```
 9       System.out.println(n +"! is \n" + factorial(n));
10     }
11
12     public static BigInteger factorial(long n) {
13       BigInteger result = BigInteger.ONE;
14       for (int i = 1; i <= n; i++)
15         result = result.multiply(new BigInteger(i + ""));
16
17       return result;
18     }
19   }
```

constant

multiply

```
Enter an integer: 50 Enter
50! is
30414093201713378043612608166064768844377641568960512000000000000
```

`BigInteger.ONE` (line 13) is a constant defined in the `BigInteger` class. `BigInteger.ONE` is the same as `new BigInteger("1")`.

A new result is obtained by invoking the `multiply` method (line 15).

10.9.1 What is the output of the following code?

```
public class Test {
  public static void main(String[] args) {
    java.math.BigInteger x = new java.math.BigInteger("3");
    java.math.BigInteger y = new java.math.BigInteger("7");
    java.math.BigInteger z = x.add(y);
    System.out.println("x is " + x);
    System.out.println("y is " + y);
    System.out.println("z is " + z);
  }
}
```

10.10 The String Class

A `String` object is immutable; its contents cannot be changed once the string is created.

VideoNote

The `String` class

Strings were introduced in Section 4.4. You know strings are objects. You can invoke the `charAt(index)` method to obtain a character at the specified index from a string, the `length()` method to return the size of a string, the `substring` method to return a substring in a string, the `indexOf` and `lastIndexOf` methods to return the first or last index of a matching character or a substring, the `equals` and `compareTo` methods to compare two strings, and the `trim()` method to trim whitespace characters from the two ends of a string, and the `toLowerCase()` and `toUpperCase()` methods to return the lowercase and uppercase from a string. We will take a closer look at strings in this section.

The `String` class has 13 constructors and more than 40 methods for manipulating strings. Not only is it very useful in programming, but it is also a good example for learning classes and objects.

You can create a string object from a string literal or from an array of characters. To create a string from a string literal, use the syntax:

```
String newString = new String(stringLiteral);
```

The argument `stringLiteral` is a sequence of characters enclosed in double quotes. The following statement creates a `String` object `message` for the string literal `"Welcome to Java"`:

```
String message = new String("Welcome to Java");
```

Java treats a string literal as a `String` object. Thus, the following statement is valid: string literal object

```
String message = "Welcome to Java";
```

You can also create a string from an array of characters. For example, the following statements create the string `"Good Day"`:

```
char[] charArray = {'G', 'o', 'o', 'd', ' ', 'D', 'a', 'y'};
String message = new String(charArray);
```

Note

A `String` variable holds a reference to a `String` object that stores a string value. Strictly speaking, the terms `String` *variable*, `String` *object*, and *string value* are different, but most of the time the distinctions between them can be ignored. For simplicity, the term *string* will often be used to refer to `String` variable, `String` object, and string value.

`String` variable, `string` object, string value

10.10.1 Immutable Strings and Interned Strings

A `String` object is immutable; its contents cannot be changed. Does the following code change the contents of the string? immutable

```
String s = "Java";
s = "HTML";
```

The answer is no. The first statement creates a `String` object with the content `"Java"` and assigns its reference to `s`. The second statement creates a new `String` object with the content `"HTML"` and assigns its reference to `s`. The first `String` object still exists after the assignment, but it can no longer be accessed, because variable `s` now points to the new object, as shown in Figure 10.15.

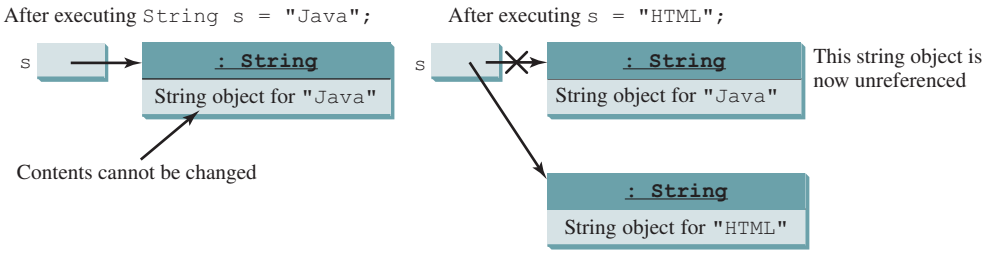


FIGURE 10.15 Strings are immutable; once created, their contents cannot be changed.

