutOfBoundsException**.*

Method `getChars` returns a character array containing a copy of the characters in the **StringBuffer**. This method takes four arguments—the starting index from which characters should be copied in the **StringBuffer**, the index one past the last character to be copied from the **StringBuffer**, the character array into which the characters are to be copied and the starting location in the character array where the first character should be placed. Method `reverse` reverses the contents of the **StringBuffer**. Each of these methods is demonstrated in Fig. 10.14.

```

1  // Fig. 10.14: StringBufferChars.java
2  // The charAt, setCharAt, getChars, and reverse methods
3  // of class StringBuffer.
4
5  // Java extension packages
6  import javax.swing.*;
7
8  public class StringBufferChars {
9
10     // test StringBuffer character methods
11     public static void main( String args[] )
12     {
13         StringBuffer buffer = new StringBuffer( "hello there" );
14
15         String output = "buffer = " + buffer.toString() +
16             "\nCharacter at 0: " + buffer.charAt( 0 ) +
17             "\nCharacter at 4: " + buffer.charAt( 4 );
18     }
19 }

```

Fig. 10.14 **StringBuffer** class character manipulation methods.

```

19     char charArray[] = new char[ buffer.length() ];
20     buffer.getChars( 0, buffer.length(), charArray, 0 );
21     output += "\n\nThe characters are: ";
22
23     for ( int count = 0; count < charArray.length; ++count )
24         output += charArray[ count ];
25
26     buffer.setCharAt( 0, 'H' );
27     buffer.setCharAt( 6, 'T' );
28     output += "\n\nbuf = " + buffer.toString();
29
30     buffer.reverse();
31     output += "\n\nbuf = " + buffer.toString();
32
33     JOptionPane.showMessageDialog( null, output,
34         "Demonstrating StringBuffer Character Methods",
35         JOptionPane.INFORMATION_MESSAGE );
36
37     System.exit( 0 );
38 }
39
40 } // end class StringBufferChars

```

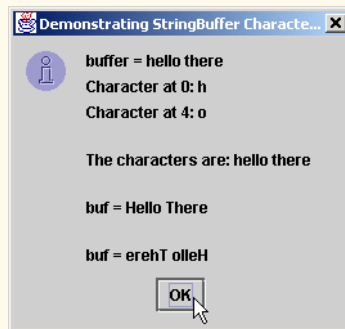


Fig. 10.14 **StringBuffer** class character manipulation methods.

10.17 StringBuffer append Methods

Class **StringBuffer** provides 10 overloaded **append** methods to allow various data-type values to be added to the end of a **StringBuffer**. Versions are provided for each of the primitive data types and for character arrays, **Strings** and **Objects**. (Remember that method **toString** produces a **String** representation of any **Object**.) Each of the methods takes its argument, converts it to a **String** and appends it to the **StringBuffer**. The **append** methods are demonstrated in Fig. 10.15. [Note: Line 20 specifies the literal value **2.5f** as the initial value of a **float** variable. Normally, Java treats a floating-point literal value as type **double**. Appending the letter **f** to the literal **2.5** indicates to the compiler that **2.5** should be treated as type **float**. Without this indication the compiler generates a syntax error, because a **double** value cannot be assigned directly to a **float** variable in Java.]

```
1  // Fig. 10.15: StringBufferAppend.java
2  // This program demonstrates the append
3  // methods of the StringBuffer class.
4
5  // Java extension packages
6  import javax.swing.*;
7
8  public class StringBufferAppend {
9
10     // test StringBuffer append methods
11     public static void main( String args[] )
12     {
13         Object o = "hello";
14         String s = "good bye";
15         char charArray[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
16         boolean b = true;
17         char c = 'Z';
18         int i = 7;
19         long l = 10000000;
20         float f = 2.5f;
21         double d = 33.333;
22         StringBuffer buffer = new StringBuffer();
23
24         buffer.append( o );
25         buffer.append( " " );
26
27         buffer.append( s );
28         buffer.append( " " );
29         buffer.append( charArray );
30         buffer.append( " " );
31         buffer.append( charArray, 0, 3 );
32         buffer.append( " " );
33         buffer.append( b );
34         buffer.append( " " );
35         buffer.append( c );
36         buffer.append( " " );
37         buffer.append( i );
38         buffer.append( " " );
39         buffer.append( l );
40         buffer.append( " " );
41         buffer.append( f );
42         buffer.append( " " );
43         buffer.append( d );
44
45         JOptionPane.showMessageDialog( null,
46             "buffer = " + buffer.toString(),
47             "Demonstrating StringBuffer append Methods",
48             JOptionPane.INFORMATION_MESSAGE );
49
50         System.exit( 0 );
51     }
52 }
53 // end StringBufferAppend
```

Fig. 10.15 **StringBuffer** class **append** methods (part 1 of 2).

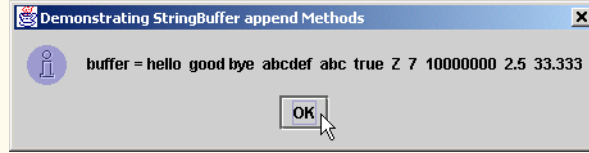


Fig. 10.15 **StringBuffer** class **append** methods (part 2 of 2).

Actually, **StringBuffers** and the **append** methods are used by the compiler to implement the **+** and **+=** operators for concatenating **Strings**. For example, assuming the following declarations:

```
String string1 = "hello";
String string2 = "BC"
int value = 22;
```

the statement

```
String s = string1 + string2 + value;
```

concatenates "hello", "BC" and 22. The concatenation is performed as follows:

```
new StringBuffer().append( "hello" ).append( "BC" ).append(
    22 ).toString();
```

First, Java creates a **StringBuffer**, then appends to the **StringBuffer** the **String** "hello", the **String** "BC" and the integer 22. Next, **StringBuffer**'s **toString** converts the **StringBuffer** to a **String** representation and the result is assigned to **String s**. The statement

```
s += "!";
```

is performed as follows:

```
s = new StringBuffer().append( s ).append( "!" ).toString();
```

First, Java creates a **StringBuffer**, then appends to the **StringBuffer** the current contents of **s** followed by "!". Next, **StringBuffer**'s **toString** converts the **StringBuffer** to a **String** representation and the result is assigned to **s**

10.18 **StringBuffer** Insertion and Deletion Methods

Class **StringBuffer** provides nine overloaded **insert** methods to allow various data-type values to be inserted at any position in a **StringBuffer**. Versions are provided for each of the primitive data types and for character arrays, **Strings** and **Objects**. (Remember that method **toString** produces a **String** representation of any **Object**.) Each of the methods takes its second argument, converts it to a **String** and inserts it preceding the index specified by the first argument. The index specified by the first argument must be greater than or equal to 0 and less than the length of the **StringBuffer**; otherwise, a **StringIndexOutOfBoundsException** is generated. Class **StringBuffer** also provides methods **delete** and **deleteCharAt** for deleting characters at

any position in a **StringBuffer**. Method **delete** takes two arguments—the starting subscript and the subscript one past the end of the characters to delete. All characters beginning at the starting subscript up to, but not including the ending subscript are deleted. Method **deleteCharAt** takes one argument—the subscript of the character to delete. Invalid subscripts cause both methods to throw a **StringIndexOutOfBoundsException**. The **insert** and delete methods are demonstrated in Fig. 10.16.

```

1  // Fig. 10.16: StringBufferInsert.java
2  // This program demonstrates the insert and delete
3  // methods of class StringBuffer.
4
5  // Java extension packages
6  import javax.swing.*;
7
8  public class StringBufferInsert {
9
10     // test StringBuffer insert methods
11     public static void main( String args[] )
12     {
13         Object o = "hello";
14         String s = "good bye";
15         char charArray[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
16         boolean b = true;
17         char c = 'K';
18         int i = 7;
19         long l = 10000000;
20         float f = 2.5f;
21         double d = 33.333;
22         StringBuffer buffer = new StringBuffer();
23
24         buffer.insert( 0, o );
25         buffer.insert( 0, " " );
26         buffer.insert( 0, s );
27         buffer.insert( 0, " " );
28         buffer.insert( 0, charArray );
29         buffer.insert( 0, " " );
30         buffer.insert( 0, b );
31         buffer.insert( 0, " " );
32         buffer.insert( 0, c );
33         buffer.insert( 0, " " );
34         buffer.insert( 0, i );
35         buffer.insert( 0, " " );
36         buffer.insert( 0, l );
37         buffer.insert( 0, " " );
38         buffer.insert( 0, f );
39         buffer.insert( 0, " " );
40         buffer.insert( 0, d );
41
42         String output =
43             "buffer after inserts:\n" + buffer.toString();
44     }

```

Fig. 10.16 **StringBuffer** class **insert** and **delete** methods (part 1 of 2).

```

45     buffer.deleteCharAt( 10 ); // delete 5 in 2.5
46     buffer.delete( 2, 6 );    // delete .333 in 33.333
47
48     output +=
49         "\n\nbuffer after deletes:\n" + buffer.toString();
50
51     JOptionPane.showMessageDialog( null, output,
52         "Demonstrating StringBuffer Inserts and Deletes",
53         JOptionPane.INFORMATION_MESSAGE );
54
55     System.exit( 0 );
56 }
57
58 } // end class StringBufferInsert

```

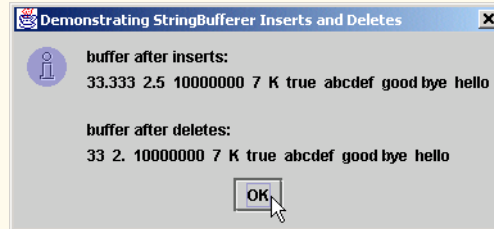


Fig. 10.16 `StringBuffer` class `insert` and `delete` methods (part 2 of 2).

10.19 Character Class Examples

Java provides a number of classes that enable primitive variables to be treated as objects. The classes are **Boolean**, **Character**, **Double**, **Float**, **Byte**, **Short**, **Integer** and **Long**. These classes (except **Boolean** and **Character**) are derived from **Number**. These eight classes are known as *type wrappers*, and they are part of the **java.lang** package. Objects of these classes can be used anywhere in a program that an **Object** or a **Number** is expected. In this section, we present class **Character**—the type-wrapper class for characters.

