9

STRINGS

Objectives

- To use the `String` class to process fixed strings (§9.2).
- To construct strings (§9.2.1).
- To understand that strings are immutable and to create an interned string (§9.2.2).
- To compare strings (§9.2.3).
- To get string length and characters, and combine strings (§9.2.4).
- To obtain substrings (§9.2.5).
- To convert, replace, and split strings (§9.2.6).
- To match, replace, and split strings by patterns (§9.2.7).
- To search for a character or substring in a string (§9.2.8).
- To convert between a string and an array (§9.2.9).
- To convert characters and numbers into a string (§9.2.10).
- To obtain a formatted string (§9.2.11).
- To check whether a string is a palindrome (§9.3).
- To convert hexadecimal numbers to decimal numbers (§9.4).
- To use the `Character` class to process a single character (§9.5).
- To use the `StringBuilder` and `StringBuffer` classes to process flexible strings (§9.6).
- To distinguish among the `String`, `StringBuilder`, and `StringBuffer` classes (§9.2–9.6).
- To learn how to pass arguments to the `main` method from the command line (§9.7).

9.1 Introduction

The classes **String**, **StringBuilder**, *and* **StringBuffer** *are used for processing strings.*

A *string* is a sequence of characters. Strings are frequently used in programming. In many languages, strings are treated as an array of characters, but in Java a string is treated as an object. This chapter introduces the classes for processing strings.

9.2 The `String` Class

A **String** *object is immutable: Its content cannot be changed once the string is created.*

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.

9.2.1 Constructing a String

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 inside double quotes. The following statement creates a **String** object **message** for the string literal **"Welcome to Java"**:

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

string literal object

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

```
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

9.2.2 Immutable Strings and Interned Strings

immutable

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

```
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 9.1.

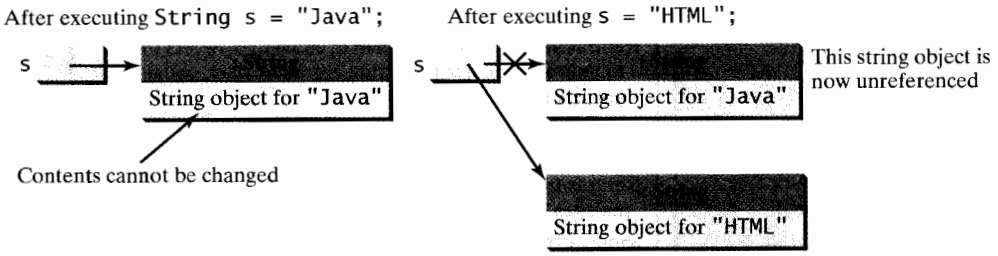
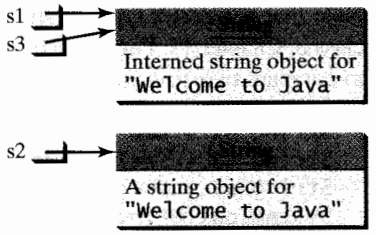


FIGURE 9.1 Strings are immutable; once created, their contents cannot be changed.

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



display

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

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.

9.2.3 String Comparisons

The **String** class provides the methods for comparing strings, as shown in Figure 9.2.

How do you compare the contents of two strings? You might attempt to use the == operator, as follows: ==

```
if (string1 == string2)
  System.out.println("string1 and string2 are the same object");
else
  System.out.println("string1 and string2 are different objects");
```

However, the == operator checks only whether **string1** and **string2** refer to the same object; it does not tell you whether they have the same contents. Therefore, you cannot use the == operator to find out whether two string variables have the same contents. Instead, you should use the **equals** method. The following code, for instance, can be used to compare two strings:

string1.equals(string2)

```
if (string1.equals(string2))
  System.out.println("string1 and string2 have the same contents");
else
  System.out.println("string1 and string2 are not equal");
```

java.lang.String	
+equals(s1: Object): boolean	Returns true if this string is equal to string s1.
+equalsIgnoreCase(s1: String): boolean	Returns true if this string is equal to string s1 case insensitive.
+compareTo(s1: String): int	Returns an integer greater than 0, equal to 0, or less than 0 to indicate whether this string is greater than, equal to, or less than s1.
+compareToIgnoreCase(s1: String): int	Same as compareTo except that the comparison is case insensitive.
+regionMatches(index: int, s1: String, s1Index: int, len: int): boolean	Returns true if the specified subregion of this string exactly matches the specified subregion in string s1.
+regionMatches(ignoreCase: boolean, index: int, s1: String, s1Index: int, len: int): boolean	Same as the preceding method except that you can specify whether the match is case sensitive.
+startsWith(prefix: String): boolean	Returns true if this string starts with the specified prefix.
+endsWith(suffix: String): boolean	Returns true if this string ends with the specified suffix.

FIGURE 9.2 The `String` class contains the methods for comparing strings.

Note that parameter type for the **`equals`** method is **`Object`**. We will introduce the **`Object`** class in Chapter 11. For now, you can replace **`Object`** by **`String`** for using the **`equals`** method to compare two strings. For example, the following statements display **`true`** and then **`false`**.

```
String s1 = new String("Welcome to Java");
String s2 = "Welcome to Java";
String s3 = "Welcome to C++";
System.out.println(s1.equals(s2)); // true
System.out.println(s1.equals(s3)); // false
```

The **`compareTo`** method can also be used to compare two strings. For example, consider the following code:

`s1.compareTo(s2)`

```
s1.compareTo(s2)
```

The method returns the value **`0`** if **`s1`** is equal to **`s2`**, a value less than **`0`** if **`s1`** is lexicographically (i.e., in terms of Unicode ordering) less than **`s2`**, and a value greater than **`0`** if **`s1`** is lexicographically greater than **`s2`**.

The actual value returned from the **`compareTo`** method depends on the offset of the first two distinct characters in **`s1`** and **`s2`** from left to right. For example, suppose **`s1`** is **`abc`** and **`s2`** is **`abg`**, and **`s1.compareTo(s2)`** returns **`-4`**. The first two characters (**`a`** vs. **`a`**) from **`s1`** and **`s2`** are compared. Because they are equal, the second two characters (**`b`** vs. **`b`**) are compared. Because they are also equal, the third two characters (**`c`** vs. **`g`**) are compared. Since the character **`c`** is **`4`** less than **`g`**, the comparison returns **`-4`**.

