14.3 Class `String`

9

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

Performance Tip 14.2

As of Java SE 9, Java uses a more compact `String` *representation. This significantly reduces the amount of memory used to store* `String`*s containing only Latin-1 characters—that is, those with the character codes 0–255. For more information, see JEP 254's proposal at* `http://openjdk.java.net/jeps/254`.

14.3.1 `String` Constructors

Class `String` provides constructors for initializing `String` objects in a variety of ways. Four of the constructors are demonstrated in the `main` method of Fig. 14.1.

```
1   // Fig. 14.1: StringConstructors.java
```

```
2   // String class constructors.
3
4   public class StringConstructors {
5     public static void main(String[] args) {
6       char[] charArray = {'b', 'i', 'r', 't', 'h
7         String s = new String("hello");
8
9         // use String constructors
10        String s1 = new String();
11        String s2 = new String(s);
12        String s3 = new String(charArray);
13        String s4 = new String(charArray, 6, 3)
14
15        System.out.printf(
16          "s1 = %s%ns2 = %s%ns3 = %s%ns4 = %s%
17    }
18  }
```

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

Fig. 14.1

`String` class constructors.

Line 10 instantiates a new `String` using class `String`'s no-argument constructor and assigns its reference to `s1`. The new `String` object contains no characters (i.e., the **empty string**, which can also be represented as `""`) and has a length of 0.

Line 11 instantiates a new `String` object using class `String`'s constructor that takes a `String` object as an argument and assigns its reference to `s2`. The new `String` object contains the same sequence of characters as the `String` object `s` that's passed as an argument to the constructor.

Performance Tip 14.3

It's not necessary to copy an existing `String` *object.* `String` *objects are immutable, because class* `String` *does not provide methods that allow the contents of a* `String` *object to be modified after it is created. In fact, it's rarely necessary to call* `String` *constructors.*

Line 12 instantiates a new `String` object and assigns its reference to `s3` using class `String`'s constructor that takes a `char` array as an argument. The new `String` object contains a copy of the characters in the array.

Line 13 instantiates a new `String` object and assigns its reference to `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 accessed. Remember that the first character is at position 0. The third argument specifies the number of characters (the count) to access in the array. The new `String` object is formed from the accessed characters. If

the offset or the count specified as an argument results in accessing an element outside the bounds of the character array, a `StringIndexOutOfBoundsException` is thrown.

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

`String` methods `length`, `charAt` and `getChars` return the length of a `String`, obtain the character at a specific location in a `String` and retrieve a set of characters from a `String` as a `char` array, respectively. Figure 14.2 demonstrates each of these methods.

```
    1   // Fig. 14.2: StringMiscellaneous.java
2   // This application demonstrates the length, char
       3   // methods of the String class.
                        4
      5   public class StringMiscellaneous {
  6       public static void main(String[] args) {
        7         String s1 = "hello there";
     8          char[] charArray = new char[5];
                        9
   10          System.out.printf("s1: %s", s1);
                       11
         12         // test length method
13          System.out.printf("%nLength of s1: %d", s1
                       14
15          // loop through characters in s1 with char
16          System.out.printf("%nThe string reversed i
                       17
```

```
18          for (int count = s1.length() - 1; count >=
19             System.out.printf("%c ", s1.charAt(coun
                    20          }
                         21
22          // copy characters from string into charAr
   23          s1.getChars(0, 5, charArray, 0);
24          System.out.printf("%nThe character array i
                         25
  26          for (char character : charArray) {
    27             System.out.print(character);
                    28          }
                         29
          30          System.out.println();
                       31       }
                         32    }
```

```
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. 14.2

`String` methods `length`, `charAt` and `getChars`.

Line 13 uses `String` method `length` to determine the number of characters in `String s1`. Like arrays, strings know their own length. However, unlike arrays, you access a `String`'s length via class `String`'s `length` method.

Lines 18–20 print the characters of the `String s1` in reverse

order (and separated by spaces). `String` method `charAt` (line 19) returns the character at a specific position in the `String`. Method `charAt` receives an integer argument that's used as the index and returns the character at that position. Like arrays, the first element of a `String` is at position 0.

Line 23 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 to be copied. The second argument is the index that's 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. Next, lines 26–28 print the `char` array contents one character at a time.

14.3.3 Comparing Strings

Chapter 19 discusses sorting and searching arrays. Frequently, the information being sorted or searched consists of `String`s that must be compared to place them into order or to determine whether a string appears in an array (or other collection). Class `String` provides methods for *comparing* strings, as demonstrated in the next two examples.

To understand what it means for one string to be greater than or less than another, consider the process of alphabetizing a series of last names. No doubt, you'd 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’s 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—it’s specifically the twenty-sixth letter of the alphabet.

How does the computer know that one letter “comes before” another? All characters are represented in the computer as numeric codes (see Appendix B). When the computer compares `String`s, it actually compares the numeric codes of the characters in the `String`s.

Figure 14.3 demonstrates `String` methods `equals`, `equalsIgnoreCase`, `compareTo` and `regionMatches` and using the equality operator `==` to compare `String` objects.