Most **Character** class methods are **static**, take at least a character argument and perform either a test or a manipulation of the character. This class also contains a constructor that receives a **char** argument to initialize a **Character** object and several non-**static** methods. Most of the methods of class **Character** are presented in the next three examples. For more information on class **Character** (and all the wrapper classes), see the **java.lang** package in the Java API documentation.

Figure 10.17 demonstrates some **static** methods that test characters to determine whether they are a specific character type and the **static** methods that perform case conversions on characters. Each method is used in method **buildOutput** of class **StaticCharMethods**. You can enter any character and apply the preceding methods to the character. Note the use of inner classes for the event handling as demonstrated in Chapter 9.

Line 63 uses **Character** method **isDefined** to determine if character **c** is defined in the Unicode character set. If so, the method returns **true**; otherwise, it returns **false**.

```

1  // Fig. 10.17: StaticCharMethods.java
2  // Demonstrates the static character testing methods
3  // and case conversion methods of class Character
4  // from the java.lang package.
5
6  // Java core packages
7  import java.awt.*;
8  import java.awt.event.*;
9
10 // Java extension packages
11 import javax.swing.*;
12
13 public class StaticCharMethods extends JFrame {
14     private char c;
15     private JLabel promptLabel;
16     private JTextField inputField;
17     private JTextArea outputArea;
18
19     // set up GUI
20     public StaticCharMethods()
21     {
22         super( "Static Character Methods" );
23
24         Container container = getContentPane();
25         container.setLayout( new FlowLayout() );
26
27         promptLabel =
28             new JLabel( "Enter a character and press Enter" );
29         container.add( promptLabel );
30
31         inputField = new JTextField( 5 );
32
33         inputField.addActionListener(
34
35             // anonymous inner class
36             new ActionListener() {
37
38                 // handle text field event
39                 public void actionPerformed((ActionEvent event) )
40                 {
41                     String s = event.getActionCommand();
42                     c = s.charAt( 0 );
43                     buildOutput();
44                 }
45
46             } // end anonymous inner class
47
48         ); // end call to addActionListener

```

Fig. 10.17 **static** character testing methods and case conversion methods of class **Character** (part 1 of 3).

```

49
50     container.add( inputField );
51
52     outputArea = new JTextArea( 10, 20 );
53     container.add( outputArea );
54
55     setSize( 300, 250 ); // set the window size
56     show();             // show the window
57 }
58
59 // display character info in outputArea
60 public void buildOutput()
61 {
62     outputArea.setText(
63         "is defined: " + Character.isDefined( c ) +
64         "\nis digit: " + Character.isDigit( c ) +
65         "\nis Java letter: " +
66         Character.isJavaIdentifierStart( c ) +
67         "\nis Java letter or digit: " +
68         Character.isJavaIdentifierPart( c ) +
69         "\nis letter: " + Character.isLetter( c ) +
70         "\nis letter or digit: " +
71         Character.isLetterOrDigit( c ) +
72         "\nis lower case: " + Character.isLowerCase( c ) +
73         "\nis upper case: " + Character.isUpperCase( c ) +
74         "\nto upper case: " + Character.toUpperCase( c ) +
75         "\nto lower case: " + Character.toLowerCase( c ) );
76 }
77
78 // execute application
79 public static void main( String args[] )
80 {
81     StaticCharMethods application = new StaticCharMethods();
82
83     application.addWindowListener(
84
85         // anonymous inner class
86         new WindowAdapter() {
87
88             // handle event when user closes window
89             public void windowClosing( WindowEvent windowEvent )
90             {
91                 System.exit( 0 );
92             }
93
94             } // end anonymous inner class
95
96     ); // end call to addWindowListener
97
98 } // end method main
99
100 } // end class StaticCharMethods

```

Fig. 10.17 **static** character testing methods and case conversion methods of class **Character** (part 2 of 3).



Fig. 10.17 **static** character testing methods and case conversion methods of class **Character** (part 3 of 3).

Line 64 uses **Character** method **isDigit** to determine whether character **c** is a defined Unicode digit. If so, the method returns **true**; otherwise, it returns **false**.

Line 66 uses **Character** method **isJavaIdentifierStart** to determine whether **c** is a character that can be used as the first character of an identifier in Java—i.e., a letter, an underscore (**_**) or a dollar sign (**\$**). If so, the method returns **true**; otherwise, it returns **false**. Line 68 uses method **Character** method **isJavaIdentifierPart** to determine whether character **c** is a character that can be used in an identifier in Java—i.e., a digit, a letter, an underscore (**_**) or a dollar sign (**\$**). If so, the method returns **true**; otherwise, it returns **false**.

Line 69 uses method **Character** method **isLetter** to determine whether character **c** is a letter. If so, the method returns **true**; otherwise, it returns **false**. Line 71 uses method **Character** method **isLetterOrDigit** to determine whether character **c** is a letter or a digit. If so, the method returns **true**; otherwise, it returns **false**.

Line 72 uses method **Character** method **isLowerCase** to determine whether character **c** is a lowercase letter. If so, the method returns **true**; otherwise, it returns **false**. Line 73 uses method **Character** method **isUpperCase** to determine whether character **c** is an uppercase letter. If so, the method returns **true**; otherwise, it returns **false**.

Line 74 uses method **Character** method **toUpperCase** to convert the character **c** to its uppercase equivalent. The method returns the converted character if the character has an uppercase equivalent; otherwise, the method returns its original argument. Line 75 uses method **Character** method **toLowerCase** to convert the character **c** to its lowercase

equivalent. The method returns the converted character if the character has a lowercase equivalent; otherwise, the method returns its original argument.

Figure 10.18 demonstrates **static Character** methods **digit** and **forDigit**, which perform conversions between characters and digits in different number systems. Common number systems include decimal (base 10), octal (base 8), hexadecimal (base 16) and binary (base 2). The base of a number is also known as its *radix*. For more information on conversions between number systems, see Appendix E.

Line 52 uses method **forDigit** to convert the integer **digit** into a character in the number system specified by the integer **radix** (also known as the base of the number). For example, the integer **13** in base 16 (the **radix**) has the character value **'d'**. Note that the lowercase and uppercase letters are equivalent in number systems.

Line 77 uses method **digit** to convert the character **c** into an integer in the number system specified by the integer **radix** (i.e., the base of the number). For example, the character **'A'** in base 16 (the **radix**) has the integer value 10.

```

1  // Fig. 10.18: StaticCharMethods2.java
2  // Demonstrates the static character conversion methods
3  // of class Character from the java.lang package.
4
5  // Java core packages
6  import java.awt.*;
7  import java.awt.event.*;
8
9  // Java extension packages
10 import javax.swing.*;
11
12 public class StaticCharMethods2 extends JFrame {
13     private char c;
14     private int digit, radix;
15     private JLabel prompt1, prompt2;
16     private JTextField input, radixField;
17     private JButton toChar, toInt;
18
19     public StaticCharMethods2()
20     {
21         super( "Character Conversion Methods" );
22
23         // set up GUI and event handling
24         Container container = getContentPane();
25         container.setLayout( new FlowLayout() );
26
27         prompt1 = new JLabel( "Enter a digit or character " );
28         input = new JTextField( 5 );
29         container.add( prompt1 );
30         container.add( input );
31
32         prompt2 = new JLabel( "Enter a radix " );
33         radixField = new JTextField( 5 );
34         container.add( prompt2 );
35         container.add( radixField );

```

Fig. 10.18 **Character** class **static** conversion methods (part 1 of 3).

```

36
37     toChar = new JButton( "Convert digit to character" );
38
39     toChar.addActionListener(
40
41         // anonymous inner class
42         new ActionListener() {
43
44             // handle toChar JButton event
45             public void actionPerformed((ActionEvent actionEvent) )
46             {
47                 digit = Integer.parseInt( input.getText() );
48                 radix =
49                     Integer.parseInt( radixField.getText() );
50                 JOptionPane.showMessageDialog( null,
51                     "Convert digit to character: " +
52                     Character.forDigit( digit, radix ) );
53             }
54
55         } // end anonymous inner class
56
57 ); // end call to addActionListener
58
59 container.add( toChar );
60
61 toInt = new JButton( "Convert character to digit" );
62
63 toInt.addActionListener(
64
65     // anonymous inner class
66     new ActionListener() {
67
68         // handle toInt JButton event
69         public void actionPerformed((ActionEvent actionEvent) )
70         {
71             String s = input.getText();
72             c = s.charAt( 0 );
73             radix =
74                 Integer.parseInt( radixField.getText() );
75             JOptionPane.showMessageDialog( null,
76                 "Convert character to digit: " +
77                 Character.digit( c, radix ) );
78         }
79
80     } // end anonymous inner class
81
82 ); // end call to addActionListener
83
84 container.add( toInt );
85
86 setSize( 275, 150 ); // set the window size
87 show();              // show the window
88 }

```

Fig. 10.18 **Character** class **static** conversion methods (part 2 of 3).

```

89
90 // execute application
91 public static void main( String args[] )
92 {
93     StaticCharMethods2 application = new StaticCharMethods2();
94
95     application.addWindowListener(
96
97         // anonymous inner class
98         new WindowAdapter() {
99
100             // handle event when user closes window
101             public void windowClosing( WindowEvent windowEvent )
102             {
103                 System.exit( 0 );
104             }
105
106         } // end anonymous inner class
107
108     ); // end call to addWindowListener
109
110 } // end method main
111
112 } // end class StaticCharMethods2

```

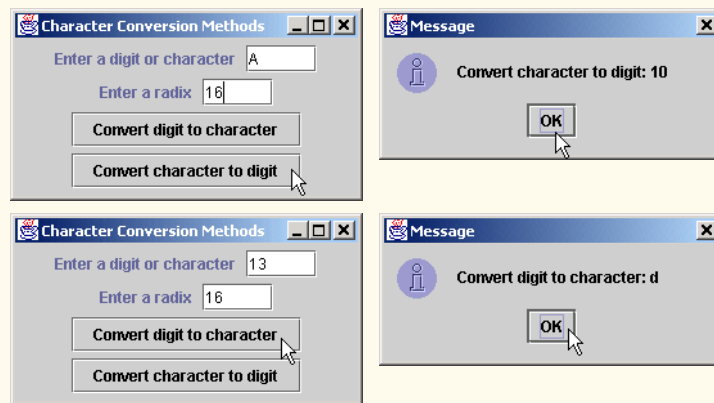


Fig. 10.18 **Character** class **static** conversion methods (part 3 of 3).