Caution

Syntax errors will occur if you compare strings by using comparison operators >, >=, <, or <=. Instead, you have to use **`s1.compareTo(s2)`**.

Note

The **`equals`** method returns **`true`** if two strings are equal and **`false`** if they are not. The **`compareTo`** method returns **`0`**, a positive integer, or a negative integer, depending on whether one string is equal to, greater than, or less than the other string.

The **`String`** class also provides the **`equalsIgnoreCase`**, **`compareToIgnoreCase`**, and **`regionMatches`** methods for comparing strings. The **`equalsIgnoreCase`** and

compareToIgnoreCase methods ignore the case of the letters when comparing two strings. The **regionMatches** method compares portions of two strings for equality. You can also use **str.startsWith(prefix)** to check whether string **str** starts with a specified prefix, and **str.endsWith(suffix)** to check whether string **str** ends with a specified suffix.

9.2.4 Getting String Length and Characters, and Combining Strings

The **String** class provides the methods for obtaining a string's length, retrieving individual characters, and concatenating strings, as shown in Figure 9.3.

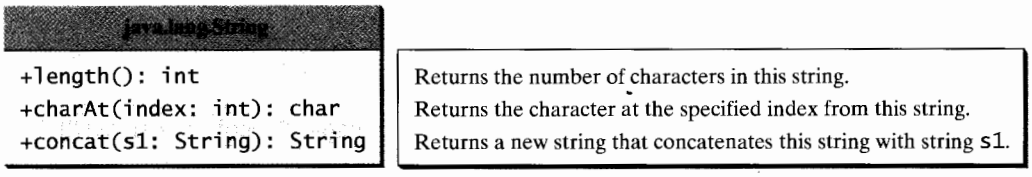


FIGURE 9.3 The **String** class contains the methods for getting string length, individual characters, and combining strings.

You can get the length of a string by invoking its **length()** method. For example, **message.length()** returns the length of the string **message**.

length()

Caution

length is a method in the **String** class but is a property of an array object. Therefore, you have to use **s.length()** to get the number of characters in string **s**, and **a.length** to get the number of elements in array **a**.

string length vs. array length

The **s.charAt(index)** method can be used to retrieve a specific character in a string **s**, where the index is between **0** and **s.length()-1**. For example, **message.charAt(0)** returns the character **W**, as shown in Figure 9.4.

charAt(index)

Note

When you use a string, you often know its literal value. For convenience, Java allows you to use the string literal to refer directly to strings without creating new variables. Thus, **"Welcome to Java".charAt(0)** is correct and returns **W**.

string literal

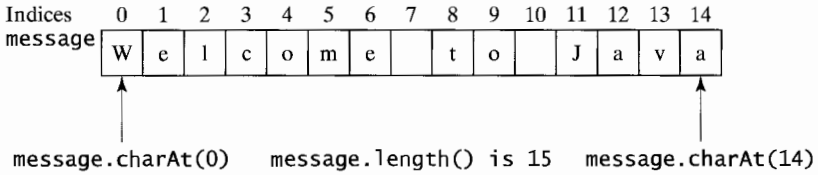


FIGURE 9.4 The characters in a **String** object are stored using an array internally.

Note

The **String** class uses an array to store characters internally. The array is private and cannot be accessed outside of the **String** class. The **String** class provides many public methods, such as **length()** and **charAt(index)**, to retrieve the string information. This is a good example of encapsulation: the data field of the class is hidden from the user through the private modifier, and thus the user cannot directly manipulate it. If the array were not private, the user would be able to change the string content by modifying the array. This would violate the tenet that the **String** class is immutable.

encapsulating string

string index range

Caution

Attempting to access characters in a string **s** out of bounds is a common programming error. To avoid it, make sure that you do not use an index beyond **s.length() - 1**. For example, **s.charAt(s.length())** would cause a **StringIndexOutOfBoundsException**.

You can use the **concat** method to concatenate two strings. The statement shown below, for example, concatenates strings **s1** and **s2** into **s3**:

s1.concat(s2)

```
String s3 = s1.concat(s2);
```

Because string concatenation is heavily used in programming, Java provides a convenient way to accomplish it. You can use the plus (+) operator to concatenate two strings, so the previous statement is equivalent to

s1 + s2

```
String s3 = s1 + s2;
```

The following code combines the strings **message**, **" and "**, and **"HTML"** into one string:

```
String myString = message + " and " + "HTML";
```

Recall that the + operator can also concatenate a number with a string. In this case, the number is converted into a string and then concatenated. Note that at least one of the operands must be a string in order for concatenation to take place.

9.2.5 Obtaining Substrings

You can obtain a single character from a string using the **charAt** method, as shown in Figure 9.3. You can also obtain a substring from a string using the **substring** method in the **String** class, as shown in Figure 9.5.

For example,

```
String message = "Welcome to Java".substring(0, 11) + "HTML";
```

The string **message** now becomes **Welcome to HTML**.

`+substring(beginIndex: int): String`	Returns this string's substring that begins with the character at the specified `beginIndex` and extends to the end of the string, as shown in Figure 9.6.
`+substring(beginIndex: int, endIndex: int): String`	Returns this string's substring that begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`, as shown in Figure 9.6. Note that the character at `endIndex` is not part of the substring.

FIGURE 9.5 The **String** class contains the methods for obtaining substrings.

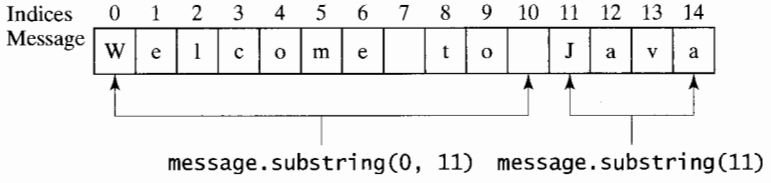


FIGURE 9.6 The **substring** method obtains a substring from a string.

Note

If **beginIndex** is **endIndex**, **substring(beginIndex, endIndex)** returns an empty string with length **0**. If **beginIndex** > **endIndex**, it would be a runtime error. `beginIndex <= endIndex`

9.2.6 Converting, Replacing, and Splitting Strings

The **String** class provides the methods for converting, replacing, and splitting strings, as shown in Figure 9.7.

