



Chapter 16

Strings, Characters and

Regular Expressions

Java How to Program, 9/e



OBJECTIVES

In this chapter you'll learn:

- To create and manipulate immutable character-string objects of class `String`.
- To create and manipulate mutable character-string objects of class `StringBuilder`.
- To create and manipulate objects of class `Character`.
- To break a `String` object into tokens using `String` method `split`.
- To use regular expressions to validate `String` data entered into an application.



16.1 Introduction

16.2 Fundamentals of Characters and Strings

16.3 Class String

16.3.1 **String** Constructors

16.3.2 **String** Methods **length**, **charAt** and **getChars**

16.3.3 Comparing Strings

16.3.4 Locating Characters and Substrings in Strings

16.3.5 Extracting Substrings from Strings

16.3.6 Concatenating Strings

16.3.7 Miscellaneous **String** Methods

16.3.8 **String** Method **valueOf**

16.4 Class StringBuilder

16.4.1 **StringBuilder** Constructors

16.4.2 **StringBuilder** Methods **length**, **capacity**, **setLength** and **ensureCapacity**

16.4.3 **StringBuilder** Methods **charAt**, **setCharAt**, **getChars** and **reverse**

16.4.4 **StringBuilder** **append** Methods

16.4.5 **StringBuilder** Insertion and Deletion Methods

16.5 Class Character

16.6 Tokenizing Strings

16.7 Regular Expressions, Class Pattern and Class Matcher

16.8 Wrap-Up



16.1 Introduction

- ▶ This chapter discusses class **String**, class **StringBuilder** and class **Character** from the **java.lang** package.
- ▶ These classes provide the foundation for string and character manipulation in Java.
- ▶ The chapter also discusses regular expressions that provide applications with the capability to validate input.



16.2 Fundamentals of Characters and Strings

- ▶ A program may contain **character literals**.
 - An integer value represented as a character in single quotes.
 - The value of a character literal is the integer value of the character in the **Unicode character set**.
- ▶ **String literals** (stored in memory as **String** objects) are written as a sequence of characters in double quotation marks.



Performance Tip 16.1

To conserve memory, Java treats all string literals with the same contents as a single String object that has many references to it.



16.3 Class String

- ▶ Class **String** is used to represent strings in Java.
- ▶ The next several subsections cover many of class **String**'s capabilities.



16.3.1 String Constructors

- ▶ No-argument constructor creates a **String** that contains no characters (i.e., the **empty string**, which can also be represented as "") and has a length of 0.
- ▶ Constructor that takes a **String** object copies the argument into the new **String**.
- ▶ Constructor that takes a **char** array creates a **String** containing a copy of the characters in the array.
- ▶ Constructor that takes a **char** array and two integers creates a **String** containing the specified portion of the array.



```
1 // Fig. 16.1: StringConstructors.java
2 // String class constructors.
3
4 public class StringConstructors
5 {
6     public static void main( String[] args )
7     {
8         char[] charArray = { 'b', 'i', 'r', 't', 'h', ' ', 'd', 'a', 'y' };
9         String s = new String( "hello" );
10
11     // use String constructors
12     String s1 = new String();
13     String s2 = new String( s );
14     String s3 = new String( charArray );
15     String s4 = new String( charArray, 6, 3 );
16
17     System.out.printf(
18         "s1 = %s\ns2 = %s\ns3 = %s\ns4 = %s\n",
19         s1, s2, s3, s4 ); // display strings
20 } // end main
21 } // end class StringConstructors
```

Fig. 16.1 | String class constructors. (Part I of 2.)



```
s1 =  
s2 = hello  
s3 = birth day  
s4 = day
```

Fig. 16.1 | String class constructors. (Part 2 of 2.)



Software Engineering Observation 16.1

*It's not necessary to copy an existing `String` object.
`String` objects are **immutable**—their character
contents cannot be changed after they're created, because
class `String` does not provide methods that allow the
contents of a `String` object to be modified.*



Common Programming Error 16.1

*Accessing a character 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`.*



16.3.2 String Methods `length`, `charAt` and `getChars`

- ▶ **String** method `length` determines the number of characters in a string.
- ▶ **String** method `charAt` returns the character at a specific position in the **String**.
- ▶ **String** method `getChars` copies the characters of a **String** into a character array.
 - The first argument is the starting index in the **String** from which characters are to be copied.
 - 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 to be copied.
 - The last argument is the starting index where the copied characters are placed in the target character array.



```
1 // Fig. 16.2: StringMiscellaneous.java
2 // This application demonstrates the length, charAt and getChars
3 // methods of the String class.
4
5 public class StringMiscellaneous
{
6     public static void main( String[] args )
7     {
8         String s1 = "hello there";
9         char[] charArray = new char[ 5 ];
10
11        System.out.printf( "s1: %s", s1 );
12
13        // test length method
14        System.out.printf( "\nLength of s1: %d", s1.length() );
15
16        // loop through characters in s1 with charAt and display reversed
17        System.out.print( "\nThe string reversed is: " );
18
19        for ( int count = s1.length() - 1; count >= 0; count-- )
20            System.out.printf( "%c ", s1.charAt( count ) );
21
22
```

Fig. 16.2 | String methods length, charAt and getChars. (Part 1 of 2.)



```
23     // copy characters from string into charArray
24     s1.getChars( 0, 5, charArray, 0 );
25     System.out.print( "\nThe character array is: " );
26
27     for ( char character : charArray )
28         System.out.print( character );
29
30     System.out.println();
31 } // end main
32 } // end class StringMiscellaneous
```

```
s1: hello there
Length of s1: 11
The string reversed is: e r e h t   o l l e h
The character array is: hello
```

Fig. 16.2 | String methods length, charAt and getChars. (Part 2 of 2.)