```
1   // Fig. 14.3: StringCompare.java
2   // String methods equals, equalsIgnoreCase, comp
3
4   public class StringCompare {
5      public static void main(String[] args) {
6         String s1 = new String("hello"); // s1 is
7         String s2 = "goodbye";
8         String s3 = "Happy Birthday";
9         String s4 = "happy birthday";
10
11         System.out.printf(
12            "s1 = %s%ns2 = %s%ns3 = %s%ns4 = %s%n%n
13
14         // test for equality
15         if (s1.equals("hello")) {  // true
16            System.out.println("s1 equals \"hello\"
```

```
                }
            else {
System.out.println("s1 does not equal \
                }

        // test for equality with ==
    if (s1 == "hello") { // false; they are no
System.out.println("s1 is the same obje
                }
            else {
System.out.println("s1 is not the same
                }

      // test for equality (ignore case)
    if (s3.equalsIgnoreCase(s4)) { // true
System.out.printf("%s equals %s with ca
                }
            else {
System.out.println("s3 does not equal s
                }

        // test compareTo
        System.out.printf(
    "%ns1.compareTo(s2) is %d", s1.compareT
        System.out.printf(
    "%ns2.compareTo(s1) is %d", s2.compareT
        System.out.printf(
    "%ns1.compareTo(s1) is %d", s1.compareT
        System.out.printf(
    "%ns3.compareTo(s4) is %d", s3.compareT
        System.out.printf(
    "%ns4.compareTo(s3) is %d%n%n", s4.comp

    // test regionMatches (case sensitive)
     if (s3.regionMatches(0, s4, 0, 5)) {
        System.out.println("First 5 characters
                }
            else {
          System.out.println(
           "First 5 characters of s3 and s4 do
```

```
                    57         }
                         58
  59          // test regionMatches (ignore case)
60          if (s3.regionMatches(true, 0, s4, 0, 5)) {
        61             System.out.println(
62               "First 5 characters of s3 and s4 mat
                    63         }
                   64          else {
        65             System.out.println(
66               "First 5 characters of s3 and s4 do
                    67         }
                      68      }
                        69   }
```

```
                    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 ignore
```

```
                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 ignor
```

Fig. 14.3

`String` methods `equals`, `equalsIgnoreCase`, `compareTo` and `regionMatches`.

`String` Method `equals`

Line 15 uses method `equals` (an `Object` method overridden in `String`) to compare `String s1` and the `String` literal `"hello"` for equality. For `String`s, the method determines whether the contents of the two `String`s are *identical*. If so, it returns `true`; otherwise, it returns `false`. The preceding condition is `true` because `String s1` was initialized with the literal `"hello"`. Method `equals` uses a **lexicographical comparison**—it compares the integer Unicode values (see online Appendix H for more information) that represent each character in each `String`. Thus, if the `String "hello"` is compared to the string `"HELLO"`, the result is `false`, because the integer representation of a lowercase letter is *different* from that of the corresponding uppercase letter.

Comparing `Strings` with the == Operator

The condition at line 23 uses the equality operator `==` to

compare `String s1` for equality with the `String` literal `"hello"`. 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*. To compare the actual contents (or state information) of objects for equality, a method must be invoked. In the case of `String`s, that method is `equals`. The condition evaluates to `false` at line 23 because the reference `s1` was initialized with the statement

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

which creates a new `String` object with a copy of string literal `"hello"` and assigns the new object to variable `s1`. If `s1` had been initialized with the statement

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

which directly assigns the string literal `"hello"` to variable `s1`, the condition would be `true`. Remember that Java treats all string literal objects with the same contents as one `String` object to which there can be many references. Thus, the `"hello"` literals in lines 6, 15 and 23 all refer to the same `String` object.

Common Programming

Error 14.1

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 separate objects that contain the same values are compared with* `==`*, the result will be* `false`. *When comparing objects to determine whether they have the same contents, use method* `equals`.

`String` Method `equalsIgnoreCase`

When sorting `String`s, you might compare them for equality with method `equalsIgnoreCase`, which performs a case-insensitive comparison. Thus, `"hello"` and `"HELLO"` compare as equal. Line 31 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.

`String` Method `compareTo`

Lines 39–48 use method `compareTo` to compare `String`s.

Class `String` implements interface `Comparable` which declares method `compareTo`. Line 40 compares `String s1` to `String s2`. Method `compareTo` returns 0 if the `String`s are equal, a negative number if the `String` that invokes `compareTo` is less than the `String` that's passed as an argument and a positive number if the `String` that invokes `compareTo` is greater than the `String` that's passed as an argument. Method `compareTo` uses a *lexicographical* comparison—it compares the numeric values of corresponding characters in each `String`.

`String` Method `regionMatches`

The condition at line 51 uses a version of `String` method `regionMatches` to compare portions of two `String`s for equality. The first argument to this version of the method 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 `String`s. The method returns `true` only if the specified number of characters are lexicographically equal.

Finally, the condition at line 60 uses a five-argument version of `String` method `regionMatches` to compare portions of two `String`s for equality. 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.

`String` Methods `startsWith` and `endsWith`

Figure 14.4 demonstrates `String` methods `startsWith` and `endsWith`. Method `main` creates array `strings` containing `"started"`, `"starting"`, `"ended"` and `"ending"`. The remainder of method `main` consists of three `for` statements that test the elements of the array to determine whether they start with or end with a particular set of characters.