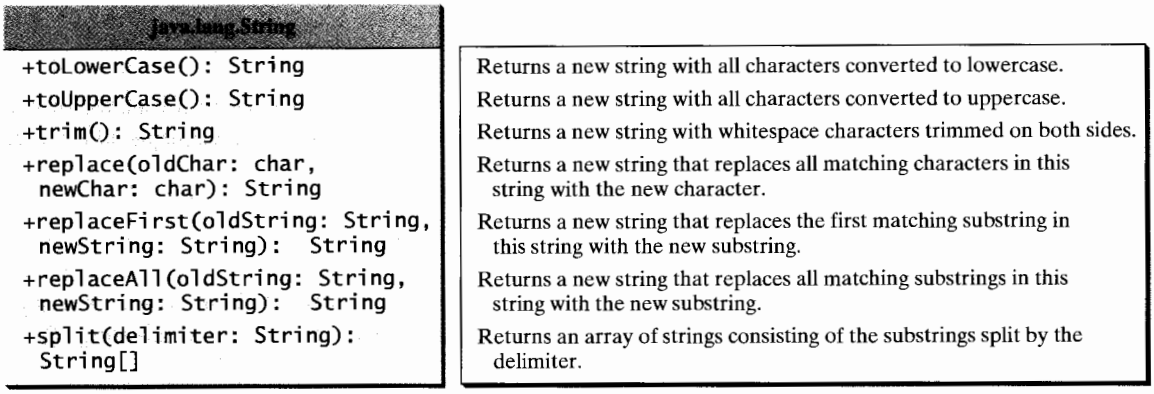


FIGURE 9.7 The **String** class contains the methods for converting, replacing, and splitting strings.

Once a string is created, its contents cannot be changed. The methods **toLowerCase**, **toUpperCase**, **trim**, **replace**, **replaceFirst**, and **replaceAll** return a new string derived from the original string (without changing the original string!). The **toLowerCase** and **toUpperCase** methods return a new string by converting all the characters in the string to lowercase or uppercase. The **trim** method returns a new string by eliminating whitespace characters from both ends of the 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,

"Welcome". toLowerCase() returns a new string, **welcome**.	`toLowerCase()`
"Welcome". toUpperCase() returns a new string, **WELCOME**.	`toUpperCase()`
"\t Good Night \n". trim() returns a new string, **Good Night**.	`trim()`
"Welcome". replace('e', 'A') returns a new string, **WAlcomA**.	`replace`
"Welcome". replaceFirst("e", "AB") returns a new string, **WABlcome**.	`replaceFirst`
"Welcome". replace("e", "AB") returns a new string, **WABlcomAB**.	`replace`
"Welcome". replace("el", "AB") returns a new string, **WABcome**.	`replace`

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

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

displays

Java HTML Perl

9.2.7 Matching, Replacing and Splitting by Patterns

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.

why 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**.

replaceAll(regex)

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

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

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

split(regex)

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

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

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 Supplement III.H, Regular Expressions, to learn more about these patterns.

9.2.8 Finding a Character or a Substring in a String

The **String** class provides several overloaded **indexOf** and **lastIndexOf** methods to find a character or a substring in a string, as shown in Figure 9.8.

`+indexOf(ch: char): int`	Returns the index of the first occurrence of `ch` in the string. Returns `-1` if not matched.
`+indexOf(ch: char, fromIndex: int): int`	Returns the index of the first occurrence of `ch` after `fromIndex` in the string. Returns `-1` if not matched.
`+indexOf(s: String): int`	Returns the index of the first occurrence of string `s` in this string. Returns `-1` if not matched.
`+indexOf(s: String, fromIndex: int): int`	Returns the index of the first occurrence of string `s` in this string after `fromIndex`. Returns `-1` if not matched.
`+lastIndexOf(ch: int): int`	Returns the index of the last occurrence of `ch` in the string. Returns `-1` if not matched.
`+lastIndexOf(ch: int, fromIndex: int): int`	Returns the index of the last occurrence of `ch` before `fromIndex` in this string. Returns `-1` if not matched.
`+lastIndexOf(s: String): int`	Returns the index of the last occurrence of string `s`. Returns `-1` if not matched.
`+lastIndexOf(s: String, fromIndex: int): int`	Returns the index of the last occurrence of string `s` before `fromIndex`. Returns `-1` if not matched.

FIGURE 9.8 The **String** class contains the methods for matching substrings.

For example,

indexOf

```
"Welcome to Java".indexOf('W') returns 0.
"Welcome to Java".indexOf('o') returns 4.
"Welcome to Java".indexOf('o', 5) returns 9.
"Welcome to Java".indexOf("come") returns 3.
"Welcome to Java".indexOf("Java", 5) returns 11.
"Welcome to Java".indexOf("java", 5) returns -1.
```

lastIndexOf

```
"Welcome to Java".lastIndexOf('W') returns 0.
"Welcome to Java".lastIndexOf('o') returns 9.
"Welcome to Java".lastIndexOf('o', 5) returns 4.
"Welcome to Java".lastIndexOf("come") returns 3.
"Welcome to Java".lastIndexOf("Java", 5) returns -1.
"Welcome to Java".lastIndexOf("Java") returns 11.
```

9.2.9 Conversion between Strings and Arrays

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.

toCharArray

```
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'});
```

9.2.10 Converting Characters and Numeric Values to Strings

overloaded valueOf

The static **valueOf** method can be used to convert an array of characters into a string. There are several overloaded versions of the **valueOf** method that can be used to convert a character and numeric values to strings with different parameter types, **char**, **double**, **long**, **int**, and **float**, as shown in Figure 9.9.

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 9.9 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'**.

Note

You can use **Double.parseDouble(str)** or **Integer.parseInt(str)** to convert a string to a **double** value or an **int** value. **Double** and **Integer** are two classes in the **java.lang** package.