16.3.3 Comparing Strings

- ▶ Strings are compared using the numeric codes of the characters in the strings.
- ▶ Figure 16.3 demonstrates `String` methods `equals`, `equalsIgnoreCase`, `compareTo` and `regionMatches` and using the equality operator `==` to compare `String` objects.



```
1 // Fig. 16.3: StringCompare.java
2 // String methods equals, equalsIgnoreCase, compareTo and regionMatches.
3
4 public class StringCompare
5 {
6     public static void main( String[] args )
7     {
8         String s1 = new String( "hello" ); // s1 is a copy of "hello"
9         String s2 = "goodbye";
10        String s3 = "Happy Birthday";
11        String s4 = "happy birthday";
12
13        System.out.printf(
14             "s1 = %s\ns2 = %s\ns3 = %s\ns4 = %s\n\n", s1, s2, s3, s4 );
15    }
}
```

Fig. 16.3 | String methods equals, equalsIgnoreCase, compareTo and regionMatches. (Part I of 5.)



```
16    // test for equality
17    if ( s1.equals( "hello" ) ) // true
18        System.out.println( "s1 equals \"hello\"" );
19    else
20        System.out.println( "s1 does not equal \"hello\"" );
21
22    // test for equality with ==
23    if ( s1 == "hello" ) // false; they are not the same object
24        System.out.println( "s1 is the same object as \"hello\"" );
25    else
26        System.out.println( "s1 is not the same object as \"hello\"" );
27
28    // test for equality (ignore case)
29    if ( s3.equalsIgnoreCase( s4 ) ) // true
30        System.out.printf( "%s equals %s with case ignored\n", s3, s4 );
31    else
32        System.out.println( "s3 does not equal s4" );
33
```

Fig. 16.3 | String methods equals, equalsIgnoreCase, compareTo and regionMatches. (Part 2 of 5.)



```
34     // test compareTo
35     System.out.printf(
36         "\ns1.compareTo( s2 ) is %d", s1.compareTo( s2 ) );
37     System.out.printf(
38         "\ns2.compareTo( s1 ) is %d", s2.compareTo( s1 ) );
39     System.out.printf(
40         "\ns1.compareTo( s1 ) is %d", s1.compareTo( s1 ) );
41     System.out.printf(
42         "\ns3.compareTo( s4 ) is %d", s3.compareTo( s4 ) );
43     System.out.printf(
44         "\ns4.compareTo( s3 ) is %d\n\n", s4.compareTo( s3 ) );
45
```

Fig. 16.3 | String methods equals, equalsIgnoreCase, compareTo and regionMatches. (Part 3 of 5.)



```
46     // test regionMatches (case sensitive)
47     if ( s3.regionMatches( 0, s4, 0, 5 ) )
48         System.out.println( "First 5 characters of s3 and s4 match" );
49     else
50         System.out.println(
51             "First 5 characters of s3 and s4 do not match" );
52
53     // test regionMatches (ignore case)
54     if ( s3.regionMatches( true, 0, s4, 0, 5 ) )
55         System.out.println(
56             "First 5 characters of s3 and s4 match with case ignored" );
57     else
58         System.out.println(
59             "First 5 characters of s3 and s4 do not match" );
60 } // end main
61 } // end class StringCompare
```

Fig. 16.3 | String methods equals, equalsIgnoreCase, compareTo and regionMatches. (Part 4 of 5.)



```
s1 = hello
s2 = goodbye
s3 = Happy Birthday
s4 = happy birthday

s1 equals "hello"
s1 is not the same object as "hello"
Happy Birthday equals happy birthday with case ignored

s1.compareTo( s2 ) is 1
s2.compareTo( s1 ) is -1
s1.compareTo( s1 ) is 0
s3.compareTo( s4 ) is -32
s4.compareTo( s3 ) is 32

First 5 characters of s3 and s4 do not match
First 5 characters of s3 and s4 match with case ignored
```

Fig. 16.3 | String methods equals, equalsIgnoreCase, compareTo and regionMatches. (Part 5 of 5.)



16.3.3 Comparing Strings (cont.)

- ▶ Method `equals` tests any two objects for equality
 - The method returns `true` if the contents of the objects are equal, and `false` otherwise.
 - Uses a [lexicographical comparison](#).
- ▶ When primitive-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 string literal objects with the same contents as one `String` object to which there can be many references.



Common Programming Error 16.2

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.



16.3.3 Comparing Strings (cont.)

- ▶ String method `equalsIgnoreCase` ignores whether the letters in each `String` are uppercase or lowercase when performing the comparison.
- ▶ Method `compareTo` is declared in the `Comparable` interface and implemented in the `String` class.
 - 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.



16.3.3 Comparing Strings (cont.)

- ▶ Method **regionMatches** compares portions of two **Strings** 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.
- ▶ Five-argument version of method **regionMatches**:
 - When 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 four-argument **regionMatches** method.



16.3.3 Comparing Strings (cont.)

- ▶ **String** methods `startsWith` and `endsWith` determine whether strings start with or end with a particular set of characters



```
1 // Fig. 16.4: StringStartEnd.java
2 // String methods startsWith and endsWith.
3
4 public class StringStartEnd
5 {
6     public static void main( String[] args )
7     {
8         String[] strings = { "started", "starting", "ended", "ending" };
9
10        // test method startsWith
11        for ( String string : strings )
12        {
13            if ( string.startsWith( "st" ) )
14                System.out.printf( "\"%s\" starts with \"st\"\n", string );
15        } // end for
16
17        System.out.println();
18    }
}
```

Fig. 16.4 | String methods `startsWith` and `endsWith`. (Part 1 of 3.)



```
19     // test method startsWith starting from position 2 of string
20     for ( String string : strings )
21     {
22         if ( string.startsWith( "art", 2 ) )
23             System.out.printf(
24                 "%" starts with "art" at position 2\n", string );
25     } // end for
26
27     System.out.println();
28
29     // test method endsWith
30     for ( String string : strings )
31     {
32         if ( string.endsWith( "ed" ) )
33             System.out.printf( "%" ends with "ed"\n", string );
34     } // end for
35 } // end main
36 } // end class StringStartEnd
```

Fig. 16.4 | String methods `startsWith` and `endsWith`. (Part 2 of 3.)



```
"started" starts with "st"  
"starting" starts with "st"

"started" starts with "art" at position 2  
"starting" starts with "art" at position 2

"started" ends with "ed"  
"ended" ends with "ed"
```

Fig. 16.4 | String methods `startsWith` and `endsWith`. (Part 3 of 3.)