Because strings are immutable and are ubiquitous in programming, the JVM uses a unique instance for string literals with the same character sequence in order to improve efficiency and save memory. Such an instance is called an *interned string*. For example, the following statements: interned string

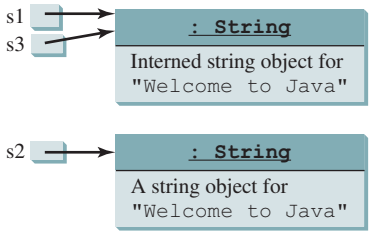
```
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";

String s4 = new String("Welcome to Java");

System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
System.out.println("s2 == s4 is " + (s2 == s4));
```



display

```
s1 == s2 is false
s1 == s3 is true
s2 == s4 is false
```

In the preceding statements, `s1` and `s3` refer to the same interned string—`"Welcome to Java"`—so `s1 == s3` is `true`. However, `s1 == s2` is `false`, because `s1` and `s2` are two different string objects, even though they have the same contents. `S2 == s4` is also `false`, because `s2` and `s4` are two different string objects.

Tip

You can create a `String` using `new String(stringLiteral)`. However, this is inefficient because it creates an unnecessary object. You should always simply use the `stringLiteral`. For example, use `String s = stringLiteral`; rather than `String s = new String(stringLiteral)`;

10.10.2 Replacing and Splitting Strings

The `String` class provides the methods for replacing and splitting strings, as shown in Figure 10.16.

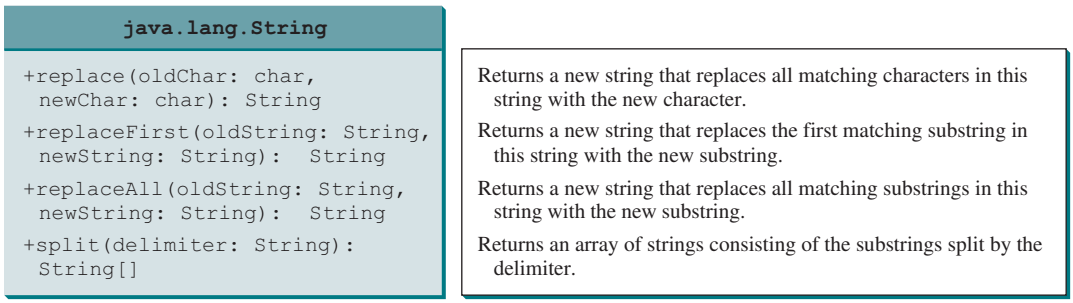


FIGURE 10.16 The `String` class contains the methods for replacing and splitting strings.

Once a string is created, its contents cannot be changed. The methods `replace`, `replaceFirst`, and `replaceAll` return a new string derived from the original string (without changing the original string!). Several versions of the `replace` methods are provided to replace a character or a substring in the string with a new character or a new substring.

For example,

replace — `"Welcome".replace('e', 'A')` returns a new string, `WAlcomA`.

replaceFirst — `"Welcome".replaceFirst("e", "AB")` returns a new string, `WABlcome`.

replace — `"Welcome".replace("e", "AB")` returns a new string, `WABlcomAB`.

replace — `"Welcome".replace("el", "AB")` returns a new string, `WABcome`.

`"Welcome".replaceAll("e", "AB")` returns a new string, `WABlcomAB`.

replaceAll — Note that `replaceAll(oldStr, newStr)` is the same as `replace(oldStr, newStr)` when used to replace all `oldStr` with `newStr`.

split — The `split` method can be used to extract tokens from a string with the specified delimiters. For example, the following code

```
String[] tokens = "Java#HTML#Perl".split("#");
for (int i = 0; i < tokens.length; i++)
  System.out.print(tokens[i] + " ");
```

displays

```
Java HTML Perl
```

10.10.3 Matching, Replacing, and Splitting by Patterns

why regular expression?

Often you will need to write code that validates user input, such as to check whether the input is a number, a string with all lowercase letters, or a Social Security number. How do you write this type of code? A simple and effective way to accomplish this task is to use the regular expression.

regular expression
regex

A *regular expression* (abbreviated *regex*) is a string that describes a pattern for matching a set of strings. You can match, replace, or split a string by specifying a pattern. This is an extremely useful and powerful feature.

matches(regex)

Let us begin with the `matches` method in the `String` class. At first glance, the `matches` method is very similar to the `equals` method. For example, the following two statements both evaluate to `true`:

```
"Java".matches("Java");
"Java".equals("Java");
```

However, the `matches` method is more powerful. It can match not only a fixed string, but also a set of strings that follow a pattern. For example, the following statements all evaluate to `true`:

```
"Java is fun".matches("Java.*")
"Java is cool".matches("Java.*")
"Java is powerful".matches("Java.*")
```

`Java.*` in the preceding statements is a regular expression. It describes a string pattern that begins with Java followed by *any* zero or more characters. Here, the substring matches any zero or more characters.

The following statement evaluates to `true`:

```
"440-02-4534".matches("\\d{3}-\\d{2}-\\d{4}")
```

Here, `\\d` represents a single digit, and `\\d{3}` represents three digits.