9.2.11 Formatting Strings

The **String** class contains the static **format** method to create 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□□
```

Note that

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

is equivalent to

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

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

MyProgrammingLab™

9.1 Suppose that **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. `s2 == s3`
c. `s1.equals(s2)`
d. `s2.equals(s3)`
e. `s1.compareTo(s2)`
f. `s2.compareTo(s3)`
g. `s1 == s4`
h. `s1.charAt(0)`
i. `s1.indexOf('j')`
j. `s1.indexOf("to")`
k. `s1.lastIndexOf('a')`
l. `s1.lastIndexOf("o", 15)`
m. `s1.length()`
n. `s1.substring(5)`
o. `s1.substring(5, 11)`
p. `s1.startsWith("Wel")`
q. `s1.endsWith("Java")`
r. `s1.toLowerCase()`
s. `s1.toUpperCase()`
t. `"Welcome ".trim()`
u. `s1.replace('o', 'T')`
v. `s1.replaceAll("o", "T")`
w. `s1.replaceFirst("o", "T")`
x. `s1.toCharArray()`

9.2 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?

9.3 Suppose that **s1** and **s2** are two strings. Which of the following statements or expressions are incorrect?

```
String s = new String("new string");
String s3 = s1 + s2;
String s3 = s1 - s2;
s1 == s2;
s1 >= s2;
s1.compareTo(s2);
int i = s1.length();
char c = s1(0);
char c = s1.charAt(s1.length());
```

9.4 What is the printout of the following code?

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

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

a. Check whether **s1** is equal to **s2** and assign the result to a Boolean variable **isEqual**.

b. Check whether **s1** is equal to **s2**, ignoring case, and assign the result to a Boolean variable **isEqual**.

c. Compare **s1** with **s2** and assign the result to an **int** variable **x**.

d. Compare **s1** with **s2**, ignoring case, and assign the result to an **int** variable **x**.

e. Check whether **s1** has the prefix **AAA** and assign the result to a Boolean variable **b**.

f. Check whether **s1** has the suffix **AAA** and assign the result to a Boolean variable **b**.

g. Assign the length of **s1** to an **int** variable **x**.

h. Assign the first character of **s1** to a **char** variable **x**.

i. Create a new string **s3** that combines **s1** with **s2**.

j. Create a substring of **s1** starting from index **1**.

k. Create a substring of **s1** from index **1** to index **4**.

l. Create a new string **s3** that converts **s1** to lowercase.

m. Create a new string **s3** that converts **s1** to uppercase.

n. Create a new string **s3** that trims blank spaces on both ends of **s1**.

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

p. Split **Welcome to Java and HTML** into an array **tokens** delimited by a space.

q. Assign the index of the first occurrence of the character **e** in **s1** to an **int** variable **x**.

r. Assign the index of the last occurrence of the string **abc** in **s1** to an **int** variable **x**.

9.6 Does any method in the **String** class change the contents of the string?

9.7 Suppose string **s** is created using **new String()**; what is **s.length()**?

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

9.9 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 }
```

9.10 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 }
```

9.11 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] +  " ");
  }
}
```

9.3 Case Study: Checking Palindromes

This section presents a program that checks whether a string is a palindrome.

A string is a palindrome if it reads the same forward and backward. The words "mom," "dad," and "noon," for instance, are all palindromes.

VideoNote
Check palindrome

The problem is to write a program that prompts the user to enter a string and reports whether the string is a palindrome. One solution is to check whether the first character in the string is the same as the last character. If so, check whether the second character is the same as the second-to-last character. This process continues until a mismatch is found or all the characters in the string are checked, except for the middle character if the string has an odd number of characters.

To implement this idea, use two variables, say **low** and **high**, to denote the position of the two characters at the beginning and the end in a string **s**, as shown in Listing 9.1 (lines 22, 25). Initially, **low** is **0** and **high** is **s.length() - 1**. If the two characters at these positions match, increment **low** by **1** and decrement **high** by **1** (lines 31–32). This process continues until (**low >= high**) or a mismatch is found.

LISTING 9.1 CheckPalindrome.java

```
1  import java.util.Scanner;
2
3  public class CheckPalindrome {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a Scanner
7      Scanner input = new Scanner(System.in);
8
9      // Prompt the user to enter a string
10     System.out.print("Enter a string: ");
11     String s = input.nextLine();
12
13     if (isPalindrome(s))
```

input string

```
14        System.out.println(s + " is a palindrome");
15      else
16        System.out.println(s + " is not a palindrome");
17    }
18
19    /** Check if a string is a palindrome */
20    public static boolean isPalindrome(String s) {
21      // The index of the first character in the string
22      int low = 0;
23
24      // The index of the last character in the string
25      int high = s.length() - 1;
26
27      while (low < high) {
28        if (s.charAt(low) != s.charAt(high))
29          return false; // Not a palindrome
30
31        low++;
32        high--;
33      }
34
35      return true; // The string is a palindrome
36    }
37  }
```

low index

high index

update indices

```
Enter a string: noon Enter
noon is a palindrome
```

```
Enter a string: moon Enter
moon is not a palindrome
```

The **nextLine()** method in the **Scanner** class (line 11) reads a line into **s**, and then **isPalindrome(s)** checks whether **s** is a palindrome (line 13).

9.4 Case Study: Converting Hexadecimals to Decimals

This section presents a program that converts a hexadecimal number into a decimal number.

Section 5.7 gives a program that converts a decimal to a hexadecimal. How do you convert a hex number into a decimal?

Given a hexadecimal number $h_n h_{n-1} h_{n-2} \ldots h_2 h_1 h_0$, the equivalent decimal value is

$$h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \ldots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

For example, the hex number **AB8C** is

$$10 \times 16^3 + 11 \times 16^2 + 8 \times 16^1 + 12 \times 16^0 = 43916$$

Our program will prompt the user to enter a hex number as a string and convert it into a decimal using the following method:

```
public static int hexToDecimal(String hex)
```

A brute-force approach is to convert each hex character into a decimal number, multiply it by 16^i for a hex digit at the **i**'s position, and then add all the items together to obtain the equivalent decimal value for the hex number.