16.3.4 Locating Characters and Substrings in Strings

- ▶ Figure 16.5 demonstrates the many versions of **String** methods `indexOf` and `lastIndexOf` that search for a specified character or substring in a **String**.



```
1 // Fig. 16.5: StringTokenizer.java
2 // StringTokenizer class demonstrating the StringTokenizer class.
3
4 public class StringTokenizer
5 {
6     public static void main( String[] args )
7     {
8         String letters = "abcdefghijklmnopqrstuvwxyz";
9
10        // test indexOf to locate a character in a string
11        System.out.printf(
12            "'c' is located at index %d\n", letters.indexOf( 'c' ) );
13        System.out.printf(
14            "'a' is located at index %d\n", letters.indexOf( 'a', 1 ) );
15        System.out.printf(
16            "'$' is located at index %d\n", letters.indexOf( '$' ) );
17    }
}
```

Fig. 16.5 | StringTokenizer class demonstrating the StringTokenizer class. (Part 1 of 4.)



```
18 // test lastIndexOf to find a character in a string
19 System.out.printf( "Last 'c' is located at index %d\n",
20     letters.lastIndexOf( 'c' ) );
21 System.out.printf( "Last 'a' is located at index %d\n",
22     letters.lastIndexOf( 'a', 25 ) );
23 System.out.printf( "Last '$' is located at index %d\n\n",
24     letters.lastIndexOf( '$' ) );
25
26 // test indexOf to locate a substring in a string
27 System.out.printf( "\"def\" is located at index %d\n",
28     letters.indexOf( "def" ) );
29 System.out.printf( "\"def\" is located at index %d\n",
30     letters.indexOf( "def", 7 ) );
31 System.out.printf( "\"hello\" is located at index %d\n\n",
32     letters.indexOf( "hello" ) );
33
```

Fig. 16.5 | String-searching methods `indexOf` and `lastIndexOf`. (Part 2 of 4.)



```
34     // test lastIndexOf to find a substring in a string
35     System.out.printf( "Last \"def\" is located at index %d\n",
36                         letters.lastIndexOf( "def" ) );
37     System.out.printf( "Last \"def\" is located at index %d\n",
38                         letters.lastIndexOf( "def", 25 ) );
39     System.out.printf( "Last \"hello\" is located at index %d\n",
40                         letters.lastIndexOf( "hello" ) );
41 } // end main
42 } // end class StringTokenizer
```

Fig. 16.5 | String-searching methods `indexOf` and `lastIndexOf`. (Part 3 of 4.)



```
'c' is located at index 2
'a' is located at index 13
'$' is located at index -1

Last 'c' is located at index 15
Last 'a' is located at index 13
Last '$' is located at index -1

"def" is located at index 3
"def" is located at index 16
"hello" is located at index -1

Last "def" is located at index 16
Last "def" is located at index 16
Last "hello" is located at index -1
```

Fig. 16.5 | String-searching methods `indexOf` and `lastIndexOf`. (Part 4 of 4.)



16.3.4 Locating Characters and Substrings in Strings (cont.)

- ▶ Method `indexOf` locates the first occurrence of a character in a **String**. If the method finds the character, it returns the character's index in the **String**—otherwise, it returns `-1`.
- ▶ A second version of `indexOf` takes two integer arguments—the character and the starting index at which the search of the **String** should begin.
- ▶ Method `lastIndexOf` locates the last occurrence of a character in a **String**. The method searches from the end of the **String** toward the beginning. If it finds the character, it returns the character's index in the **String**—otherwise, it returns `-1`.
- ▶ A second version of `lastIndexOf` takes two integer arguments—the integer representation of the character and the index from which to begin searching backward.
- ▶ There are also versions of these methods that search for substrings in **Strings**.



16.3.5 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.
- ▶ The version that takes one integer argument specifies the starting index in the original **String** from which characters are to be copied.
- ▶ The version that takes two integer arguments receives the starting index from which to copy characters in the original **String** and the index one beyond the last character to copy.



```
1 // Fig. 16.6: SubString.java
2 // String class substring methods.
3
4 public class SubString
5 {
6     public static void main( String[] args )
7     {
8         String letters = "abcdefghijklmabcdefghijklm";
9
10        // test substring methods
11        System.out.printf( "Substring from index 20 to end is \"%s\"\n",
12                           letters.substring( 20 ) );
13        System.out.printf( "%s \"%s\"\n",
14                           "Substring from index 3 up to, but not including 6 is",
15                           letters.substring( 3, 6 ) );
16    } // end main
17 } // end class SubString
```

```
Substring from index 20 to end is "hijklm"
Substring from index 3 up to, but not including 6 is "def"
```

Fig. 16.6 | String class substring methods.



16.3.6 Concatenating Strings

- ▶ **String** method `concat` concatenates two **String** objects and returns a new **String** object containing the characters from both original **Strings**.
- ▶ The original **Strings** to which `s1` and `s2` refer are not modified.



```
1 // Fig. 16.7: StringConcatenation.java
2 // String method concat.
3
4 public class StringConcatenation
5 {
6     public static void main( String[] args )
7     {
8         String s1 = "Happy ";
9         String s2 = "Birthday";
10
11        System.out.printf( "s1 = %s\ns2 = %s\n\n", s1, s2 );
12        System.out.printf(
13            "Result of s1.concat( s2 ) = %s\n", s1.concat( s2 ) );
14        System.out.printf( "s1 after concatenation = %s\n", s1 );
15    } // end main
16 } // end class StringConcatenation
```

```
s1 = Happy
s2 = Birthday
```

```
Result of s1.concat( s2 ) = Happy Birthday
s1 after concatenation = Happy
```

Fig. 16.7 | String method concat.