The `replaceAll`, `replaceFirst`, and `split` methods can be used with a regular expression. For example, the following statement returns a new string that replaces `$`, `+`, or `#` in `a+b$#c` with the string `NNN`.

```
String s = "a+b$#c".replaceAll("[$+#]", "NNN");
System.out.println(s);
```

replaceAll(regex)

Here, the regular expression `[$+#]` specifies a pattern that matches `$`, `+`, or `#`. Thus, the output is `aNNNbNNNNNNc`.

The following statement splits the string into an array of strings delimited by punctuation marks.

```
String[] tokens = "Java,C?C#,C++".split("[.,:;?]");

for (int i = 0; i < tokens.length; i++)
  System.out.println(tokens[i]);
```

split(regex)

In this example, the regular expression `[.,:;?]` specifies a pattern that matches `.`, `,`, `:`, `;`, or `?`. Each of these characters is a delimiter for splitting the string. Thus, the string is split into `Java`, `C`, `C#`, and `C++`, which are stored in array `tokens`.

further studies

Regular expression patterns are complex for beginning students to understand. For this reason, simple patterns are introduced in this section. Please refer to Appendix H, Regular Expressions, to learn more about these patterns.

10.10.4 Conversion between Strings and Arrays

toCharArray

Strings are not arrays, but a string can be converted into an array and vice versa. To convert a string into an array of characters, use the **toCharArray** method. For example, the following statement converts the string **Java** to an array:

```
char[] chars = "Java".toCharArray();
```

Thus, **chars[0]** is **J**, **chars[1]** is **a**, **chars[2]** is **v**, and **chars[3]** is **a**.

You can also use the **getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)** method to copy a substring of the string from index **srcBegin** to index **srcEnd–1** into a character array **dst** starting from index **dstBegin**. For example, the following code copies a substring **"3720"** in **"CS3720"** from index **2** to index **6–1** into the character array **dst** starting from index **4**:

getChars

```
char[] dst = {'J', 'A', 'V', 'A', '1', '3', '0', '1'};
"CS3720".getChars(2, 6, dst, 4);
```

Thus, **dst** becomes **{'J', 'A', 'V', 'A', '3', '7', '2', '0'}**.

To convert an array of characters into a string, use the **String(char[])** constructor or the **valueOf(char[])** method. For example, the following statement constructs a string from an array using the **String** constructor:

```
String str = new String(new char[]{'J', 'a', 'v', 'a'});
```

valueOf

The next statement constructs a string from an array using the **valueOf** method.

```
String str = String.valueOf(new char[]{'J', 'a', 'v', 'a'});
```

10.10.5 Converting Characters and Numeric Values to Strings

Recall that you can use **Double.parseDouble(str)** or **Integer.parseInt(str)** to convert a string to a **double** value or an **int** value, and you can convert a character or a number into a string by using the string concatenating operator. Another way of converting a number into a string is to use the overloaded static **valueOf** method. This method can also be used to convert a character or an array of characters into a string, as shown in Figure 10.17.

java.lang.String	
+valueOf(c: char): String	Returns a string consisting of the character c.
+valueOf(data: char[]): String	Returns a string consisting of the characters in the array.
+valueOf(d: double): String	Returns a string representing the double value.
+valueOf(f: float): String	Returns a string representing the float value.
+valueOf(i: int): String	Returns a string representing the int value.
+valueOf(l: long): String	Returns a string representing the long value.
+valueOf(b: boolean): String	Returns a string representing the boolean value.

FIGURE 10.17 The **String** class contains the static methods for creating strings from primitive-type values.

For example, to convert a **double** value **5.44** to a string, use **String.valueOf(5.44)**. The return value is a string consisting of the characters **'5'**, **'.'**, **'4'**, and **'4'**.

10.10.6 Formatting Strings

The **String** class contains the static **format** method to return a formatted string. The syntax to invoke this method is

```
String.format(format, item1, item2, ..., itemk);
```

This method is similar to the `printf` method except that the `format` method returns a formatted string, whereas the `printf` method displays a formatted string. For example,

```
String s = String.format("%7.2f%6d%-4s", 45.556, 14, "AB");
System.out.println(s);
```

displays

```
□□45.56□□□□14AB□□
```

where the square box (□) denotes a blank space.

Note

```
System.out.printf(format, item1, item2, ..., itemk);
```

is equivalent to

```
System.out.print(
  String.format(format, item1, item2, ..., itemk));
```

10.10.1 Suppose `s1`, `s2`, `s3`, and `s4` are four strings, given as follows:


```
String s1 = "Welcome to Java";
String s2 = s1;
String s3 = new String("Welcome to Java");
String s4 = "Welcome to Java";
```

What are the results of the following expressions?

a. `s1 == s2`
b. `s1 == s3`
c. `s1 == s4`
d. `s1.equals(s3)`
e. `s1.equals(s4)`
f. `"Welcome to Java".replace("Java", "HTML")`
g. `s1.replace('o', 'T')`
h. `s1.replaceAll("o", "T")`
i. `s1.replaceFirst("o", "T")`
j. `s1.toCharArray()`

10.10.2 To create the string `Welcome to Java`, you may use a statement like this:

```
String s = "Welcome to Java";
```

or

```
String s = new String("Welcome to Java");
```

Which one is better? Why?