Note that

$$h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \ldots + h_1 \times 16^1 + h_0 \times 16^0$$

$$= (\ldots ((h_n \times 16 + h_{n-1}) \times 16 + h_{n-2}) \times 16 + \ldots + h_1) \times 16 + h_0$$

This observation, known as the Horner's algorithm, leads to the following efficient code for converting a hex string to a decimal number:

```
int decimalValue = 0;
for (int i = 0; i < hex.length(); i++) {
  char hexChar = hex.charAt(i);
  decimalValue = decimalValue * 16 + hexCharToDecimal(hexChar);
}
```

Here is a trace of the algorithm for hex number **AB8C**:

	i	hexChar	hexCharToDecimal(hexChar)	decimalValue
before the loop				0
after the 1st iteration	0	A	10	10
after the 2nd iteration	1	B	11	10 * 16 + 11
after the 3rd iteration	2	8	8	(10 * 16 + 11) * 16 + 8
after the 4th iteration	3	C	12	((10 * 16 + 11) * 16 + 8) * 16 + 12

Listing 9.2 gives the complete program.

LISTING 9.2 HexToDecimalConversion.java

```
1  import java.util.Scanner;
2
3  public class HexToDecimalConversion {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a Scanner
7      Scanner input = new Scanner(System.in);
8
9      // Prompt the user to enter a string
10     System.out.print("Enter a hex number: ");
11     String hex = input.nextLine();                          input string
12
13     System.out.println("The decimal value for hex number "
14       + hex + " is " + hexToDecimal(hex.toUpperCase()));    hex to decimal
15   }
16
17   public static int hexToDecimal(String hex) {
18     int decimalValue = 0;
19     for (int i = 0; i < hex.length(); i++) {
20       char hexChar = hex.charAt(i);
21       decimalValue = decimalValue * 16 + hexCharToDecimal(hexChar);
22     }
23
24     return decimalValue;
25   }
26
```

hex char to decimal
to uppercase

```
27   public static int hexCharToDecimal(char ch) {
28     if (ch >= 'A' && ch <= 'F')
29       return 10 + ch - 'A';
30     else // ch is '0', '1', ..., or '9'
31       return ch - '0';
32   }
33 }
```

```
Enter a hex number: AB8C Enter
The decimal value for hex number AB8C is 43916
```

```
Enter a hex number: af71 Enter
The decimal value for hex number af71 is 44913
```

The program reads a string from the console (line 11), and invokes the **hexToDecimal** method to convert a hex string to decimal number (line 14). The characters can be in either lowercase or uppercase. They are converted to uppercase before invoking the **hexToDecimal** method.

The **hexToDecimal** method is defined in lines 17–25 to return an integer. The length of the string is determined by invoking **hex.length()** in line 19.

The **hexCharToDecimal** method is defined in lines 27–32 to return a decimal value for a hex character. The character can be in either lowercase or uppercase. Recall that to subtract two characters is to subtract their Unicodes. For example, **'5' - '0'** is **5**.

9.5 The `Character` Class

You can create an object for a character using the **Character** *class. A* **Character** *object contains a character value.*

Many methods in the Java API require an object argument. To enable the primitive data values to be treated as objects, Java provides a class for every primitive data type. These classes are **Character**, **Boolean**, **Byte**, **Short**, **Integer**, **Long**, **Float**, and **Double** for **char**, **boolean**, **byte**, **short**, **int**, **long**, **float**, and **double**, respectively. These classes are called *wrapper classes* because each wraps or encapsulates a primitive type value in an object. All these classes are in the **java.lang** package, and they contain useful methods for processing primitive values. This section introduces the **Character** class. The other wrapper classes will be introduced in Chapter 10, Thinking in Objects.

wrapper class

The **Character** class has a constructor and several methods for determining a character's category (uppercase, lowercase, digit, and so on) and for converting characters from uppercase to lowercase, and vice versa, as shown in Figure 9.10.

You can create a **Character** object from a **char** value. For example, the following statement creates a **Character** object for the character **a**.

```
Character character = new Character('a');
```

The **charValue** method returns the character value wrapped in the **Character** object. The **compareTo** method compares this character with another character and returns an integer that is the difference between the Unicode of this character and the Unicode of the other character. The **equals** method returns **true** if and only if the two characters are the same. For example, suppose **charObject** is **new Character('b')**:

```
charObject.compareTo(new Character('a')) returns 1
charObject.compareTo(new Character('b')) returns 0
charObject.compareTo(new Character('c')) returns -1
```

`charObject.compareTo(new Character('d'))` returns `-2`
`charObject.equals(new Character('b'))` returns `true`
`charObject.equals(new Character('d'))` returns `false`

java.lang.Character	
+Character(value: char)	Constructs a character object with char value.
+charValue(): char	Returns the char value from this object.
+compareTo(anotherCharacter: Character): int	Compares this character with another.
+equals(anotherCharacter: Character): boolean	Returns true if this character is equal to another.
+isDigit(ch: char): boolean	Returns true if the specified character is a digit.
+isLetter(ch: char): boolean	Returns true if the specified character is a letter.
+isLetterOrDigit(ch: char): boolean	Returns true if the character is a letter or a digit.
+isLowerCase(ch: char): boolean	Returns true if the character is a lowercase letter.
+isUpperCase(ch: char): boolean	Returns true if the character is an uppercase letter.
+toLowerCase(ch: char): char	Returns the lowercase of the specified character.
+toUpperCase(ch: char): char	Returns the uppercase of the specified character.

FIGURE 9.10 The **Character** class provides the methods for manipulating a character.

Most of the methods in the **Character** class are static methods. The **isDigit(char ch)** method returns **true** if the character is a digit, and the **isLetter(char ch)** method returns **true** if the character is a letter. The **isLetterOrDigit(char ch)** method returns **true** if the character is a letter or a digit. The **isLowerCase(char ch)** method returns **true** if the character is a lowercase letter, and the **isUpperCase(char ch)** method returns **true** if the character is an uppercase letter. The **toLowerCase(char ch)** method returns the lowercase letter for the character, and the **toUpperCase(char ch)** method returns the uppercase letter for the character.

Now let's write a program that prompts the user to enter a string and counts the number of occurrences of each letter in the string regardless of case.

Here are the steps to solve this problem:

1. Convert all the uppercase letters in the string to lowercase using the **toLowerCase** method in the **String** class.
2. Create an array, say **counts** of **26 int** values, each of which counts the occurrences of a letter. That is, **counts[0]** counts the number of **a**s, **counts[1]** counts the number of **b**s, and so on.
3. For each character in the string, check whether it is a (lowercase) letter. If so, increment the corresponding count in the array.