The program in Fig. 10.19 demonstrates the non**static** methods of class **Character**—the constructor, **charValue**, **toString**, **hashCode** and **equals**.

```

1 // Fig. 10.19: OtherCharMethods.java
2 // Demonstrate the non-static methods of class
3 // Character from the java.lang package.
4
5 // Java extension packages
6 import javax.swing.*;

```

Fig. 10.19 Non-**static** methods of class **Character** (part 1 of 2).

```

7
8 public class OtherCharMethods {
9
10     // test non-static Character methods
11     public static void main( String args[] )
12     {
13         Character c1, c2;
14
15         c1 = new Character( 'A' );
16         c2 = new Character( 'a' );
17
18         String output = "c1 = " + c1.charValue() +
19             "\nc2 = " + c2.toString() +
20             "\n\nhash code for c1 = " + c1.hashCode() +
21             "\nhash code for c2 = " + c2.hashCode();
22
23         if ( c1.equals( c2 ) )
24             output += "\n\nc1 and c2 are equal";
25         else
26             output += "\n\nc1 and c2 are not equal";
27
28         JOptionPane.showMessageDialog( null, output,
29             "Demonstrating Non-Static Character Methods",
30             JOptionPane.INFORMATION_MESSAGE );
31
32         System.exit( 0 );
33     }
34 } // end class OtherCharMethods
35

```

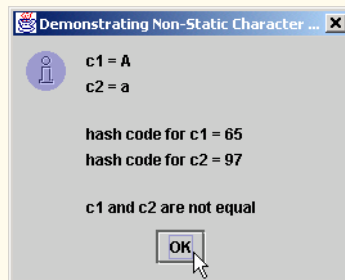


Fig. 10.19 Non-**static** methods of class **Character** (part 2 of 2).

Lines 15–16 instantiate two **Character** objects and pass character literals to the constructor to initialize those objects.

Line 18 uses **Character** method **charValue** to return the **char** value stored in **Character** object **c1**. Line 19 returns a **String** representation of **Character** object **c2** using method **toString**.

Lines 20–21 perform **hashCode** calculations on the **Character** objects **c1** and **c2**, respectively. Remember that hash code values are used to store objects in hash tables for fast lookup capabilities (see Chapter 20).

The condition in the **if** structure on line 23 uses method **equals** to determine whether the object **c1** has the same contents as the object **c2** (i.e., the characters inside each object are equal).

10.20 Class **StringTokenizer**

When you read a sentence, your mind breaks the sentence into individual words and punctuation, or *tokens*, each of which conveys meaning to you. Compilers also perform tokenization. They break up statements into individual pieces like keywords, identifiers, operators and other elements of a programming language. In this section, we study Java's **StringTokenizer** class (from package **java.util**), which breaks a string into its component tokens. Tokens are separated from one another by delimiters, typically white-space characters such as blank, tab, newline and carriage return. Other characters can also be used as delimiters to separate tokens. The program in Fig. 10.20 demonstrates class **StringTokenizer**. The window for class **TokenTest** displays a **JTextField** where the user types a sentence to tokenize. Output in this program is displayed in a **JTextArea**.

When the user presses the *Enter* key in the **JTextField**, method **actionPerformed** (lines 37–49) is invoked. Lines 39–40 assign **String** reference **stringToTokenize** the value in the text in the **JTextField** returned by calling **event.getActionCommand()**. Next, lines 41–42 create an instance of class **StringTokenizer**. This **StringTokenizer** constructor takes a **String** argument and creates a **StringTokenizer** for **stringToTokenize** that will use the default delimiter string "**\n\t\r**" consisting of a space, a newline, a tab and a carriage return for tokenization. There are two other constructors for class **StringTokenizer**. In the version that takes two **String** arguments, the second **String** is the delimiter **String**. In the version that takes three arguments, the second **String** is the delimiter **String** and the third argument (a **boolean**) determines whether the delimiters are also returned as tokens (only if the argument is **true**). This is useful if you need to know what the delimiters are.

```

1  // Fig. 10.20: TokenTest.java
2  // Testing the StringTokenizer class of the java.util package
3
4  // Java core packages
5  import java.util.*;
6  import java.awt.*;
7  import java.awt.event.*;
8
9  // Java extension packages
10 import javax.swing.*;
11
12 public class TokenTest extends JFrame {
13     private JLabel promptLabel;
14     private JTextField inputField;
15     private JTextArea outputArea;
16

```

Fig. 10.20 Tokenizing strings with a **StringTokenizer** object (part 1 of 3).

```

17 // set up GUI and event handling
18 public TokenTest()
19 {
20     super( "Testing Class StringTokenizer" );
21
22     Container container = getContentPane();
23     container.setLayout( new FlowLayout() );
24
25     promptLabel =
26         new JLabel( "Enter a sentence and press Enter" );
27     container.add( promptLabel );
28
29     inputField = new JTextField( 20 );
30
31     inputField.addActionListener(
32
33         // anonymous inner class
34         new ActionListener() {
35
36             // handle text field event
37             public void actionPerformed((ActionEvent event) )
38             {
39                 String stringToTokenize =
40                     event.getActionCommand();
41                 StringTokenizer tokens =
42                     new StringTokenizer( stringToTokenize );
43
44                 outputArea.setText( "Number of elements: " +
45                     tokens.countTokens() + "\nThe tokens are:\n" );
46
47                 while ( tokens.hasMoreTokens() )
48                     outputArea.append( tokens.nextToken() + "\n" );
49             }
50
51             } // end anonymous inner class
52
53     ); // end call to addActionListener
54
55     container.add( inputField );
56
57     outputArea = new JTextArea( 10, 20 );
58     outputArea.setEditable( false );
59     container.add( new JScrollPane( outputArea ) );
60
61     setSize( 275, 260 ); // set the window size
62     show();              // show the window
63 }
64
65 // execute application
66 public static void main( String args[] )
67 {
68     TokenTest application = new TokenTest();
69

```

Fig. 10.20 Tokenizing strings with a **StringTokenizer** object (part 2 of 3).

```

70     application.addWindowListener(
71
72         // anonymous inner class
73         new WindowAdapter() {
74
75             // handle event when user closes window
76             public void windowClosing( WindowEvent windowEvent )
77             {
78                 System.exit( 0 );
79             }
80
81         } // end anonymous inner class
82
83     ); // end call to addWindowListener
84
85 } // end method main
86
87 } // end class TokenTest

```

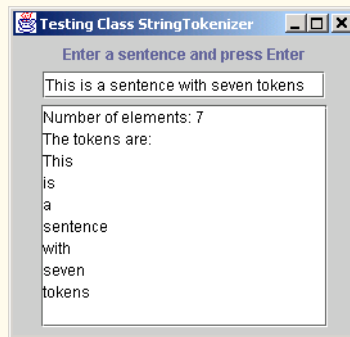


Fig. 10.20 Tokenizing strings with a **StringTokenizer** object (part 3 of 3).

The statement at lines 44–45 uses the **StringTokenizer** method **countTokens** to determine the number of tokens in the **String** to be tokenized.

The condition in the **while** structure at lines 47–48 **StringTokenizer** method **hasMoreTokens** to determine whether there are more tokens in the **String** being tokenized. If so, the **append** method is invoked for the **JTextArea** **outputArea** to append the next token to the **String** in the **JTextArea**. The next token is obtained with a call to **StringTokenizer** method **nextToken** that returns a **String**. The token is output followed by a newline character, so subsequent tokens appear on separate lines.

If you would like to change the delimiter **String** while tokenizing a **String**, you may do so by specifying a new delimiter string in a **nextToken** call as follows:

```
tokens.nextToken( newDelimiterString );
```

This feature is not demonstrated in the program.

10.21 Card Shuffling and Dealing Simulation

In this section, we use random number generation to develop a card shuffling and dealing simulation program. This program can then be used to implement programs that play specific card games.

We develop application **DeckOfCards** (Fig. 10.21), which creates a deck of 52 playing cards using **Card** objects, then enables the user to deal each card by clicking on a “**Deal card**” button. Each card dealt is displayed in a **JTextField**. The user can also shuffle the deck at any time by clicking on a “**Shuffle cards**” button.

```

1  // Fig. 10.21: DeckOfCards.java
2  // Card shuffling and dealing program
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7
8  // Java extension packages
9  import javax.swing.*;
10
11 public class DeckOfCards extends JFrame {
12     private Card deck[];
13     private int currentCard;
14     private JButton dealButton, shuffleButton;
15     private JTextField displayField;
16     private JLabel statusLabel;
17
18     // set up deck of cards and GUI
19     public DeckOfCards()
20     {
21         super( "Card Dealing Program" );
22
23         String faces[] = { "Ace", "Deuce", "Three", "Four",
24                             "Five", "Six", "Seven", "Eight", "Nine", "Ten",
25                             "Jack", "Queen", "King" };
26         String suits[] =
27             { "Hearts", "Diamonds", "Clubs", "Spades" };
28
29         deck = new Card[ 52 ];
30         currentCard = -1;
31
32         // populate deck with Card objects
33         for ( int count = 0; count < deck.length; count++ )
34             deck[ count ] = new Card( faces[ count % 13 ],
35                                       suits[ count / 13 ] );
36
37         // set up GUI and event handling
38         Container container = getContentPane();
39         container.setLayout( new FlowLayout() );
40
41         dealButton = new JButton( "Deal card" );

```

Fig. 10.21 Card dealing program (part 1 of 4).

```
42     dealButton.addActionListener(  
43  
44         // anonymous inner class  
45         new ActionListener() {  
46  
47             // deal one card  
48             public void actionPerformed( ActionEvent actionEvent )  
49             {  
50                 Card dealt = dealCard();  
51  
52                 if ( dealt != null ) {  
53                     displayField.setText( dealt.toString() );  
54                     statusLabel.setText( "Card #: " + currentCard );  
55                 }  
56                 else {  
57                     displayField.setText(  
58                         "NO MORE CARDS TO DEAL" );  
59                     statusLabel.setText(  
60                         "Shuffle cards to continue" );  
61                 }  
62             }  
63  
64         } // end anonymous inner class  
65  
66     ); // end call to addActionListener  
67  
68     container.add( dealButton );  
69  
70     shuffleButton = new JButton( "Shuffle cards" );  
71     shuffleButton.addActionListener(  
72  
73         // anonymous inner class  
74         new ActionListener() {  
75  
76             // shuffle deck  
77             public void actionPerformed( ActionEvent actionEvent )  
78             {  
79                 displayField.setText( "SHUFFLING ..." );  
80                 shuffle();  
81                 displayField.setText( "DECK IS SHUFFLED" );  
82             }  
83  
84         } // end anonymous inner class  
85  
86     ); // end call to addActionListener  
87  
88     container.add( shuffleButton );  
89  
90     displayField = new JTextField( 20 );  
91     displayField.setEditable( false );  
92     container.add( displayField );  
93  
94     statusLabel = new JLabel();
```