10.10.3 What is the output of the following code?

```
String s1 = "Welcome to Java";
String s2 = s1.replace("o", "abc");
System.out.println(s1);
System.out.println(s2);
```

10.10.4 Let `s1` be `" Welcome "` and `s2` be `" welcome "`. Write the code for the following statements:

a. Replace all occurrences of the character `e` with `E` in `s1` and assign the new string to `s3`.

b. Split `Welcome to Java and HTML` into an array `tokens` delimited by a space and assign the first two tokens into `s1` and `s2`.

10.10.5 Does any method in the `String` class change the contents of the string?

10.10.6 Suppose string `s` is created using `new String()`; what is `s.length()`?

10.10.7 How do you convert a `char`, an array of characters, or a number to a string?

10.10.8 Why does the following code cause a `NullPointerException`?

```
1  public class Test {
2    private String text;
3
4    public Test(String s) {
5      String text = s;
6    }
7
8    public static void main(String[] args) {
9      Test test = new Test("ABC");
10     System.out.println(test.text.toLowerCase());
11   }
12 }
```

10.10.9 What is wrong in the following program?

```
1  public class Test {
2    String text;
3
4    public void Test(String s) {
5      text = s;
6    }
7
8    public static void main(String[] args) {
9      Test test = new Test("ABC");
10     System.out.println(test);
11   }
12 }
```

10.10.10 Show the output of the following code:

```
public class Test {
  public static void main(String[] args) {
    System.out.println("Hi, ABC, good".matches("ABC "));
    System.out.println("Hi, ABC, good".matches(".*ABC.*"));
    System.out.println("A,B;C".replaceAll(",;", "#"));
    System.out.println("A,B;C".replaceAll("[,;]", "#"));

    String[] tokens = "A,B;C".split("[,;]");
    for (int i = 0; i < tokens.length; i++)
      System.out.print(tokens[i] + " ");
  }
}
```

10.10.11 Show the output of the following code:

```
public class Test {
  public static void main(String[] args) {
    String s = "Hi, Good Morning";
    System.out.println(m(s));
  }

  public static int m(String s) {
    int count = 0;
    for (int i = 0; i < s.length(); i++)
      if (Character.isUpperCase(s.charAt(i)))
        count++;

    return count;
  }
}
```

10.11 The StringBuilder and StringBuffer Classes

The `StringBuilder` *and* `StringBuffer` *classes are similar to the* `String` *class except that the* `String` *class is immutable.*

In general, the `StringBuilder` and `StringBuffer` classes can be used wherever a string is used. `StringBuilder` and `StringBuffer` are more flexible than `String`. You can add, insert, or append new contents into `StringBuilder` and `StringBuffer` objects, whereas the value of a `String` object is fixed once the string is created.

`StringBuilder`

The `StringBuilder` class is similar to `StringBuffer` except that the methods for modifying the buffer in `StringBuffer` are *synchronized*, which means that only one task is allowed to execute the methods. Use `StringBuffer` if the class might be accessed by multiple tasks concurrently, because synchronization is needed in this case to prevent corruptions to `StringBuffer`. Concurrent programming will be introduced in Chapter 32. Using `StringBuilder` is more efficient if it is accessed by just a single task, because no synchronization is needed in this case. The constructors and methods in `StringBuffer` and `StringBuilder` are almost the same. This section covers `StringBuilder`. You can replace `StringBuilder` in all occurrences in this section by `StringBuffer`. The program can compile and run without any other changes.

`StringBuilder` constructors

The `StringBuilder` class has three constructors and more than 30 methods for managing the builder and modifying strings in the builder. You can create an empty string builder using `new StringBuilder()` or a string builder from a string using `new StringBuilder(String)`, as shown in Figure 10.18.

`java.lang.StringBuilder`	
`+StringBuilder()`	Constructs an empty string builder with capacity 16.
`+StringBuilder(capacity: int)`	Constructs a string builder with the specified capacity.
`+StringBuilder(s: String)`	Constructs a string builder with the specified string.

FIGURE 10.18 The `StringBuilder` class contains the constructors for creating instances of `StringBuilder`.

10.11.1 Modifying Strings in the `StringBuilder`

You can append new contents at the end of a string builder, insert new contents at a specified position in a string builder, and delete or replace characters in a string builder, using the methods listed in Figure 10.19.