Listing 9.3 gives the complete program.

LISTING 9.3 CountEachLetter.java

```
1  import java.util.Scanner;
2
3  public class CountEachLetter {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a Scanner
7      Scanner input = new Scanner(System.in);
8
```

input string

count letters

count a letter

```
 9      // Prompt the user to enter a string
10      System.out.print("Enter a string: ");
11      String s = input.nextLine();
12
13      // Invoke the countLetters method to count each letter
14      int[] counts = countLetters(s.toLowerCase());
15
16      // Display results
17      for (int i = 0; i < counts.length; i++) {
18        if (counts[i] != 0)
19          System.out.println((char)('a' + i) + " appears  " +
20            counts[i] + ((counts[i] == 1) ? " time" : " times"));
21      }
22    }
23
24    /** Count each letter in the string */
25    public static int[] countLetters(String s) {
26      int[] counts = new int[26];
27
28      for (int i = 0; i < s.length(); i++) {
29        if (Character.isLetter(s.charAt(i)))
30          counts[s.charAt(i) - 'a']++;
31      }
32
33      return counts;
34    }
35  }
```

```
Enter a string: abababx Enter
a appears  3 times
b appears  3 times
x appears  1 time
```

The main method reads a line (line 11) and counts the number of occurrences of each letter in the string by invoking the **countLetters** method (line 14). Since the case of the letters is ignored, the program uses the **toLowerCase** method to convert the string into all lowercase and pass the new string to the **countLetters** method.

The **countLetters** method (lines 25–34) returns an array of **26** elements. Each element counts the number of occurrences of a letter in the string **s**. The method processes each character in the string. If the character is a letter, its corresponding count is increased by **1**. For example, if the character (**s.charAr(i)**) is **a**, the corresponding count is **counts['a' - 'a']** (i.e., **counts[0]**). If the character is **b**, the corresponding count is **counts['b' - 'a']** (i.e., **counts[1]**), since the Unicode of **b** is **1** more than that of **a**. If the character is **z**, the corresponding count is **counts['z' - 'a']** (i.e., **counts[25]**), since the Unicode of **z** is **25** more than that of **a**.

9.12 How do you determine whether a character is in lowercase or uppercase?

9.13 How do you determine whether a character is alphanumeric?

MyProgrammingLab™

9.14 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;
  }
}
```

9.6 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. Concurrent programming will be introduced in Chapter 32. Using **StringBuilder** is more efficient if it is accessed by just a single task. 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 or a string builder from a string using the constructors, as shown in Figure 9.11.

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 9.11 The **StringBuilder** class contains the constructors for creating instances of **StringBuilder**.

9.6.1 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 9.12.

The **StringBuilder** class provides several overloaded methods to append **boolean**, **char**, **char[]**, **double**, **float**, **int**, **long**, and **String** into a string builder. For example, the following code appends strings and characters into **stringBuilder** to form a new string, **Welcome to Java**.

```
StringBuilder stringBuilder = new StringBuilder();
stringBuilder.append("Welcome");
stringBuilder.append(' ');
stringBuilder.append("to");
stringBuilder.append(' ');
stringBuilder.append("Java");
```

append

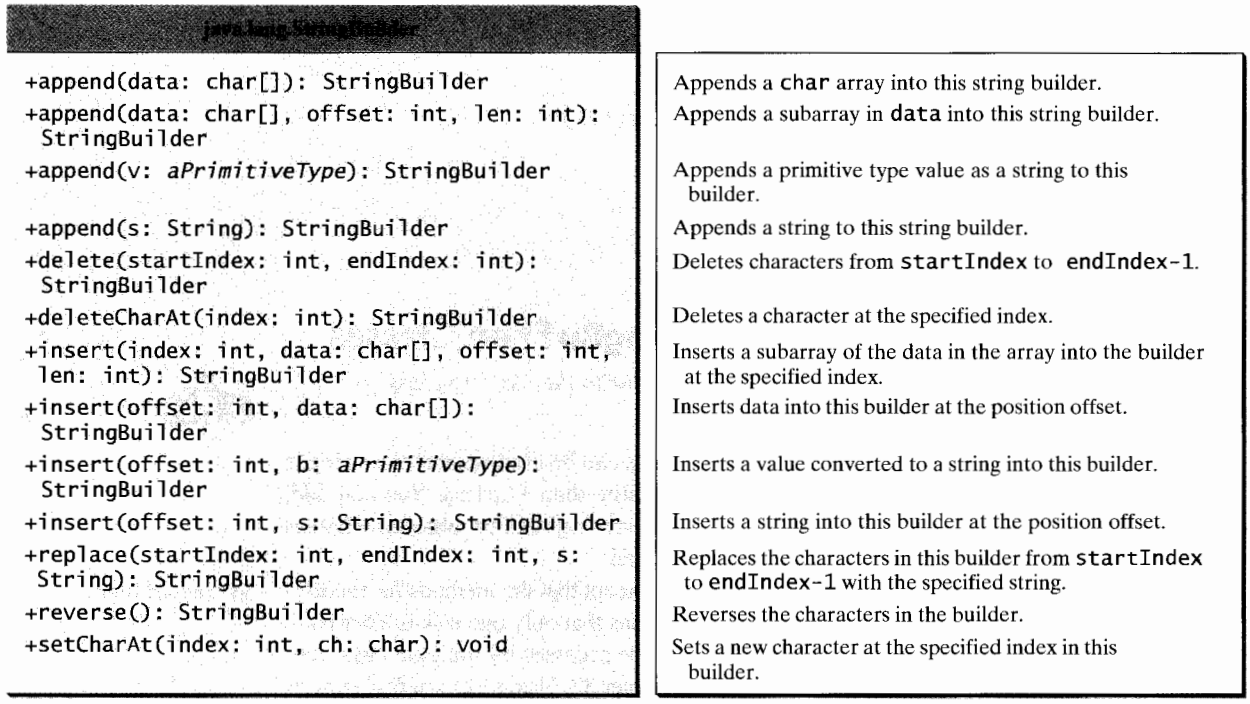


FIGURE 9.12 The `StringBuilder` class contains the methods for modifying string builders.

The `StringBuilder` class also contains overloaded methods to insert `boolean`, `char`, `char array`, `double`, `float`, `int`, `long`, and `String` into a string builder. Consider the following code:

insert