Fig. 10.21 Card dealing program (part 2 of 4).

```

95     container.add( statusLabel );
96
97     setSize( 275, 120 ); // set window size
98     show();             // show window
99 }
100
101 // shuffle deck of cards with one-pass algorithm
102 public void shuffle()
103 {
104     currentCard = -1;
105
106     // for each card, pick another random card and swap them
107     for ( int first = 0; first < deck.length; first++ ) {
108         int second = ( int ) ( Math.random() * 52 );
109         Card temp = deck[ first ];
110         deck[ first ] = deck[ second ];
111         deck[ second ] = temp;
112     }
113
114     dealButton.setEnabled( true );
115 }
116
117 // deal one card
118 public Card dealCard()
119 {
120     if ( ++currentCard < deck.length )
121         return deck[ currentCard ];
122     else {
123         dealButton.setEnabled( false );
124         return null;
125     }
126 }
127
128 // execute application
129 public static void main( String args[] )
130 {
131     DeckOfCards app = new DeckOfCards();
132
133     app.addWindowListener(
134
135         // anonymous inner class
136         new WindowAdapter() {
137
138             // terminate application when user closes window
139             public void windowClosing( WindowEvent windowEvent )
140             {
141                 System.exit( 0 );
142             }
143
144             } // end anonymous inner class
145
146     ); // end call to addWindowListener
147

```

Fig. 10.21 Card dealing program (part 3 of 4).

```

148     } // end method main
149
150 } // end class DeckOfCards
151
152 // class to represent a card
153 class Card {
154     private String face;
155     private String suit;
156
157     // constructor to initialize a card
158     public Card( String cardFace, String cardSuit )
159     {
160         face = cardFace;
161         suit = cardSuit;
162     }
163
164     // return String representation of Card
165     public String toString()
166     {
167         return face + " of " + suit;
168     }
169
170 } // end class Card

```

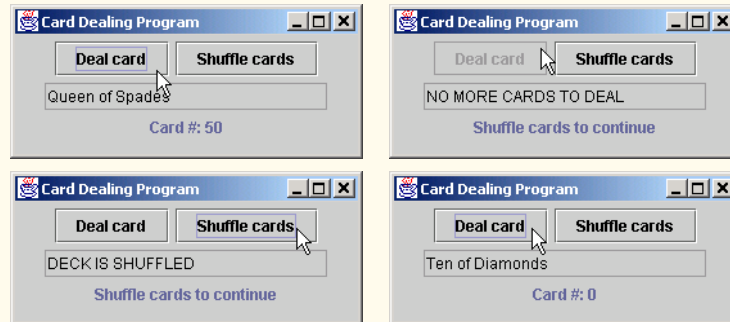


Fig. 10.21 Card dealing program (part 4 of 4).

Class **Card** (lines 153–170) contains two **String** instance variables—**face** and **suit**—that are used to store references to the face name and suit name for a specific **Card**. The constructor for the class receives two **Strings** that it uses to initialize **face** and **suit**. Method **toString** is provided to create a **String** consisting of the **face** of the card, the string " of " and the **suit** of the card.

Class **DeckOfCards** (lines 11–150) consists of an array **deck** of 52 **Cards**, an integer **currentCard** representing the most recently dealt card in the deck array (–1 if no cards have been dealt yet) and the GUI components used to manipulate the deck of cards. The constructor method of the application instantiates the **deck** array (line 29) and uses the **for** structure at lines 33–35 to fill the **deck** array with **Cards**. Note that each **Card** is instantiated and initialized with two **Strings**—one from the **faces** array (**Strings** "Ace" through "King") and one from the **suits** array ("Hearts", "Diamonds", "Clubs" and "Spades"). The calculation **count % 13** always results in a value from 0 to 12 (the

thirteen subscripts of the **faces** array), and the calculation **count / 13** always results in a value from 0 to 3 (the four subscripts in the **suits** array). When the **deck** array is initialized, it contains the cards with faces ace through king in order for each suit.

When the user clicks the **Deal card** button, method **actionPerformed** at lines 48–62 invokes method **dealCard** (defined at lines 118–126) to get the next card in the array. If the **deck** is not empty, a **Card** object reference is returned; otherwise, **null** is returned. If the reference is not **null**, lines 53–54 display the **Card** in the **JTextField displayField** and display the card number in the **JLabel statusLabel**. If the reference returned by **dealCard** was **null**, the **String** “NO MORE CARDS TO DEAL” is displayed in the **JTextField** and the **String** “Shuffle cards to continue” is displayed in the **JLabel**.

When the user clicks the **Shuffle cards** button, its **actionPerformed** method at lines 77–82 invokes method **shuffle** (defined on lines 102–115) to shuffle the cards. The method loops through all 52 cards (array subscripts 0 to 51). For each card, a number between 0 and 51 is picked randomly. Next, the current **Card** object and the randomly selected **Card** object are swapped in the array. A total of only 52 swaps are made in a single pass of the entire array, and the array of **Card** objects is shuffled! When the shuffling is complete, the **String** “DECK IS SHUFFLED” is displayed in the **JTextField**.

Notice the use of method **setEnabled** at lines 114 and 123 to activate and deactivate the **dealButton**. Method **setEnabled** can be used on many GUI components. When it is called with a **false** argument, the GUI component for which it is called is disabled so the user cannot interact with it. To reactivate the button, method **setEnabled** is called with a **true** argument.

10.22 (Optional Case Study) Thinking About Objects: Event Handling

Objects do not ordinarily perform their operations spontaneously. Rather, a specific operation is normally invoked when a sending object (a *client object*) sends a *message* to a receiving object (a *server object*) requesting that the receiving object perform that specific operation. In earlier sections, we mentioned that objects interact by sending and receiving messages. We began to model the behavior of our elevator system by using statechart and activity diagrams in Section 5.11 and collaboration diagrams in Section 7.10. In this section, we discuss how the objects of the elevator system interact.

Events

In Fig. 7.24, we presented an example of a person pressing a button by sending a **press-Button** message to the button—specifically, the **Person** object called method **press-Button** of the **Button** object. This message describes an action that is currently happening; in other words, the **Person** presses a **Button**. In general, the message name structure is a verb preceding a noun—e.g., the name of the **pressButton** message consists of the verb “press” followed by the noun “button.”

An *event* is a message that notifies an object of an action that has already happened. For example, in this section, we modify our simulation so the **Elevator** sends an **elevatorArrived** event to the **Elevator**’s **Door** when the **Elevator** arrives at a **Floor**. In Section 7.10, the **Elevator** opens this **Door** directly by sending an **openDoor** message. Listening for an **elevatorArrived** event allows the **Door** to determine the actions

to take when the **Elevator** has arrived, such as notifying the **Person** that the **Door** has opened. This reinforces the OOD principle of encapsulation and models the real world more closely. In reality, the door—not the elevator—“notifies” a person of a door’s opening.

Notice that the event-naming structure is the inverse of the first type of message’s naming structure. By convention, the event name consists of the noun preceding the verb. For instance, the **elevatorArrived** event name consists of the noun “elevator” preceding the verb “arrived.”

In our simulation, we create a superclass called **ElevatorModelEvent** (Fig. 10.23) that represents an event in our model. **ElevatorModelEvent** contains a **Location** reference (line 11) that represents the location where the event was generated and an **Object** reference (line 14) to the source of the event. In our simulation, objects use instances of **ElevatorModelEvent** to send events to other objects. When an object receives an event, that object may use method **getLocation** (lines 31–34) and method **getSource** (lines 43–46) to determine the event’s location and origin.

```

1  // ElevatorModelEvent.java
2  // Basic event packet holding Location object
3  package com.deitel.jhtp4.elevator.event;
4
5  // Deitel packages
6  import com.deitel.jhtp4.elevator.model.*;
7
8  public class ElevatorModelEvent {
9
10     // Location that generated ElevatorModelEvent
11     private Location location;
12
13     // source of generated ElevatorModelEvent
14     private Object source;
15
16     // ElevatorModelEvent constructor sets Location
17     public ElevatorModelEvent( Object source,
18                               Location location )
19     {
20         setSource( source );
21         setLocation( location );
22     }
23
24     // set ElevatorModelEvent Location
25     public void setLocation( Location eventLocation )
26     {
27         location = eventLocation;
28     }
29
30     // get ElevatorModelEvent Location
31     public Location getLocation()
32     {
33         return location;
34     }

```

Fig. 10.23 Class **ElevatorModelEvent** is the superclass for all other event classes in our model (part 1 of 2).

```

35
36     // set ElevatorModelEvent source
37     private void setSource( Object eventSource )
38     {
39         source = eventSource;
40     }
41
42     // get ElevatorModelEvent source
43     public Object getSource()
44     {
45         return source;
46     }
47 }

```

Fig. 10.23 Class **ElevatorModelEvent** is the superclass for all other event classes in our model (part 2 of 2).

For example, a **Door** may send an **ElevatorModelEvent** to a **Person** when opening or closing, and the **Elevator** may send an **ElevatorModelEvent** informing a person of a departure or arrival. Having different objects send the same event type to describe different actions could be confusing. To eliminate ambiguity as we discuss what events are sent by objects, we create several **ElevatorModelEvent** subclasses in Fig. 10.24, so we will have an easier time associating each event with its sender. According to Fig. 10.24, classes **BellEvent**, **PersonMoveEvent**, **LightEvent**, **ButtonEvent**, **ElevatorMoveEvent** and **DoorEvent** are subclasses of class **ElevatorModelEvent**. Using these event subclasses, a **Door** sends a different event (a **DoorEvent**) than does a **Button** (which sends a **ButtonEvent**). Figure 10.25 displays the triggering actions of the subclass events. Note that all actions in Fig. 10.25 appear in the form “noun” + “verb”.