16.3.7 Miscellaneous String Methods

- ▶ Method `replace` return a new **String** object in which every occurrence of the first **char** argument is replaced with the second.
 - An overloaded version enables you to replace substrings rather than individual characters.
- ▶ Method `toUpperCase` generates a new **String** with uppercase letters.
- ▶ Method `toLowerCase` returns a new **String** object with lowercase letters.
- ▶ Method `trim` generates a new **String** object that removes all whitespace characters that appear at the beginning or end of the **String** on which `trim` operates.
- ▶ Method `toCharArray` creates a new character array containing a copy of the characters in the **String**.



```
1 // Fig. 16.8: StringMiscellaneous2.java
2 // String methods replace, toLowerCase, toUpperCase, trim and toCharArray.
3
4 public class StringMiscellaneous2
5 {
6     public static void main( String[] args )
7     {
8         String s1 = "he11o";
9         String s2 = "GOODBYE";
10        String s3 = "    spaces    ";
11
12        System.out.printf( "s1 = %s\ns2 = %s\ns3 = %s\n\n", s1, s2, s3 );
13    }
}
```

Fig. 16.8 | String methods replace, toLowerCase, toUpperCase, trim and toCharArray. (Part 1 of 3.)



```
14 // test method replace
15 System.out.printf(
16     "Replace 'l' with 'L' in s1: %s\n\n", s1.replace( 'l', 'L' ) );
17
18 // test toLowerCase and toUpperCase
19 System.out.printf( "s1.toUpperCase() = %s\n", s1.toUpperCase() );
20 System.out.printf( "s2.toLowerCase() = %s\n\n", s2.toLowerCase() );
21
22 // test trim method
23 System.out.printf( "s3 after trim = \"%s\"\n\n", s3.trim() );
24
25 // test toCharArray method
26 char[] charArray = s1.toCharArray();
27 System.out.print( "s1 as a character array = " );
28
29 for ( char character : charArray )
30     System.out.print( character );
31
32 System.out.println();
33 } // end main
34 } // end class StringMiscellaneous2
```

Fig. 16.8 | String methods replace, toLowerCase, toUpperCase, trim and toCharArray. (Part 2 of 3.)



```
s1 = hello
s2 = GOODBYE
s3 =    spaces

Replace 'l' with 'L' in s1: heLLo

s1.toUpperCase() = HELLO
s2.toLowerCase() = goodbye

s3 after trim = "spaces"

s1 as a character array = hello
```

Fig. 16.8 | String methods replace, toLowerCase, toUpperCase, trim and toCharArray. (Part 3 of 3.)



16.3.8 String Method `valueOf`

- ▶ Class `String` provides `static` `valueOf` methods that take an argument of any type and convert it to a `String` object.
- ▶ Class `StringBuilder` is used to create and manipulate dynamic string information.
- ▶ Every `StringBuilder` is capable of storing a number of characters specified by its capacity.
- ▶ If the capacity of a `StringBuilder` is exceeded, the capacity expands to accommodate the additional characters.



```
1 // Fig. 16.9: StringValueOf.java
2 // String valueOf methods.
3
4 public class StringValueOf
5 {
6     public static void main( String[] args )
7     {
8         char[] charArray = { 'a', 'b', 'c', 'd', 'e', 'f' };
9         boolean booleanValue = true;
10        char characterValue = 'Z';
11        int integerValue = 7;
12        long longValue = 1000000000L; // L suffix indicates long
13        float floatValue = 2.5f; // f indicates that 2.5 is a float
14        double doubleValue = 33.333; // no suffix, double is default
15        Object objectRef = "hello"; // assign string to an Object reference
16
```

Fig. 16.9 | String valueOf methods. (Part I of 3.)



```
17 System.out.printf(  
18     "char array = %s\n", String.valueOf( charArray ) );  
19 System.out.printf( "part of char array = %s\n",  
20     String.valueOf( charArray, 3, 3 ) );  
21 System.out.printf(  
22     "boolean = %s\n", String.valueOf( booleanValue ) );  
23 System.out.printf(  
24     "char = %s\n", String.valueOf( characterValue ) );  
25 System.out.printf( "int = %s\n", String.valueOf( integerValue ) );  
26 System.out.printf( "long = %s\n", String.valueOf( longValue ) );  
27 System.out.printf( "float = %s\n", String.valueOf( floatValue ) );  
28 System.out.printf(  
29     "double = %s\n", String.valueOf( doubleValue ) );  
30 System.out.printf( "Object = %s", String.valueOf( objectRef ) );  
31 } // end main  
32 } // end class StringValueOf
```

Fig. 16.9 | String valueOf methods. (Part 2 of 3.)



```
char array = abcdef
part of char array = def
boolean = true
char = Z
int = 7
long = 10000000000
float = 2.5
double = 33.333
Object = hello
```

Fig. 16.9 | String valueOf methods. (Part 3 of 3.)



© Copyright 1992–2012 by Pearson
Education, Inc. All Rights Reserved.



16.4 Class **StringBuilder**

- ▶ We now discuss the features of class **StringBuilder** for creating and manipulating *dynamic* string information—that is, *modifiable* strings.
- ▶ Every **StringBuilder** is capable of storing a number of characters specified by its capacity.
- ▶ If a **StringBuilder**'s capacity is exceeded, the capacity expands to accommodate additional characters.



Performance Tip 16.2

Java can perform certain optimizations involving String objects (such as referring to one String object from multiple variables) because it knows these objects will not change. Strings (not StringBuilder) should be used if the data will not change.



Performance Tip 16.3

In programs that frequently perform string concatenation, or other string modifications, it's often more efficient to implement the modifications with class `StringBuilder`.



Software Engineering Observation 16.2

*StringBuilders are not thread safe. If multiple threads require access to the same dynamic string information, use class **StringBuffer** in your code.*

*Classes **StringBuilder** and **StringBuffer** provide identical capabilities, but class **StringBuffer** is thread safe. For more details on threading, see Chapter 26.*

108



16.4.1 **StringBuilder** Constructors

- ▶ No-argument constructor creates a **StringBuilder** with no characters in it and an initial capacity of 16 characters.
- ▶ Constructor that takes an integer argument creates a **StringBuilder** with no characters in it and the initial capacity specified by the integer argument.
- ▶ Constructor that takes a **String** argument creates a **StringBuilder** containing the characters in the **String** argument. The initial capacity is the number of characters in the **String** argument plus 16.
- ▶ Method **toString** of class **StringBuilder** returns the **StringBuilder** contents as a **String**.



```
1 // Fig. 16.10: StringBuilderConstructors.java
2 // StringBuilder constructors.
3
4 public class StringBuilderConstructors
5 {
6     public static void main( String[] args )
7     {
8         StringBuilder buffer1 = new StringBuilder();
9         StringBuilder buffer2 = new StringBuilder( 10 );
10        StringBuilder buffer3 = new StringBuilder( "hello" );
11
12        System.out.printf( "buffer1 = %s\n", buffer1 );
13        System.out.printf( "buffer2 = %s\n", buffer2 );
14        System.out.printf( "buffer3 = %s\n", buffer3 );
15    } // end main
16 } // end class StringBuilderConstructors
```

```
buffer1 = ""
buffer2 = ""
buffer3 = "hello"
```

Fig. 16.10 | StringBuilder constructors.