```
stringBuilder.insert(11, "HTML and ");
```

Suppose `stringBuilder` contains `Welcome to Java` before the `insert` method is applied. This code inserts `"HTML and "` at position 11 in `stringBuilder` (just before the `J`). The new `stringBuilder` is `Welcome to HTML and Java`.

You can also delete characters from a string in the builder using the two `delete` methods, reverse the string using the `reverse` method, replace characters using the `replace` method, or set a new character in a string using the `setCharAt` method.

For example, suppose `stringBuilder` contains `Welcome to Java` before each of the following methods is applied:

delete
deleteCharAt
reverse
replace
setCharAt

`stringBuilder.delete(8, 11)` changes the builder to `Welcome Java`.
`stringBuilder.deleteCharAt(8)` changes the builder to `Welcome o Java`.
`stringBuilder.reverse()` changes the builder to `avaJ ot emocleW`.
`stringBuilder.replace(11, 15, "HTML")` changes the builder to `Welcome to HTML`.
`stringBuilder.setCharAt(0, 'w')` sets the builder to `welcome to Java`.

All these modification methods except `setCharAt` do two things:

- Change the contents of the string builder
- Return the reference of the string builder

ignore return value

For example, the following statement

```
StringBuilder stringBuilder1 = stringBuilder.reverse();
```

reverses the string in the builder and assigns the builder's reference to **stringBuilder1**. Thus, **stringBuilder** and **stringBuilder1** both point to the same **StringBuilder** object. Recall that a value-returning method can be invoked as a statement, if you are not interested in the return value of the method. In this case, the return value is simply ignored. For example, in the following statement

```
stringBuilder.reverse();
```

the return value is ignored.

Tip

If a string does not require any change, use **String** rather than **StringBuilder**. Java can perform some optimizations for **String**, such as sharing interned strings.

String or StringBuilder?

9.6.2 The toString, capacity, length, setLength, and charAt Methods

The **StringBuilder** class provides the additional methods for manipulating a string builder and obtaining its properties, as shown in Figure 9.13.

+toString(): String	Returns a string object from the string builder.
+capacity(): int	Returns the capacity of this string builder.
+charAt(index: int): char	Returns the character at the specified index.
+length(): int	Returns the number of characters in this builder.
+setLength(newLength: int): void	Sets a new length in this builder.
+substring(startIndex: int): String	Returns a substring starting at **startIndex**.
+substring(startIndex: int, endIndex: int): String	Returns a substring from **startIndex** to **endIndex-1**.
+trimToSize(): void	Reduces the storage size used for the string builder.

FIGURE 9.13 The **StringBuilder** class contains the methods for modifying string builders.

The **capacity()** method returns the current capacity of the string builder. The capacity is the number of characters the string builder is able to store without having to increase its size.

capacity()

The **length()** method returns the number of characters actually stored in the string builder. The **setLength(newLength)** method sets the length of the string builder. If the **newLength** argument is less than the current length of the string builder, the string builder is truncated to contain exactly the number of characters given by the **newLength** argument. If the **newLength** argument is greater than or equal to the current length, sufficient null characters (**\u0000**) are appended to the string builder so that **length** becomes the **newLength** argument. The **newLength** argument must be greater than or equal to **0**.

length()
setLength(int)

The **charAt(index)** method returns the character at a specific **index** in the string builder. The index is **0** based. The first character of a string builder is at index **0**, the next at index **1**, and so on. The **index** argument must be greater than or equal to **0**, and less than the length of the string builder.

charAt(int)

Note

The length of the string is always less than or equal to the capacity of the builder. The length is the actual size of the string stored in the builder, and the capacity is the current size of the builder. The builder's capacity is automatically increased if more characters are added to exceed its capacity. Internally, a string builder is an array of characters, so

length and capacity

the builder's capacity is the size of the array. If the builder's capacity is exceeded, the array is replaced by a new array. The new array size is **2 * (the previous array size + 1)**.

Tip

initial capacity

You can use **new StringBuilder(initialCapacity)** to create a **StringBuilder** with a specified initial capacity. By carefully choosing the initial capacity, you can make your program more efficient. If the capacity is always larger than the actual length of the builder, the JVM will never need to reallocate memory for the builder. On the other hand, if the capacity is too large, you will waste memory space. You can use the **trimToSize()** method to reduce the capacity to the actual size.

trimToSize()

9.6.3 Case Study: Ignoring Nonalphanumeric Characters When Checking Palindromes

Listing 9.1, CheckPalindrome.java, considered all the characters in a string to check whether it was a palindrome. Write a new program that ignores nonalphanumeric characters in checking whether a string is a palindrome.

Here are the steps to solve the problem:

1. Filter the string by removing the nonalphanumeric characters. This can be done by creating an empty string builder, adding each alphanumeric character in the string to a string builder, and returning the string from the string builder. You can use the **isLetterOrDigit(ch)** method in the **Character** class to check whether character **ch** is a letter or a digit.
2. Obtain a new string that is the reversal of the filtered string. Compare the reversed string with the filtered string using the **equals** method.

The complete program is shown in Listing 9.4.

LISTING 9.4 PalindromeIgnoreNonAlphanumeric.java

```
1  import java.util.Scanner;
2
3  public class PalindromeIgnoreNonAlphanumeric {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a Scanner
7      Scanner input = new Scanner(System.in);
8
9      // Prompt the user to enter a string
10     System.out.print("Enter a string: ");
11     String s = input.nextLine();
12
13     // Display result
14     System.out.println("Ignoring nonalphanumeric characters, \nis "
15       + s + " a palindrome? " + isPalindrome(s));
16   }
17
18   /** Return true if a string is a palindrome */
19   public static boolean isPalindrome(String s) {
20     // Create a new string by eliminating nonalphanumeric chars
21     String s1 = filter(s);
22
23     // Create a new string that is the reversal of s1
24     String s2 = reverse(s1);
25
26     // Check if the reversal is the same as the original string
```

check palindrome