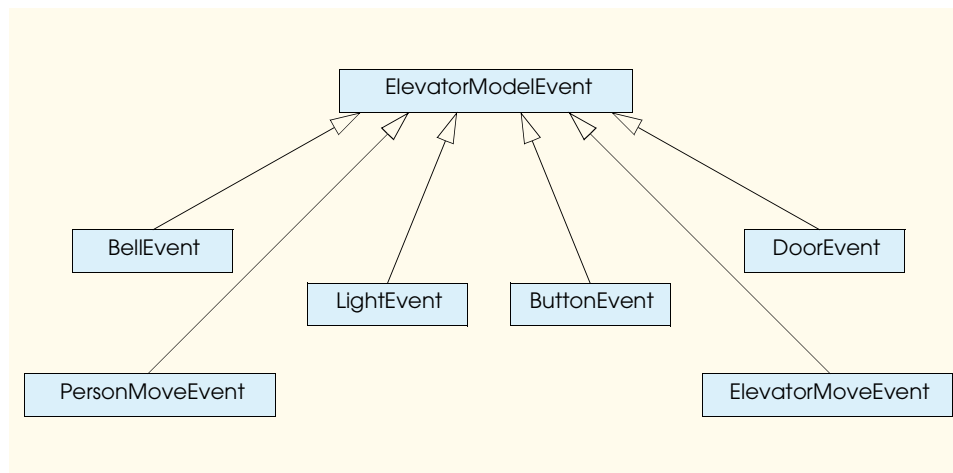


Fig. 10.24 Class diagram that models the generalization between **ElevatorModelEvent** and its subclasses.

Event	Sent when (triggering action)	Sent by object of class
BellEvent	the Bell has rung	Bell
ButtonEvent	a Button has been pressed a Button has been reset	Button Button
DoorEvent	a Door has opened a Door has closed	Door Door
LightEvent	a Light has turned on a Light has turned off	Light
PersonMoveEvent	a Person has been created a Person has arrived at the Elevator a Person has entered the Elevator a Person has exited the Elevator a Person has pressed a Button a Person has exited the simulation	Person
ElevatorMoveEvent	the Elevator has arrived at a Floor the Elevator has departed from a Floor	Elevator

Fig. 10.25 Triggering actions of the **ElevatorModelEvent** subclass events.

Event Handling

The concept of event handling in Java is similar to the concept of a *collaboration* described in Section 7.10. Event handling consists of an object of one class sending a particular message (which Java calls an *event*) to objects of other classes *listening for that type of message*.¹ The difference is that the objects receiving the message must *register* to receive the message; therefore, event handling describes *how* an object sends an event to other objects “listening” for that type of event—these objects are called *event listeners*. To send an event, the sending object invokes a particular method of the receiving object while passing the desired event object as a parameter. In our simulation, this event object belongs to a class that extends **ElevatorModelEvent**.

We presented a collaboration diagram in Fig. 7.25 showing interactions of two **Person** objects—**waitingPassenger** and **ridingPassenger**—as they enter and exit the **Elevator**. Figure 10.26 shows a modified diagram that incorporates event handling. There are three differences between the diagrams. First, we provide *notes*—explanatory remarks about some of the graphics in the diagram. The UML represents notes as rectangles with the upper right corners “folded over.” Notes in the UML are similar to *comments* in Java. A dotted line associates a note with any component of the UML (object, class, arrow, etc.). In this diagram, the **<<parameter>>** notation specifies that the note contains the parameters of a given message: all **doorOpened** events pass a **DoorEvent** object as a parameter; all **elevatorArrived** events pass an **ElevatorMoveEvent** object; all **openDoor** messages pass a **Location** object.

1. Technically, one object sends a notification of an event—or some triggering action—to another object. However, Java parlance refers to sending this notification as “sending an event.”

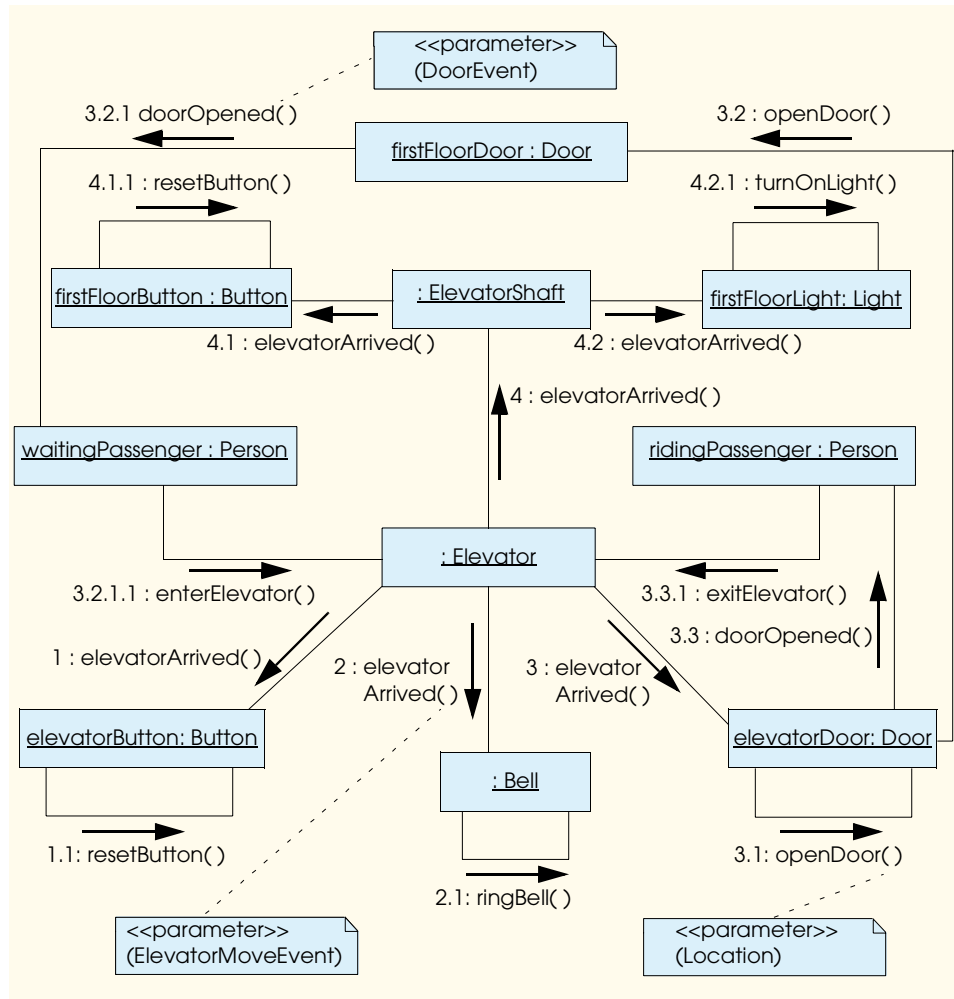


Fig. 10.26 Modified collaboration diagram for passengers entering and exiting the **Elevator** on the first **Floor**.

The second difference between the diagrams is that the interactions of Fig. 10.26 occur on the first **Floor**. This allows us to name all **Button** and **Door** objects (**firstFloorDoor** and **firstFloorButton**) to eliminate ambiguity, because the **Button** and **Door** classes each have three objects in our simulation. The interactions that occur on the second **Floor** are identical to the ones that occur on the first **Floor**.

The most substantial difference between Fig. 10.26 and Fig. 7.25 is that the **Elevator** informs objects (via an event) of an action *that has already happened*—the **Elevator** *has arrived*. The objects that receive the event then perform some action in response to the type of message they receive.

According to messages **1**, **2**, **3** and **4**, the **Elevator** performs only one action—it sends **elevatorArrived** events to objects interested in receiving those events. Specif-

ically, the **Elevator** object sends an **ElevatorMoveEvent** using the receiving object's **elevatorArrived** method. Figure 10.26 begins with the **Elevator** sending an **elevatorArrived** event to the **elevatorButton**. The **elevatorButton** then *resets itself* (message 1.1). The **Elevator** then sends an **elevatorArrived** event to the **Bell** (message 2), and the **Bell** invokes its **ringBell** method, accordingly (i.e., the **Bell** object sends *itself* a **ringBell** message in message 2.1).

The **Elevator** sends an **elevatorArrived** message to the **elevatorDoor** (message 3). The **elevatorDoor** then opens itself by invoking its **openDoor** method (message 3.1). At this point, the **elevatorDoor** is open but has not informed the **ridingPassenger** of opening. Before informing the **ridingPassenger**, the **elevatorDoor** opens the **firstFloorDoor** by sending an **openDoor** message to the **firstFloorDoor** (message 3.2)—this guarantees that the **ridingPassenger** will not exit before the **firstFloorDoor** opens. The **firstFloorDoor** then informs the **waitingPassenger** that the **firstFloorDoor** has opened (message 3.2.1), and the **waitingPassenger** enters the **Elevator** (message 3.2.1.1). All messages nested in 3.2 have been passed, so the **elevatorDoor** may inform the **ridingPassenger** that **elevatorDoor** has opened by invoking method **doorOpened** of the **ridingPassenger** (message 3.3). The **ridingPassenger** responds by exiting the **Elevator** (message 3.3.1).²

Lastly, the **Elevator** informs the **ElevatorShaft** of the arrival (message 4). The **ElevatorShaft** then informs the **firstFloorButton** of the arrival (message 4.1), and the **firstFloorButton** resets itself (message 4.1.1). The **ElevatorShaft** then informs the **firstFloorLight** of the arrival (message 4.2), and the **firstFloorLight** illuminates itself (message 4.2.1).