```
1   // Fig. 14.4: StringStartEnd.java
2   // String methods startsWith and endsWith.
3
4   public class StringStartEnd {
5      public static void main(String[] args) {
6         String[] strings = {"started", "starting",
7
8         // test method startsWith
9         for (String string : strings) {
10           if (string.startsWith("st")) {
11              System.out.printf("\"%s\" starts wit
12           }
13        }
14
15        System.out.println();
16
```

```
17         // test method startsWith starting from po
18         for (String string : strings) {
19            if (string.startsWith("art", 2)) {
20               System.out.printf(
21                  "\"%s\" starts with \"art\" at po
22            }
23         }
24
25         System.out.println();
26
27         // test method endsWith
28         for (String string : strings) {
29            if (string.endsWith("ed")) {
30               System.out.printf("\"%s\" ends with
31            }
32         }
33      }
34   }
```

```
"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. 14.4

`String` methods `startsWith` and `endsWith`.

Lines 9–13 use the version of method `startsWith` that

takes a `String` argument. The condition in the `if` statement (line 10) determines whether each `String` in the array starts with the characters `"st"`. If so, the method returns `true` and the application prints that `String`. Otherwise, the method returns `false` and nothing happens.

Lines 18–23 use the `startsWith` method that takes a `String` and an integer as arguments. The integer specifies the index at which the comparison should begin in the `String`. The condition in the `if` statement (line 19) determines whether each `String` in the array has the characters `"art"` beginning with the third character in each `String`. If so, the method returns `true` and the application prints the `String`.

The third `for` statement (lines 28–32) uses method `endsWith`, which takes a `String` argument. The condition at line 29 determines whether each `String` in the array ends with the characters `"ed"`. If so, the method returns `true` and the application prints the `String`.

14.3.4 Locating Characters and Substrings in Strings

Often it’s useful to search a string for a character or set of characters. For example, if you’re creating your own word processor, you might want to provide a capability for searching through documents. Figure 14.5 demonstrates the