16.4.2 **StringBuilder** Methods `length`, `capacity`, `setLength` and `ensureCapacity`

- ▶ Methods `length` and `capacity` return the number of characters currently in a **StringBuilder** and the number of characters that can be stored in it without allocating more memory, respectively.
- ▶ Method `ensureCapacity` guarantees that a **StringBuilder** has at least the specified capacity.
- ▶ Method `setLength` increases or decreases the length of a **StringBuilder**.
 - If the specified length is less than the current number of characters, the buffer is truncated to the specified length.
 - If the specified length is greater than the number of characters, `null` characters are appended until the total number of characters in the **StringBuilder** is equal to the specified length.



```
1 // Fig. 16.11: StringBuilderCapLen.java
2 // StringBuilder length, setLength, capacity and ensureCapacity methods.
3
4 public class StringBuilderCapLen
5 {
6     public static void main( String[] args )
7     {
8         StringBuilder buffer = new StringBuilder( "Hello, how are you?" );
9
10        System.out.printf( "buffer = %s\nlength = %d\ncapacity = %d\n\n",
11                           buffer.toString(), buffer.length(), buffer.capacity() );
12
13        buffer.ensureCapacity( 75 );
14        System.out.printf( "New capacity = %d\n\n", buffer.capacity() );
15
16        buffer.setLength( 10 );
17        System.out.printf( "New length = %d\nbuffer = %s\n",
18                           buffer.length(), buffer.toString() );
19    } // end main
20 } // end class StringBuilderCapLen
```

Fig. 16.11 | StringBuilder length, setLength, capacity and ensureCapacity methods. (Part I of 2.)



```
buffer = Hello, how are you?  
length = 19  
capacity = 35
```

```
New capacity = 75
```

```
New length = 10  
buffer = Hello, how
```

Fig. 16.11 | `StringBuilder` `length`, `setLength`, `capacity` and `ensureCapacity` methods. (Part 2 of 2.)



Performance Tip 16.4

Dynamically increasing the capacity of a `StringBuilder` can take a relatively long time. Executing a large number of these operations can degrade the performance of an application. If a `StringBuilder` is going to increase greatly in size, possibly multiple times, setting its capacity high at the beginning will increase performance.



16.4.3 **StringBuilder** Methods **charAt**, **setCharAt**, **getChars** and **reverse**

- ▶ Method **charAt** takes an integer argument and returns the character in the **StringBuilder** at that index.
- ▶ Method **getChars** copies characters from a **StringBuilder** into the character array argument.
 - Four arguments—the starting index from which characters should be copied, the index one past the last character to be copied, 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 **setCharAt** takes an integer and a character argument and sets the character at the specified position in the **StringBuilder** to the character argument.
- ▶ Method **reverse** reverses the contents of the **StringBuilder**.



Common Programming Error 16.3

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



```
1 // Fig. 16.12: StringBuilderChars.java
2 // StringBuilder methods charAt, setCharAt, getChars and reverse.
3
4 public class StringBuilderChars
5 {
6     public static void main( String[] args )
7     {
8         StringBuilder buffer = new StringBuilder( "hello there" );
9
10        System.out.printf( "buffer = %s\n", buffer.toString() );
11        System.out.printf( "Character at 0: %s\nCharacter at 4: %s\n\n",
12                           buffer.charAt( 0 ), buffer.charAt( 4 ) );
13
14        char[] charArray = new char[ buffer.length() ];
15        buffer.getChars( 0, buffer.length(), charArray, 0 );
16        System.out.print( "The characters are: " );
17
18        for ( char character : charArray )
19            System.out.print( character );
```

Fig. 16.12 | StringBuilder methods charAt, setCharAt, getChars and reverse. (Part I of 2.)



```
20      buffer.setCharAt( 0, 'H' );
21      buffer.setCharAt( 6, 'T' );
22      System.out.printf( "\n\nbuffer = %s", buffer.toString() );
23
24      buffer.reverse();
25      System.out.printf( "\n\nbuffer = %s\n", buffer.toString() );
26  } // end main
27 } // end class StringBuilderChars
```

```
buffer = hello there
Character at 0: h
Character at 4: o
```

The characters are: hello there

```
buffer = Hello There
buffer = erehT olleH
```

Fig. 16.12 | `StringBuilder` methods `charAt`, `setCharAt`, `getChars` and `reverse`. (Part 2 of 2.)



16.4.4 `StringBuilder` append Methods

- ▶ Overloaded `append` methods allow values of various types to be appended to the end of a `StringBuilder`.
- ▶ Versions are provided for each of the primitive types, and for character arrays, `Strings`, `Objects`, and more.



16.4.4 **StringBuilder** append Methods (cont.)