Event Listeners

We demonstrated event handling between the **Elevator** and object **elevatorDoor** using the modified collaboration diagram of Fig. 10.26—the **Elevator** sends an **elevatorArrived** event to the **elevatorDoor** (message 3). We first must determine the event object that the **Elevator** will pass to the **elevatorDoor**. According to the note in the lower left-hand corner of Fig. 10.26, the **Elevator** passes an **ElevatorMoveEvent** (Fig. 10.27) object when the **Elevator** invokes an **elevatorArrived** method. The generalization diagram of Fig. 10.24 indicates that **ElevatorMoveEvent** is a subclass of **ElevatorModelEvent**, so **ElevatorMoveEvent** inherits the **Object** and **Location** references from **ElevatorModelEvent**.³

2. The problem of the **waitingPassenger** entering the **Elevator** (message 3.2.1.1) before the **ridingPassenger** has exited (message 3.3.1) remains in our collaboration diagram. We show in “Thinking About Objects” Section 15.12 how to solve this problem by using multithreading, synchronization and active classes.
3. In our simulation, all event classes have this structure—that is, the structure of class **ElevatorMoveEvent** is identical to the structure of class **DoorEvent**, **ButtonEvent**, etc. When you have finished reading the material in this section, we recommend that you view the implementation of the events in Appendix G to attain a better comprehension of the structure of our system events—Fig. G.1–G.7 present the code for the events, and Fig. G.8–G.14 present the code for the event listeners.

```

1  // ElevatorMoveEvent.java
2  // Indicates on which Floor the Elevator arrived or departed
3  package com.deitel.jhtp4.elevator.event;
4
5  // Deitel package
6  import com.deitel.jhtp4.elevator.model.*;
7
8  public class ElevatorMoveEvent extends ElevatorModelEvent {
9
10     // ElevatorMoveEvent constructor
11     public ElevatorMoveEvent( Object source, Location location )
12     {
13         super( source, location );
14     }
15 }

```

Fig. 10.27 Class **ElevatorMoveEvent**, a subclass of **ElevatorModelEvent**, is sent when the **Elevator** has arrived at or departed from, a **Floor**.

The **elevatorDoor** must implement an interface that “listens” for an **ElevatorMoveEvent**—this makes the **elevatorDoor** an event listener. Interface **ElevatorMoveListener** (Fig. 10.28) provides methods **elevatorDeparted** (line 8) and **elevatorArrived** (line 11) that enable the **Elevator** to notify the **ElevatorMoveListener** when the **Elevator** has arrived or departed. An interface that provides the methods for an event listener, such as **ElevatorMoveListener**, is called an *event listener interface*.

Methods **elevatorArrived** and **elevatorDeparted** each receive an **ElevatorMoveEvent** (Fig. 10.27) object as an argument. Therefore, when the **Elevator** “sends an **elevatorArrived** event” to another object, the **Elevator** passes an **ElevatorMoveEvent** object as an argument to the receiving object’s **elevatorArrived** method. We implement class **Door**—the class of which the **elevatorDoor** is an instance—in Appendix H, after we continue refining our design and learning more Java capabilities.

```

1  // ElevatorMoveListener.java
2  // Methods invoked when Elevator has either departed or arrived
3  package com.deitel.jhtp4.elevator.event;
4
5  public interface ElevatorMoveListener {
6
7     // invoked when Elevator has departed
8     public void elevatorDeparted( ElevatorMoveEvent moveEvent );
9
10    // invoked when Elevator has arrived
11    public void elevatorArrived( ElevatorMoveEvent moveEvent );
12 }

```

Fig. 10.28 Interface **ElevatorMoveListener** provides the methods required to listen for **Elevator** departure and arrival events.

Class Diagram Revisited

Figure 10.29 modifies the associations in the class diagram of Fig. 9.19 to include event handling. Note that like the collaboration diagram of Fig. 10.26, Fig. 10.29 indicates that an object informs, or *signals*, another object that some event has occurred. If an object receiving the event invokes a **private** method, the class diagram represents this method invocation as a *self association*—that is, the class contains an association with itself. The **Button**, **Door**, **Light** and **Bell** classes contain self associations; note that the association does not include an arrowhead indicating the direction of the association, because the class's association is with itself. Lastly, the diagram includes an association between class **Door** and class **Person** (the **Door** informs a **Person** that the **Door** has opened), because we established the relationship between all **Door** objects and a **Person** object.

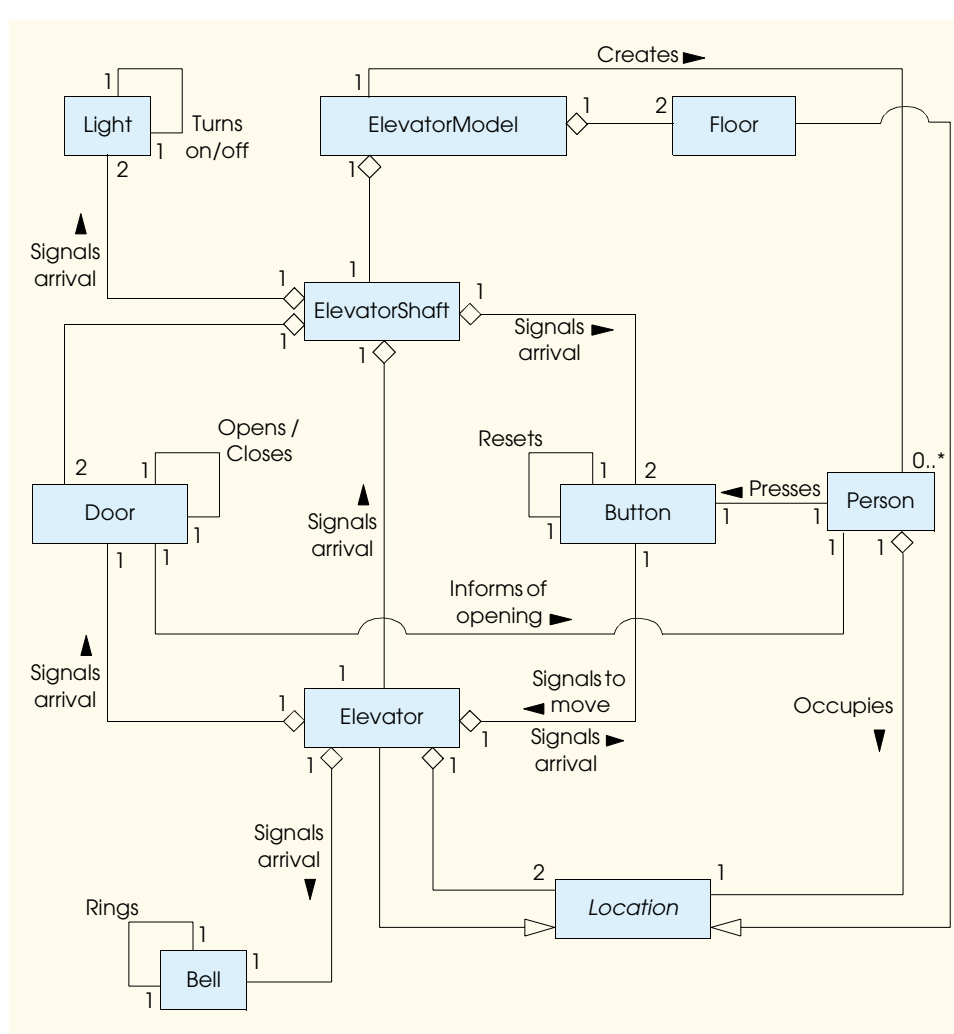


Fig. 10.29 Class diagram of our simulator (including event handling).

SUMMARY

- A character constant's value is its integer value in the Unicode character set. Strings can include letters, digits and special characters such as **+**, **-**, *****, **/** and **\$**. A string in Java is an object of class **String**. String literals or string constants are often referred to as *anonymous String objects* and are written in double quotes in a program.
- Class **String** provides nine constructors.
- **String** method **length** returns the number of characters in a **String**.
- **String** method **charAt** returns the character at a specific position.
- Method **equals** is used to test any two objects for equality (i.e., the contents of the two objects are identical). The method returns **true** if the objects are equal, **false** otherwise. Method **equals** uses a *lexicographical comparison* for **Strings**.
- When primitive-data type values are compared with **==**, the result is **true** if both values are identical. When references are compared with **==**, the result is **true** if both references refer to *the same object in memory*.
- Java treats all anonymous **Strings** with the same contents as one anonymous **String** object.
- **String** method **equalsIgnoreCase** performs a case-insensitive **String** comparison.
- **String** method **compareTo** returns 0 if the **Strings** it is comparing are equal, a negative number if the **String** that invokes **compareTo** is less than the **String** that is passed as an argument and a positive number if the **String** that invokes **compareTo** is greater than the **String** that is passed as an argument. Method **compareTo** uses a lexicographical comparison.
- **String** method **regionMatches** compares portions of two **Strings** for equality.
- **String** method **startsWith** determines whether a **String** starts with the characters specified as an argument. **String** method **endsWith** determines whether a **String** ends with the characters specified as an argument.
- Method **hashCode** performs a hash code calculation that enables a **String** object to be stored in a hash table. This method is inherited from **Object** and overridden by **String**.
- **String** method **indexOf** locates the first occurrence of a character or a substring in a **String**. Method **lastIndexOf** locates the last occurrence of a character or a substring in a **String**.
- **String** method **substring** copies and returns part of an existing **String** object.
- **String** method **concat** concatenates two **String** objects and returns a new **String** object containing the characters from both original **Strings**.
- **String** method **replace** returns a new **String** object that replaces every occurrence in a **String** of its first character argument with its second character argument.
- **String** method **toUpperCase** returns a new **String** with uppercase letters in the positions where the original **String** had lowercase letters. Method **toLowerCase** returns a new **String** with lowercase letters in the positions where the original **String** had uppercase letters.
- **String** method **trim** returns a new **String** object in which all white-space characters (such as spaces, newlines and tabs) have been removed from the beginning or end of a **String**.
- **String** method **toCharArray** returns a new character array containing a copy of the characters in a **String**.
- **String** class method **valueOf** returns its argument converted to a string.
- The first time **String** method **intern** is invoked on a **String** it returns a reference to that **String** object. Subsequent invocations of **intern** on different **String** objects that have the same contents as the original **String** result in multiple references to the original **String** object.