many versions of `String` methods `indexOf` and `lastIndexOf` that search for a specified character or substring in a `String`.

```
// Fig. 14.5: StringIndexMethods.java
// String searching methods indexOf and lastInde

public class StringIndexMethods {
   public static void main(String[] args) {
      String letters = "abcdefghijklmabcdefghijk

      // test indexOf to locate a character in a
      System.out.printf(
         "'c' is located at index %d%n", letters
      System.out.printf(
         "'a' is located at index %d%n", letters
      System.out.printf(
         "'$' is located at index %d%n%n", lette

      // test lastIndexOf to find a character in
      System.out.printf("Last 'c' is located at
         letters.lastIndexOf('c'));
      System.out.printf("Last 'a' is located at
         letters.lastIndexOf('a', 25));
      System.out.printf("Last '$' is located at
         letters.lastIndexOf('$'));

      // test indexOf to locate a substring in a
      System.out.printf("\"def\" is located at i
         letters.indexOf("def"));
      System.out.printf("\"def\" is located at i
         letters.indexOf("def", 7));
      System.out.printf("\"hello\" is located at
         letters.indexOf("hello"));

      // test lastIndexOf to find a substring in
      System.out.printf("Last \"def\" is located
         letters.lastIndexOf("def"));
```

```
35          System.out.printf("Last \"def\" is located
  36             letters.lastIndexOf("def", 25));
37          System.out.printf("Last \"hello\" is locat
   38             letters.lastIndexOf("hello"));
                    39      }
                     40   }
```

```
'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. 14.5

String-searching methods `indexOf` and `lastIndexOf`.

All the searches in this example are performed on the `String letters` (initialized with `"abcdefghijklmabcdefghijklm"`). Lines 9–14 use

method `indexOf` to locate 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`. There are two versions of `indexOf` that search for characters in a `String`. The expression in line 10 uses the version of method `indexOf` that takes an integer representation of the character to find. The expression at line 12 uses another version of method `indexOf`, which takes two integer arguments—the character and the starting index at which the search of the `String` should begin.

Lines 17–22 use method `lastIndexOf` to locate 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. There are two versions of `lastIndexOf` that search for characters in a `String`. The expression at line 18 uses the version that takes the integer representation of the character. The expression at line 20 uses the version that takes two integer arguments—the integer representation of the character and the index from which to begin searching *backward*.

Lines 25–38 demonstrate versions of methods `indexOf` and `lastIndexOf` that each take a `String` as the first argument. These versions perform identically to those described earlier except that they search for sequences of characters (or substrings) that are specified by their `String` arguments. If the substring is found, these methods return the index in the `String` of the first character in the substring.

14.3.5 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. Both methods are demonstrated in Fig. 14.6.

```
1   // Fig. 14.6: SubString.java
2   // String class substring methods.
3
4   public class SubString {
5      public static void main(String[] args) {
6         String letters = "abcdefghijklmabcdefghijk
7
8         // test substring methods
9         System.out.printf("Substring from index 20
10            letters.substring(20));
11         System.out.printf("%s \"%s\"%n",
12            "Substring from index 3 up to, but not
13            letters.substring(3, 6));
14      }
15   }
```

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

Fig. 14.6

`String` class `substring` methods.

The expression `letters.substring(20)` at line 10 uses the `substring` method that takes one integer argument. The argument specifies the starting index in the original `String` `letters` from which characters are to be copied. The substring returned contains a copy of the characters from the starting index to the end of the `String`. Specifying an index outside the bounds of the `String` causes a `StringIndexOutOfBoundsException`.

Line 13 uses the `substring` method that takes two integer arguments—the starting index from which to copy characters in the original `String` and the index one beyond the last character to copy (i.e., copy up to, but *not including*, that index in the `String`). The substring returned contains a copy of the specified characters from the original `String`. An index outside the bounds of the `String` causes a `StringIndexOutOfBoundsException`.

14.3.6 Concatenating Strings

`String` method `concat` (Fig. 14.7) concatenates two `String` objects (similar to using the + operator) and returns a new `String` object containing the characters from both original `String`s. The expression `s1.concat(s2)` at line 11 forms a `String` by appending the characters in `s2` to the

those in `s1`. The original `String`s to which `s1` and `s2` refer are *not modified*.

```
1   // Fig. 14.7: StringConcatenation.java
2   // String method concat.
3
4   public class StringConcatenation {
5      public static void main(String[] args) {
6         String s1 = "Happy ";
7         String s2 = "Birthday";
8
9         System.out.printf("s1 = %s%ns2 = %s%n%n",s
10         System.out.printf(
11            "Result of s1.concat(s2) = %s%n", s1.co
12         System.out.printf("s1 after concatenation
13      }
14   }
```

```
s1 = Happy
s2 = Birthday

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

Fig. 14.7

`String` method `concat`.