- ▶ A compiler can use **StringBuilder** (or **StringBuffer**) and the **append** methods to implement the **+** and **$+=$** **String** concatenation operators.



```
1 // Fig. 16.13: StringBuilderAppend.java
2 // StringBuilder append methods.
3
4 public class StringBuilderAppend
5 {
6     public static void main( String[] args )
7     {
8         Object objectRef = "hello";
9         String string = "goodbye";
10        char[] charArray = { 'a', 'b', 'c', 'd', 'e', 'f' };
11        boolean booleanValue = true;
12        char characterValue = 'Z';
13        int integerValue = 7;
14        long longValue = 1000000000L;
15        float floatValue = 2.5f;
16        double doubleValue = 33.333;
17    }
}
```

Fig. 16.13 | StringBuilder append methods. (Part I of 3.)



```
18    StringBuilder lastBuffer = new StringBuilder( "last buffer" );
19    StringBuilder buffer = new StringBuilder();
20
21    buffer.append( objectRef );
22    buffer.append( "\n" );
23    buffer.append( string );
24    buffer.append( "\n" );
25    buffer.append( charArray );
26    buffer.append( "\n" );
27    buffer.append( charArray, 0, 3 );
28    buffer.append( "\n" );
29    buffer.append( booleanValue );
30    buffer.append( "\n" );
31    buffer.append( characterValue );
32    buffer.append( "\n" );
33    buffer.append( integerValue );
34    buffer.append( "\n" );
35    buffer.append( longValue );
36    buffer.append( "\n" );
37    buffer.append( floatValue );
38    buffer.append( "\n" );
39    buffer.append( doubleValue );
40    buffer.append( "\n" );
41    buffer.append( lastBuffer );
```

Fig. 16.13 | `StringBuilder` append methods. (Part 2 of 3.)



```
42
43     System.out.printf( "buffer contains %s\n", buffer.toString() );
44 } // end main
45 } // end StringBuilderAppend
```

```
buffer contains hello
goodbye
abcdef
abc
true
Z
7
10000000000
2.5
33.333
last buffer
```

Fig. 16.13 | `StringBuilder` append methods. (Part 3 of 3.)



16.4.5 **StringBuilder** Insertion and Deletion Methods

- ▶ Overloaded `insert` methods insert values of various types at any position in a **StringBuilder**.
 - Versions are provided for the primitive types and for character arrays, **Strings**, **Objects** and **CharSequences**.
 - Each method takes its second argument, converts it to a **String** and inserts it at the index specified by the first argument.
- ▶ Methods `delete` and `deleteCharAt` delete characters at any position in a **StringBuilder**.
- ▶ Method `delete` takes two arguments—the starting index and the index one past the end of the characters to delete.
- ▶ Method `deleteCharAt` takes one argument—the index of the character to delete.



```
1 // Fig. 16.14: StringBuilderInsertDelete.java
2 // StringBuilder methods insert, delete and deleteCharAt.
3
4 public class StringBuilderInsertDelete
5 {
6     public static void main( String[] args )
7     {
8         Object objectRef = "hello";
9         String string = "goodbye";
10        char[] charArray = { 'a', 'b', 'c', 'd', 'e', 'f' };
11        boolean booleanValue = true;
12        char characterValue = 'K';
13        int integerValue = 7;
14        long longValue = 10000000;
15        float floatValue = 2.5f; // f suffix indicates that 2.5 is a float
16        double doubleValue = 33.333;
17    }
}
```

Fig. 16.14 | StringBuilder methods insert, delete and deleteCharAt. (Part 1 of 3.)



```
18  StringBuilder buffer = new StringBuilder();  
19  
20      buffer.insert( 0, objectRef );  
21      buffer.insert( 0, " " ); // each of these contains two spaces  
22      buffer.insert( 0, string );  
23      buffer.insert( 0, " " );  
24      buffer.insert( 0, charArray );  
25      buffer.insert( 0, " " );  
26      buffer.insert( 0, charArray, 3, 3 );  
27      buffer.insert( 0, " " );  
28      buffer.insert( 0, booleanValue );  
29      buffer.insert( 0, " " );  
30      buffer.insert( 0, characterValue );  
31      buffer.insert( 0, " " );  
32      buffer.insert( 0, integerValue );  
33      buffer.insert( 0, " " );  
34      buffer.insert( 0, longValue );  
35      buffer.insert( 0, " " );  
36      buffer.insert( 0, floatValue );  
37      buffer.insert( 0, " " );  
38      buffer.insert( 0, doubleValue );  
39
```

Fig. 16.14 | `StringBuilder` methods `insert`, `delete` and `deleteCharAt`. (Part 2 of 3.)



```
40     System.out.printf(
41         "buffer after inserts:\n%s\n", buffer.toString() );
42
43     buffer.deleteCharAt( 10 ); // delete 5 in 2.5
44     buffer.delete( 2, 6 ); // delete .333 in 33.333
45
46     System.out.printf(
47         "buffer after deletes:\n%s\n", buffer.toString() );
48 } // end main
49 } // end class StringBuilderInsertDelete
```

```
buffer after inserts:
33.333 2.5 10000000 7 K true def abcdef goodbye hello

buffer after deletes:
33 2. 10000000 7 K true def abcdef goodbye hello
```

Fig. 16.14 | `StringBuilder` methods `insert`, `delete` and `deleteCharAt`. (Part 3 of 3.)



16.5 Class Character

- ▶ Eight type-wrapper classes that enable primitive-type values to be treated as objects:
 - `Boolean`, `Character`, `Double`, `Float`, `Byte`, `Short`, `Integer` and `Long`
- ▶ Most `Character` methods are `static` methods designed for convenience in processing individual `char` values.



16.5 Class Character (cont.)

- ▶ Method `isDefined` determines whether a character is defined in the Unicode character set.
- ▶ Method `isDigit` determines whether a character is a defined Unicode digit.
- ▶ Method `isJavaIdentifierStart` determines whether a character can be the first character of an identifier in Java—that is, a letter, an underscore (_) or a dollar sign (\$).
- ▶ Method `isJavaIdentifierPart` determine whether a character can be used in an identifier in Java—that is, a digit, a letter, an underscore (_) or a dollar sign (\$).



```
1 // Fig. 16.15: StaticCharMethods.java
2 // Character static methods for testing characters and converting case.
3 import java.util.Scanner;
4
5 public class StaticCharMethods
6 {
7     public static void main( String[] args )
8     {
9         Scanner scanner = new Scanner( System.in ); // create scanner
10        System.out.println( "Enter a character and press Enter" );
11        String input = scanner.next();
12        char c = input.charAt( 0 ); // get input character
13    }
}
```