- Class **StringBuffer** provides three constructors that enable **StringBuffers** to be initialized with no characters and an initial capacity of 16 characters; with no characters and an initial capacity specified in the integer argument or with a copy of the characters of the **String** argument and an initial capacity that is the number of characters in the **String** argument plus 16.
- **StringBuffer** method **length** returns the number of characters currently stored in a **StringBuffer**. Method **capacity** returns the number of characters that can be stored in a **StringBuffer** without allocating more memory.
- Method **ensureCapacity** ensures that a **StringBuffer** has a minimum capacity. Method **setLength** increases or decreases the length of a **StringBuffer**.
- **StringBuffer** method **charAt** returns the character at the specified index. Method **setCharAt** sets the character at the specified position. Method **getChars** returns a character array containing a copy of the characters in the **StringBuffer**.
- Class **StringBuffer** provides overloaded **append** methods to add primitive data-type, character array, **String** and **Object** values to the end of a **StringBuffer**.
- **StringBuffers** and the **append** methods are used by the Java compiler to implement the **+** and **+=** operators for concatenating **Strings**.
- Class **StringBuffer** provides overloaded **insert** methods to insert primitive data-type, character array, **String** and **Object** values at any position in a **StringBuffer**.
- Class **Character** provides a constructor that takes a character argument.
- **Character** method **isDefined** determines whether a character is defined in the Unicode character set. If so, the method returns **true**; otherwise, it returns **false**.
- **Character** method **isDigit** determines whether a character is a defined Unicode digit. If so, the method returns **true**; otherwise, it returns **false**.
- **Character** method **isJavaIdentifierStart** determines whether a character is a character that can be used as the first character of an identifier in Java [i.e., a letter, an underscore (**_**) or a dollar sign (**\$**)]. If so, the method returns **true**; otherwise, it returns **false**.
- **Character** method **isJavaIdentifierPart** determines whether a character is a character that can be used in an identifier in Java [i.e., a digit, a letter, an underscore (**_**) or a dollar sign (**\$**)]. If so, the method returns **true**; otherwise, it returns **false**. Method **isLetter** determines whether a character is a letter. If so, the method returns **true**; otherwise, it returns **false**. Method **isLetterOrDigit** determines whether a character is a letter or a digit. If so, the method returns **true**; otherwise, it returns **false**.
- **Character** method **isLowerCase** determines whether a character is a lowercase letter. If so, the method returns **true**; otherwise, **false**. **Character** method **isUpperCase** determines if a character is an uppercase letter. If so, the method returns **true**; otherwise, **false**.
- **Character** method **toUpperCase** converts a character to its uppercase equivalent. Method **toLowerCase** converts a character to its lowercase equivalent.
- **Character** method **digit** converts its character argument into an integer in the number system specified by its integer argument **radix**. Method **forDigit** converts its integer argument **digit** into a character in the number system specified by its integer argument **radix**.
- **Character** method **charValue** returns the **char** stored in a **Character** object. Method **toString** returns a **String** representation of a **Character**.
- **Character** method **hashCode** performs a hash code calculation on a **Character**.
- **StringTokenizer**'s default constructor creates a **StringTokenizer** for its **String** argument that will use the default delimiter string "**\n\t\r**", consisting of a space, a newline, a tab and a carriage return for tokenization.

- **StringTokenizer** method **countTokens** returns the number of tokens in a **String** to be tokenized.
- **StringTokenizer** method **hasMoreTokens** determines if there are more tokens in the **String** being tokenized.
- **StringTokenizer** method **nextToken** returns a **String** with the next token.

TERMINOLOGY

append method of class **StringBuffer**

appending strings to other strings

array of strings

capacity method of class **StringBuffer**

Character class

character code

character constant

character set

charAt method of class **String**

charAt method of class **StringBuffer**

charValue method of class **Character**

compareTo method of class **String**

comparing strings

concat method of class **String**

concatenation

copying strings

countTokens method (of **StringTokenizer**)

delimiter

digit method of class **Character**

endsWith method of class **String**

equals method of class **String**

equalsIgnoreCase method of **String**

forDigit method of class **Character**

getChars method of class **String**

getChars method of class **StringBuffer**

hash table

hashCode method of class **Character**

hashCode method of class **String**

hasMoreTokens method

hexadecimal digits

indexOf method of class **String**

insert method of class **StringBuffer**

intern method of class **String**

isDefined method of class **Character**

isDigit method of class **Character**

isJavaIdentifierPart method

isJavaIdentifierStart method

isLetter method of class **Character**

isLetterOrDigit method of class **Character**

isLowerCase method of class **Character**

isUpperCase method of class **Character**

lastIndexOf method of class **String**

length method of class **String**

length method of class **StringBuffer**

length of a string

literal

nextToken method of **StringTokenizer**

numeric code representation of a character

printing character

regionMatches method of class **String**

replace method of class **String**

search string

setCharAt method of class **StringBuffer**

startsWith method of class **String**

string

String class

string concatenation

string constant

string literal

string processing

StringBuffer class

StringIndexOutOfBoundsException

StringTokenizer class

substring method of **String** class

toCharArray method of class **String**

token

tokenizing strings

toLowerCase method of class **Character**

toLowerCase method of class **String**

toString method of class **Character**

toString method of class **String**

toString method of class **StringBuffer**

toUpperCase method of class **Character**

toUpperCase method of class **String**

trim method of class **String**

Unicode

valueOf method of class **String**

white-space characters

word processing

SELF-REVIEW EXERCISES

- 10.1** State whether each of the following is *true* or *false*. If *false*, explain why.
- When **String** objects are compared with **==**, the result is **true** if the **Strings** contain the same values.
 - A **String** can be modified after it is created.
- 10.2** For each of the following, write a single statement that performs the indicated task.
- Compare the string in **s1** to the string in **s2** for equality of contents.
 - Append the string **s2** to the string **s1**, using **+=**.
 - Determine the length of the string in **s1**.

ANSWERS TO SELF-REVIEW EXERCISES

- 10.1**
- False. **String** objects that are compared with operator **==** are actually compared to determine if they are the same object in memory.
 - False. **String** objects are constant and cannot be modified after they are created. **StringBuffer** objects can be modified after they are created.
- 10.2**
- s1.equals(s2)**
 - s1 += s2;**
 - s1.length()**

EXERCISES

Exercises 10.3 through 10.6 are reasonably challenging. Once you have done these problems, you ought to be able to implement most popular card games easily.

- 10.3** Modify the program in Fig. 10.21 so that the card-dealing method deals a five-card poker hand. Then write the following additional methods:
- Determine if the hand contains a pair.
 - Determine if the hand contains two pairs.
 - Determine if the hand contains three of a kind (e.g., three jacks).
 - Determine if the hand contains four of a kind (e.g., four aces).
 - Determine if the hand contains a flush (i.e., all five cards of the same suit).
 - Determine if the hand contains a straight (i.e., five cards of consecutive face values).
 - Determine if the hand contains a full house (i.e., two cards of one face value and three cards of another face value).
- 10.4** Use the methods developed in Exercise 10.3 to write a program that deals two five-card poker hands, evaluates each hand and determines which is the better hand.
- 10.5** Modify the program developed in Exercise 10.4 so that it can simulate the dealer. The dealer's five-card hand is dealt "face down" so the player cannot see it. The program should then evaluate the dealer's hand and, based on the quality of the hand, the dealer should draw one, two or three more cards to replace the corresponding number of unneeded cards in the original hand. The program should then reevaluate the dealer's hand. (*Caution:* This is a difficult problem!)
- 10.6** Modify the program developed in Exercise 10.5 so that it can handle the dealer's hand automatically, but the player is allowed to decide which cards of the player's hand to replace. The program should then evaluate both hands and determine who wins. Now, use this new program to play 20 games against the computer. Who wins more games, you or the computer? Have one of your friends play 20 games against the computer. Who wins more games? Based on the results of these games, make appropriate modifications to refine your poker-playing program. (This, too, is a difficult problem.) Play 20 more games. Does your modified program play a better game?

10.7 Write an application that uses **String** method **compareTo** to compare two strings input by the user. Output whether the first string is less than, equal to or greater than the second.

10.8 Write an application that uses **String** method **regionMatches** to compare two strings input by the user. The program should input the number of characters to be compared and the starting index of the comparison. The program should state whether the first string is less than, equal to or greater than the second string. Ignore the case of the characters when performing the comparison.

10.9 Write an application that uses random number generation to create sentences. Use four arrays of strings called **article**, **noun**, **verb** and **preposition**. Create a sentence by selecting a word at random from each array in the following order: **article**, **noun**, **verb**, **preposition**, **article** and **noun**. As each word is picked, concatenate it to the previous words in the sentence. The words should be separated by spaces. When the final sentence is output, it should start with a capital letter and end with a period. The program should generate 20 sentences and output them to a text area.

The arrays should be filled as follows: The article array should contain the articles "**the**", "**a**", "**one**", "**some**" and "**any**"; the noun array should contain the nouns "**boy**", "**girl**", "**dog**", "**town**" and "**car**"; the verb array should contain the verbs "**drove**", "**jumped**", "**ran**", "**walked**" and "**skipped**"; the preposition array should contain the prepositions "**to**", "**from**", "**over**", "**under**" and "**on**".

After the preceding program is written, modify the program to produce a short story consisting of several of these sentences. (How about the possibility of a random term paper writer!)

10.10 (*Limericks*) A limerick is a humorous five-line verse in which the first and second lines rhyme with the fifth, and the third line rhymes with the fourth. Using techniques similar to those developed in Exercise 10.9, write a Java program that produces random limericks. Polishing this program to produce good limericks is a challenging problem, but the result will be worth the effort!

10.11 (*Pig Latin*) Write an application that encodes English language phrases into pig Latin. Pig Latin is a form of coded language often used for amusement. Many variations exist in the methods used to form pig Latin phrases. For simplicity, use the following algorithm:

To form a pig Latin phrase from an English language phrase, tokenize the phrase into words with an object of class **StringTokenizer**. To translate each English word into a pig Latin word, place the first letter of the English word at the end of the word and add the letters "ay." Thus, the word "jump" becomes "umpjay," the word "the" becomes "hetay," and the word "computer" becomes "omputercay." Blanks between words remain as blanks. Assume the following: The English phrase consists of words separated by blanks, there are no punctuation marks and all words have two or more letters. Method **printLatinWord** should display each word. Each token returned from **nextToken** is passed to method **printLatinWord** to print the pig Latin word. Enable the user to input the sentence. Keep a running display of all the converted sentences in a text area.

10.12 Write an application that inputs a telephone number as a string in the form **(555) 555-5555**. The program should use an object of class **StringTokenizer** to extract the area code as a token, the first three digits of the phone number as a token and the last four digits of the phone number as a token. The seven digits of the phone number should be concatenated into one string. The program should convert the area code string to **int** (remember **parseInt**!) and convert the phone number string to **long**. Both the area code and the phone number should be printed. Remember that you will have to change delimiter characters during the tokenization process.

10.13 Write an application that inputs a line of text, tokenizes the line with an object of class **StringTokenizer** and outputs the tokens in reverse order.