14.3.7 Miscellaneous

`String` Methods

Class `String` provides several methods that return `String`s or character arrays containing modified copies of an original `String`'s contents. These methods—none of which modify the `String` on which they're called—are demonstrated in Fig. 14.8.

```
   1   // Fig. 14.8: StringMiscellaneous2.java
 2   // String methods replace, toLowerCase, toUpperC
                        3
     4   public class StringMiscellaneous2 {
  5      public static void main(String[] args) {
           6         String s1 = "hello";
          7         String s2 = "GOODBYE";
        8         String s3 = "  spaces  ";
                        9
10         System.out.printf("s1 = %s%ns2 = %s%ns3 =
                       11
        12         // test method replace
           13         System.out.printf(
14            "Replace 'l' with 'L' in s1: %s%n%n", s
                       15
  16         // test toLowerCase and toUpperCase
17         System.out.printf("s1.toUpperCase() = %s%n
18         System.out.printf("s2.toLowerCase() = %s%n
                       19
          20         // test trim method
21         System.out.printf("s3 after trim = \"%s\"%
                       22
      23         // test toCharArray method
 24         char[] charArray = s1.toCharArray();
25         System.out.print("s1 as a character array
                       26
  27         for (char character : charArray) {
    28            System.out.print(character);
```

```
29         }
30
31         System.out.println();
32      }
33   }
```

```
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. 14.8

`String` methods `replace`, `toLowerCase`, `toUpperCase`, `trim` and `toCharArray`.

Line 14 uses `String` method `replace` to return a new `String` object in which every occurrence in `s1` of character `'l'` (lowercase el) is replaced 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`. An overloaded

version of method `replace` enables you to replace substrings rather than individual characters.

Line 17 uses `String` method `toUpperCase` to generate a new `String` with uppercase letters where corresponding lowercase letters exist 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, method `toUpperCase` returns the original `String`.

Line 18 uses `String` method `toLowerCase` to return a new `String` object with lower-case letters where corresponding uppercase letters exist in `s2`. The original `String` remains unchanged. If there are no characters in the original `String` to convert, `toLowerCase` returns the original `String`.

Line 21 uses `String` method `trim` to generate a new `String` object that removes all white-space characters that appear at the beginning and/or end of the `String` on which `trim` operates. The method returns a new `String` object containing the `String` without leading or trailing white space. The original `String` remains unchanged. If there are no white-space characters at the beginning and end, `trim` returns the original `String`.

Line 24 uses `String` method `toCharArray` to create a new character array containing a copy of the characters in `s1`. Lines 27–29 output each `char` in the array.

14.3.8 `String` Method `valueOf`

As we've seen, every object in Java has a `toString` method that enables a program to obtain the object's *string representation*. Unfortunately, this technique cannot be used with primitive types because they do not have methods. Class `String` provides `static` methods that take an argument of any type and convert it to a `String` object. Figure 14.9 demonstrates the `String` class `valueOf` methods.

```
1   // Fig. 14.9: StringValueOf.java
2   // String valueOf methods.
3
4   public class StringValueOf {
5     public static void main(String[] args) {
6       char[] charArray = {'a', 'b', 'c', 'd', 'e
7       boolean booleanValue = true;
8       char characterValue = 'Z';
9       int integerValue = 7;
10      long longValue = 10000000000L; // L suffix
11      float floatValue = 2.5f; // f indicates th
12      double doubleValue = 33.333; // no suffix,
13      Object objectRef = "hello"; // assign stri
14
15      System.out.printf(
16        "char array = %s%n", String.valueOf(cha
17      System.out.printf("part of char array = %s
18        String.valueOf(charArray, 3, 3));
19      System.out.printf(
20        "boolean = %s%n", String.valueOf(boolea
21      System.out.printf(
22        "char = %s%n", String.valueOf(character
23      System.out.printf("int = %s%n", String.val
24      System.out.printf("long = %s%n", String.va
```

```
25          System.out.printf("float = %s%n", String.v
26          System.out.printf(
27              "double = %s%n", String.valueOf(doubleV
28          System.out.printf("Object = %s", String.va
29      }
30  }
```

```
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. 14.9

`String valueOf` methods.

The expression `String.valueOf(charArray)` at line 16 uses the character array `charArray` to create a new `String` object. The expression `String.valueOf(charArray, 3, 3)` at line 18 uses a portion of the character array `charArray` to create a new `String` object. The second argument specifies the starting index from which the characters are used. The third argument specifies the number of characters to be used.

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 19–28. The version of `valueOf` that takes an `Object` as an argument can do so because all `Object`s can be converted to `String`s with method `toString`.

[*Note:* Lines 10–11 use literal values `10000000000L` and `2.5f` as the initial values of `long` variable `longValue` and `float` variable `floatValue`, respectively. By default, Java treats integer literals as type `int` and floating-point literals as type `double`. Appending the letter `L` to the literal `10000000000` and appending the letter `f` to the literal `2.5` indicates to the compiler that `10000000000` should be treated as a `long` and `2.5` as a `float`. An upper-case `L` or lowercase `l` can be used to denote a variable of type `long` and an uppercase `F` or lowercase `f` can be used to denote a variable of type `float`.]