Fig. 16.15 | Character static methods for testing characters and converting case. (Part 1 of 5.)



```
14 // display character info
15 System.out.printf( "is defined: %b\n", Character.isDefined( c ) );
16 System.out.printf( "is digit: %b\n", Character.isDigit( c ) );
17 System.out.printf( "is first character in a Java identifier: %b\n",
18     Character.isJavaIdentifierStart( c ) );
19 System.out.printf( "is part of a Java identifier: %b\n",
20     Character.isJavaIdentifierPart( c ) );
21 System.out.printf( "is letter: %b\n", Character.isLetter( c ) );
22 System.out.printf(
23     "is letter or digit: %b\n", Character.isLetterOrDigit( c ) );
24 System.out.printf(
25     "is lower case: %b\n", Character.isLowerCase( c ) );
26 System.out.printf(
27     "is upper case: %b\n", Character.isUpperCase( c ) );
28 System.out.printf(
29     "to upper case: %s\n", Character.toUpperCase( c ) );
30 System.out.printf(
31     "to lower case: %s\n", Character.toLowerCase( c ) );
32 } // end main
33 } // end class StaticCharMethods
```

Fig. 16.15 | Character static methods for testing characters and converting case. (Part 2 of 5.)



Enter a character and press Enter

```
A
is defined: true
is digit: false
is first character in a Java identifier: true
is part of a Java identifier: true
is letter: true
is letter or digit: true
is lower case: false
is upper case: true
to upper case: A
to lower case: a
```

Fig. 16.15 | Character static methods for testing characters and converting case. (Part 3 of 5.)



Enter a character and press Enter

```
8
is defined: true
is digit: true
is first character in a Java identifier: false
is part of a Java identifier: true
is letter: false
is letter or digit: true
is lower case: false
is upper case: false
to upper case: 8
to lower case: 8
```

Fig. 16.15 | Character static methods for testing characters and converting case. (Part 4 of 5.)



```
Enter a character and press Enter
$
is defined: true
is digit: false
is first character in a Java identifier: true
is part of a Java identifier: true
is letter: false
is letter or digit: false
is lower case: false
is upper case: false
to upper case: $
to lower case: $
```

Fig. 16.15 | Character static methods for testing characters and converting case. (Part 5 of 5.)



16.5 Class Character (cont.)

- ▶ Method `isLetter` determines whether a character is a letter.
- ▶ Method `isLetterOrDigit` determines whether a character is a letter or a digit.
- ▶ Method `isLowerCase` determines whether a character is a lowercase letter.
- ▶ Method `isUpperCase` determines whether a character is an uppercase letter.
- ▶ Method `toUpperCase` converts a character to its uppercase equivalent.
- ▶ Method `toLowerCase` converts a character to its lowercase equivalent.



16.5 Class Character (cont.)

- ▶ Methods `digit` and `forDigit` convert characters to digits and digits to characters, respectively, in different number systems.
- ▶ Common number systems: 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 H.



16.5 Class Character (cont.)

- ▶ **Character** method `forDigit` converts its first argument into a character in the number system specified by its second argument.
- ▶ **Character** method `digit` converts its first argument into an integer in the number system specified by its second argument.
 - The radix (second argument) must be between 2 and 36, inclusive.



```
1 // Fig. 16.16: StaticCharMethods2.java
2 // Character class static conversion methods.
3 import java.util.Scanner;
4
5 public class StaticCharMethods2
{
6
7     // executes application
8     public static void main( String[] args )
9     {
10         Scanner scanner = new Scanner( System.in );
11
12         // get radix
13         System.out.println( "Please enter a radix:" );
14         int radix = scanner.nextInt();
15
16         // get user choice
17         System.out.printf( "Please choose one:\n1 -- %s\n2 -- %s\n",
18                           "Convert digit to character", "Convert character to digit" );
19         int choice = scanner.nextInt();
20 }
```

Fig. 16.16 | Character class static conversion methods. (Part I of 3.)



```
21 // process request
22     switch ( choice )
23     {
24         case 1: // convert digit to character
25             System.out.println( "Enter a digit:" );
26             int digit = scanner.nextInt();
27             System.out.printf( "Convert digit to character: %s\n",
28                 Character.forDigit( digit, radix ) );
29             break;
30
31         case 2: // convert character to digit
32             System.out.println( "Enter a character:" );
33             char character = scanner.next().charAt( 0 );
34             System.out.printf( "Convert character to digit: %s\n",
35                 Character.digit( character, radix ) );
36             break;
37     } // end switch
38 } // end main
39 } // end class StaticCharMethods2
```

Fig. 16.16 | Character class static conversion methods. (Part 2 of 3.)



```
Please enter a radix:  
16  
Please choose one:  
1 -- Convert digit to character  
2 -- Convert character to digit  
2  
Enter a character:  
A  
Convert character to digit: 10
```

```
Please enter a radix:  
16  
Please choose one:  
1 -- Convert digit to character  
2 -- Convert character to digit  
1  
Enter a digit:  
13  
Convert digit to character: d
```

Fig. 16.16 | Character class static conversion methods. (Part 3 of 3.)



16.5 Class Character (cont.)

- ▶ Java automatically converts **char** literals into **Character** objects when they are assigned to **Character** variables
 - Process known as autoboxing.
- ▶ Method **charValue** returns the **char** value stored in the object.
- ▶ Method **toString** returns the **String** representation of the **char** value stored in the object.
- ▶ Method **equals** determines if two **Characters** have the same contents.



```
1 // Fig. 16.17: OtherCharMethods.java
2 // Character class non-static methods.
3 public class OtherCharMethods
4 {
5     public static void main( String[] args )
6     {
7         Character c1 = 'A';
8         Character c2 = 'a';
9
10        System.out.printf(
11            "c1 = %s\n" + "c2 = %s\n\n", c1.charValue(), c2.toString() );
12
13        if ( c1.equals( c2 ) )
14            System.out.println( "c1 and c2 are equal\n" );
15        else
16            System.out.println( "c1 and c2 are not equal\n" );
17    } // end main
18 } // end class OtherCharMethods
```

```
c1 = A
c2 = a
```

```
c1 and c2 are not equal
```

Fig. 16.17 | Character class non-static methods.