10.14 Use the string comparison methods discussed and the techniques for sorting arrays developed in Chapter 7 to write a program that alphabetizes a list of strings. Allow the user to enter the strings in a text field. Display the results in a text area.

- 10.15** Write an application that inputs text and outputs the text in uppercase and lowercase letters.
- 10.16** Write an application that inputs several lines of text and a search character and uses method **String** method **indexOf** to determine the number of occurrences of the character in the text.
- 10.17** Write an application based on the program in Exercise 10.16 that inputs several lines of text and uses **String** method **indexOf** to determine the total number of occurrences of each letter of the alphabet in the text. Uppercase and lowercase letters should be counted together. Store the totals for each letter in an array and print the values in tabular format after the totals have been determined.
- 10.18** Write an application that reads a series of strings and outputs only those strings beginning with the letter “b.” The results should be output to a text area.
- 10.19** Write an application that reads a series of strings and prints only those strings ending with the letters “ED.” The results should be output to a text area.
- 10.20** Write an application that inputs an integer code for a character and displays the corresponding character. Modify this program so that it generates all possible three-digit codes in the range form 000 to 255 and attempts to print the corresponding characters. Display the results in a text area.
- 10.21** Write your own versions of the **String** methods for searching strings.
- 10.22** Write a program that reads a five-letter word from the user and produces all possible three-letter words that can be derived from the letters of the five-letter word. For example, the three-letter words produced from the word “bathe” include the commonly used words “ate,” “bat,” “bet,” “tab,” “hat,” “the” and “tea.”

SPECIAL SECTION: ADVANCED STRING MANIPULATION EXERCISES

The preceding exercises are keyed to the text and designed to test the reader’s understanding of fundamental string-manipulation concepts. This section includes a collection of intermediate and advanced string-manipulation exercises. The reader should find these problems challenging, yet entertaining. The problems vary considerably in difficulty. Some require an hour or two of program writing and implementation. Others are useful for lab assignments that might require two or three weeks of study and implementation. Some are challenging term projects.

10.23 (*Text Analysis*) The availability of computers with string-manipulation capabilities has resulted in some rather interesting approaches to analyzing the writings of great authors. Much attention has been focused on whether William Shakespeare ever lived. Some scholars believe there is substantial evidence indicating that Christopher Marlowe or other authors actually penned the masterpieces attributed to Shakespeare. Researchers have used computers to find similarities in the writings of these two authors. This exercise examines three methods for analyzing texts with a computer.

- a) Write an application that reads several lines of text from the keyboard and prints a table indicating the number of occurrences of each letter of the alphabet in the text. For example, the phrase

To be, or not to be: that is the question:

contains one “a,” two “b’s,” no “c’s,” etc.

- b) Write an application that reads several lines of text and prints a table indicating the number of one-letter words, two-letter words, three-letter words, etc. appearing in the text. For example, Fig. 10.30 shows the counts for the phrase

Whether 'tis nobler in the mind to suffer

Word length	Occurrences
1	0
2	2
3	1
4	2 (including 'tis)
5	0
6	2
7	1

Fig. 10.30 Counts for the string "Whether 'tis nobler in the mind to suffer".

- c) Write an application that reads several lines of text and prints a table indicating the number of occurrences of each different word in the text. The first version of your program should include the words in the table in the same order in which they appear in the text. For example, the lines

**To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer**

contain the words "to" three times, the word "be" two times, the word "or" once, etc. A more interesting (and useful) printout should then be attempted in which the words are sorted alphabetically.

10.24 (*Printing Dates in Various Formats*) Dates are printed in several common formats. Two of the more common formats are

04/25/1955 and April 25, 1955

Write an application that reads a date in the first format and prints that date in the second format.

10.25 (*Check Protection*) Computers are frequently employed in check-writing systems such as payroll and accounts payable applications. Many strange stories circulate regarding weekly paychecks being printed (by mistake) for amounts in excess of \$1 million. Incorrect amounts are printed by computerized check-writing systems because of human error and/or machine failure. Systems designers build controls into their systems to prevent such erroneous checks from being issued.

Another serious problem is the intentional alteration of a check amount by someone who intends to cash a check fraudulently. To prevent a dollar amount from being altered, most computerized check-writing systems employ a technique called *check protection*.

Checks designed for imprinting by computer contain a fixed number of spaces in which the computer may print an amount. Suppose a paycheck contains eight blank spaces in which the computer is supposed to print the amount of a weekly paycheck. If the amount is large, then all eight of those spaces will be filled, for example:

1,230.60 (*check amount*)

12345678 (*position numbers*)

On the other hand, if the amount is less than \$1000, then several of the spaces would ordinarily be left blank. For example,

```

    99.87
-----
12345678

```

contains three blank spaces. If a check is printed with blank spaces, it is easier for someone to alter the amount of the check. To prevent a check from being altered, many check-writing systems insert *leading asterisks* to protect the amount as follows:

```

***99.87
-----
12345678

```

Write an application that inputs a dollar amount to be printed on a check, then prints the amount in check-protected format with leading asterisks if necessary. Assume that nine spaces are available for printing the amount.

10.26 (*Writing the Word Equivalent of a Check Amount*) Continuing the discussion of the previous exercise, we reiterate the importance of designing check-writing systems to prevent alteration of check amounts. One common security method requires that the check amount be written both in numbers and “spelled out” in words as well. Even if someone is able to alter the numerical amount of the check, it is extremely difficult to change the amount in words.

- Many computerized check-writing systems do not print the amount of the check in words. Perhaps the main reason for this omission is the fact that most high-level languages used in commercial applications do not contain adequate string-manipulation features. Another reason is that the logic for writing word equivalents of check amounts is somewhat involved.
- Write an application that inputs a numeric check amount and writes the word equivalent of the amount. For example, the amount 112.43 should be written as

ONE HUNDRED TWELVE and 43/100

10.27 (*Morse Code*) Perhaps the most famous of all coding schemes is the Morse code, developed by Samuel Morse in 1832 for use with the telegraph system. The Morse code assigns a series of dots and dashes to each letter of the alphabet, each digit, and a few special characters (such as period, comma, colon and semicolon). In sound-oriented systems, the dot represents a short sound and the dash represents a long sound. Other representations of dots and dashes are used with light-oriented systems and signal-flag systems.

Separation between words is indicated by a space, or, quite simply, the absence of a dot or dash. In a sound-oriented system, a space is indicated by a short period of time during which no sound is transmitted. The international version of the Morse code appears in Fig. 10.31.

Write an application that reads an English language phrase and encodes the phrase into Morse code. Also write a program that reads a phrase in Morse code and converts the phrase into the English language equivalent. Use one blank between each Morse-coded letter and three blanks between each Morse-coded word.

10.28 (*A Metric Conversion Program*) Write an application that will assist the user with metric conversions. Your program should allow the user to specify the names of the units as strings (i.e., centimeters, liters, grams, etc. for the metric system and inches, quarts, pounds, etc. for the English system) and should respond to simple questions such as

```

"How many inches are in 2 meters?"
"How many liters are in 10 quarts?"

```

Character	Code	Character	Code
A	.-	T	-
B	-...	U	..-
C	-.-. .	V	...-
D	-..	W	.-.-
E	.	X	-..-
F	..-. .	Y	-.--
G	--.	Z	--..
H			
I	..	Digits	
J	.---	1	.----
K	-.-	2	..---
L	.-..	3	...--
M	--	4	-
N	-. .	5	
O	---	6	-....
P		7	--...
Q	--.-	8	-----
R	.-. .	9	-----
S		0	-----

Fig. 10.31 The letters of the alphabet as expressed in international Morse code .

Your program should recognize invalid conversions. For example, the question

"How many feet in 5 kilograms?"

is not a meaningful question because **"feet"** is a unit of length while **"kilograms"** is a unit of mass.

SPECIAL SECTION: CHALLENGING STRING MANIPULATION PROJECTS

10.29 (*Project: A Spelling Checker*) Many popular word processing software packages have built-in spell checkers.

In this project, you are asked to develop your own spell-checker utility. We make suggestions to help get you started. You should then consider adding more capabilities. Use a computerized dictionary (if you have access to one) as a source of words.

Why do we type so many words with incorrect spellings? In some cases, it is because we simply do not know the correct spelling, so we make a "best guess." In some cases, it is because we transpose two letters (e.g., "default" instead of "defualt"). Sometimes we double-type a letter accidentally (e.g., "hanndy" instead of "handy"). Sometimes we type a nearby key instead of the one we intended (e.g., "biryhday" instead of "birthday"), and so on.

Design and implement a spell-checker application in Java. Your program should maintain an array **wordList** of strings. Enable the user to enter these strings. [Note: In Chapter 17, we introduce file processing. Once you have this capability, you can obtain the words for the spell checker from a computerized dictionary stored in a file.]

Your program should ask a user to enter a word. The program should then look up that word in the **wordList** array. If the word is present in the array, your program should print “**Word is spelled correctly.**”

If the word is not present in the array, your program should print “**word is not spelled correctly.**” Then your program should try to locate other words in **wordList** that might be the word the user intended to type. For example, you can try all possible single transpositions of adjacent letters to discover that the word “default” is a direct match to a word in **wordList**. Of course, this implies that your program will check all other single transpositions, such as “edfault,” “dfeault,” “deafult,” “defalut,” and “defautl.” When you find a new word that matches one in **wordList**, print that word in a message, such as “**Did you mean "default?"**.”

Implement other tests, such as replacing each double letter with a single letter and any other tests you can develop to improve the value of your spell-checker.

10.30 (*Project: A Crossword Puzzle Generator*) Most people have worked a crossword puzzle, but few have ever attempted to generate one. Generating a crossword puzzle is suggested here as a string-manipulation project requiring substantial sophistication and effort.

There are many issues the programmer must resolve to get even the simplest crossword puzzle-generator program working. For example, how does one represent the grid of a crossword puzzle inside the computer? Should one use a series of strings, or should double-subscripted arrays be used?

The programmer needs a source of words (i.e., a computerized dictionary) that can be directly referenced by the program. In what form should these words be stored to facilitate the complex manipulations required by the program?

The really ambitious reader will want to generate the “clues” portion of the puzzle, in which the brief hints for each “across” word and each “down” word are printed for the puzzle worker. Merely printing a version of the blank puzzle itself is not a simple problem.