16.6 Tokenizing Strings

- ▶ When you read a sentence, your mind breaks it into **tokens**—individual words and punctuation marks that convey meaning.
- ▶ Compilers also perform tokenization.
- ▶ **String** method `split` breaks a **String** into its component tokens and returns an array of **Strings**.
- ▶ Tokens are separated by **delimiters**
 - Typically white-space characters such as space, tab, newline and carriage return.
 - Other characters can also be used as delimiters to separate tokens.



```
1 // Fig. 16.18: TokenTest.java
2 // StringTokenizer object used to tokenize strings.
3 import java.util.Scanner;
4 import java.util.StringTokenizer;
5
6 public class TokenTest
7 {
8     // execute application
9     public static void main( String[] args )
10    {
11        // get sentence
12        Scanner scanner = new Scanner( System.in );
13        System.out.println( "Enter a sentence and press Enter" );
14        String sentence = scanner.nextLine();
15
16        // process user sentence
17        String[] tokens = sentence.split( " " );
18        System.out.printf( "Number of elements: %d\nThe tokens are:\n",
19                          tokens.length );
20
21        for ( String token : tokens )
22            System.out.println( token );
23    } // end main
24 } // end class TokenTest
```

Fig. 16.18 | StringTokenizer object used to tokenize strings. (Part 1 of 2.)



```
Enter a sentence and press Enter  
This is a sentence with seven tokens  
Number of elements: 7  
The tokens are:  
This  
is  
a  
sentence  
with  
seven  
tokens
```

Fig. 16.18 | StringTokenizer object used to tokenize strings. (Part 2 of 2.)



16.7 Regular Expressions, Class Pattern and Class Matcher (cont.)

- ▶ The asterisk (*) and plus (+) are formally called **quantifiers**.
- ▶ Figure 16.22 lists all the quantifiers.
- ▶ A quantifier affects only the subexpression immediately preceding the quantifier.
- ▶ Quantifier question mark (?) matches zero or one occurrences of the expression that it quantifies.
- ▶ A set of braces containing one number ($\{n\}$) *matches exactly n occurrences of the expression it quantifies*.
- ▶ Including a comma after the number enclosed in braces matches at least n *occurrences of the quantified expression*.
- ▶ A set of braces containing two numbers ($\{n, m\}$), *matches between n and m occurrences of the expression that it qualifies*.



16.7 Regular Expressions, Class Pattern and Class Matcher (cont.)

- ▶ Quantifiers may be applied to patterns enclosed in parentheses to create more complex regular expressions.
- ▶ All of the quantifiers are **greedy**.
 - They match as many occurrences as they can as long as the match is still successful.
- ▶ If a quantifier is followed by a question mark (?), the quantifier becomes **reluctant** (sometimes called **lazy**).
 - It will match as few occurrences as possible as long as the match is still successful.
- ▶ **String** Method **matches** checks whether an entire **String** conforms to a regular expression.



Quantifier	Matches
*	Matches zero or more occurrences of the pattern.
+	Matches one or more occurrences of the pattern.
?	Matches zero or one occurrences of the pattern.
{n}	Matches exactly n occurrences.
{n,}	Matches at least n occurrences.
{n,m}	Matches between n and m (inclusive) occurrences.

Fig. 16.22 | Quantifiers used in regular expressions.



16.7 Regular Expressions, Class Pattern and Class Matcher (cont.)

- ▶ Sometimes it's useful to replace parts of a string or to split a string into pieces. For this purpose, class **String** provides methods `replaceAll`, `replaceFirst` and `split`.



16.7 Regular Expressions, Class Pattern and Class Matcher (cont.)

- ▶ **String** method `replaceAll` replaces text in a **String** with new text (the second argument) wherever the original **String** matches a regular expression (the first argument).
- ▶ Escaping a special regular-expression character with \ instructs the matching engine to find the actual character.
- ▶ **String** method `replaceFirst` replaces the first occurrence of a pattern match.



```
1 // Fig. 16.23: RegexSubstitution.java
2 // String methods replaceFirst, replaceAll and split.
3 import java.util.Arrays;
4
5 public class RegexSubstitution
6 {
7     public static void main( String[] args )
8     {
9         String firstString = "This sentence ends in 5 stars *****";
10        String secondString = "1, 2, 3, 4, 5, 6, 7, 8";
11
12        System.out.printf( "Original String 1: %s\n", firstString );
13    }
}
```

Fig. 16.23 | String methods replaceFirst, replaceAll and split. (Part 1 of 3.)



```
14 // replace '*' with '^'
15 firstString = firstString.replaceAll( "\\\*", "^" );
16
17 System.out.printf( "^ substituted for *: %s\n", firstString );
18
19 // replace 'stars' with 'carets'
20 firstString = firstString.replaceAll( "stars", "carets" );
21
22 System.out.printf(
23     "\"carets\" substituted for \"stars\": %s\n", firstString );
24
25 // replace words with 'word'
26 System.out.printf( "Every word replaced by \"word\": %s\n\n",
27     firstString.replaceAll( "\\w+", "word" ) );
28
29 System.out.printf( "Original String 2: %s\n", secondString );
30
31 // replace first three digits with 'digit'
32 for ( int i = 0; i < 3; i++ )
33     secondString = secondString.replaceFirst( "\\\d", "digit" );
34
```

Fig. 16.23 | String methods `replaceFirst`, `replaceAll` and `split`. (Part 2 of 3.)



```
35     System.out.printf(
36         "First 3 digits replaced by \"digit\" : %s\n", secondString );
37
38     System.out.print( "String split at commas: " );
39     String[] results = secondString.split( ",\\s*" ); // split on commas
40     System.out.println( Arrays.toString( results ) );
41 } // end main
42 } // end class RegexSubstitution
```

```
Original String 1: This sentence ends in 5 stars *****
^ substituted for *: This sentence ends in 5 stars ^^^^^
"carets" substituted for "stars": This sentence ends in 5 carets ^^^^^
Every word replaced by "word": word word word word word ^^^^^

Original String 2: 1, 2, 3, 4, 5, 6, 7, 8
First 3 digits replaced by "digit" : digit, digit, digit, 4, 5, 6, 7, 8
String split at commas: ["digit", "digit", "digit", "4", "5", "6", "7", "8"]
```

Fig. 16.23 | String methods replaceFirst, replaceAll and split. (Part 3 of 3.)