```
27      return s2.equals(s1);
28    }
29
30    /** Create a new string by eliminating nonalphanumeric chars */
31    public static String filter(String s) {
32      // Create a string builder
33      StringBuilder stringBuilder = new StringBuilder();
34
35      // Examine each char in the string to skip alphanumeric char
36      for (int i = 0; i < s.length(); i++) {
37        if (Character.isLetterOrDigit(s.charAt(i))) {
38          stringBuilder.append(s.charAt(i));                    add letter or digit
39        }
40      }
41
42      // Return a new filtered string
43      return stringBuilder.toString();
44    }
45
46    /** Create a new string by reversing a specified string */
47    public static String reverse(String s) {
48      StringBuilder stringBuilder = new StringBuilder(s);
49      stringBuilder.reverse(); // Invoke reverse in StringBuilder
50      return stringBuilder.toString();
51    }
52  }
```

```
Enter a string: ab<c>cb?a Enter
Ignoring nonalphanumeric characters,
is ab<c>cb?a a palindrome? true
```

```
Enter a string: abcc><?cab Enter
Ignoring nonalphanumeric characters,
is abcc><?cab a palindrome? false
```

The **filter(String s)** method (lines 31–44) examines each character in string **s** and copies it to a string builder if the character is a letter or a numeric character. The **filter** method returns the string in the builder. The **reverse(String s)** method (lines 47–51) creates a new string that reverses the specified string **s**. The **filter** and **reverse** methods both return a new string. The original string is not changed.

The program in Listing 9.1 checks whether a string is a palindrome by comparing pairs of characters from both ends of the string. Listing 9.4 uses the **reverse** method in the **StringBuilder** class to reverse the string, then compares whether the two strings are equal to determine whether the original string is a palindrome.

MyProgrammingLab™

9.15 What is the difference between **StringBuilder** and **StringBuffer**?

9.16 How do you create a string builder from a string? How do you return a string from a string builder?

9.17 Write three statements to reverse a string **s** using the **reverse** method in the **StringBuilder** class.

9.18 Write three statements to delete a substring from a string **s** of **20** characters, starting at index **4** and ending with index **10**. Use the **delete** method in the **StringBuilder** class.

9.19 What is the internal storage for characters in a string and a string builder?

9.20 Suppose that **s1** and **s2** are given as follows:

```
StringBuilder s1 = new StringBuilder("Java");
StringBuilder s2 = new StringBuilder("HTML");
```

Show the value of **s1** after each of the following statements. Assume that the statements are independent.

```
a. s1.append(" is fun");
b. s1.append(s2);
c. s1.insert(2, "is fun");
d. s1.insert(1, s2);
e. s1.charAt(2);
f. s1.length();
g. s1.deleteCharAt(3);
h. s1.delete(1, 3);
i. s1.reverse();
j. s1.replace(1, 3, "Computer");
k. s1.substring(1, 3);
l. s1.substring(2);
```

9.21 Show the output of the following program:

```
public class Test {
  public static void main(String[] args) {
    String s = "Java";
    StringBuilder builder = new StringBuilder(s);
    change(s, builder);

    System.out.println(s);
    System.out.println(builder);
  }

  private static void change(String s, StringBuilder builder) {
    s = s + " and HTML";
    builder.append(" and HTML");
  }
}
```

9.7 Command-Line Arguments

The **main** *method can receive string arguments from the command line.*

Perhaps you have already noticed the unusual header for the **main** method, which has the parameter **args** of **String[]** type. It is clear that **args** is an array of strings. The **main** method is just like a regular method with a parameter. You can call a regular method by passing actual parameters. Can you pass arguments to **main**? Yes, of course you can. In the following examples, the **main** method in class **TestMain** is invoked by a method in **A**.

```
public class A {
  public static void main(String[] args) {
    String[] strings = {"New York",
      "Boston", "Atlanta"};
    TestMain.main(strings);
  }
}
```

```
public class TestMain {
  public static void main(String[] args) {
    for (int i = 0; i < args.length; i++)
      System.out.println(args[i]);
  }
}
```

A **main** method is just a regular method. Furthermore, you can pass arguments from the command line.

9.7.1 Passing Strings to the `main` Method

You can pass strings to a **`main`** method from the command line when you run the program. The following command line, for example, starts the program **`TestMain`** with three strings: **`arg0`**, **`arg1`**, and **`arg2`**:

```
java TestMain arg0 arg1 arg2
```

`arg0`, **`arg1`**, and **`arg2`** are strings, but they don't have to appear in double quotes on the command line. The strings are separated by a space. A string that contains a space must be enclosed in double quotes. Consider the following command line:

```
java TestMain "First num" alpha 53
```

It starts the program with three strings: **`First num`**, **`alpha`**, and **`53`**. Since **`First num`** is a string, it is enclosed in double quotes. Note that **`53`** is actually treated as a string. You can use **`"53"`** instead of **`53`** in the command line.

When the **`main`** method is invoked, the Java interpreter creates an array to hold the command-line arguments and pass the array reference to **`args`**. For example, if you invoke a program with **`n`** arguments, the Java interpreter creates an array like this one:

```
args = new String[n];
```

The Java interpreter then passes **`args`** to invoke the **`main`** method.

Note

If you run the program with no strings passed, the array is created with **`new String[0]`**. In this case, the array is empty with length **`0`**. **`args`** references to this empty array. Therefore, **`args`** is not **`null`**, but **`args.length`** is **`0`**.

9.7.2 Case Study: Calculator

VideoNote
Command-line argument

Suppose you are to develop a program that performs arithmetic operations on integers. The program receives an expression in one string argument. The expression consists of an integer followed by an operator and another integer. For example, to add two integers, use this command:

```
java Calculator "2 + 3"
```

The program will display the following output:

```
2 + 3 = 5
```

Figure 9.14 shows sample runs of the program.

The strings passed to the main program are stored in **`args`**, which is an array of strings. In this case, we pass the expression as one string. Therefore, the array contains only one element in **`args[0]`** and **`args.length`** is **`1`**.

Here are the steps in the program:

1. Use **`args.length`** to determine whether the expression has been provided as one argument in the command line. If not, terminate the program using **`System.exit(1)`**.
2. Split the expression in the string **`args[0]`** into three tokens in **`tokens[0]`**, **`tokens[1]`**, and **`tokens[2]`**.
3. Perform a binary arithmetic operation on the operands **`tokens[0]`** and **`tokens[2]`** using the operator in **`tokens[1]`**.

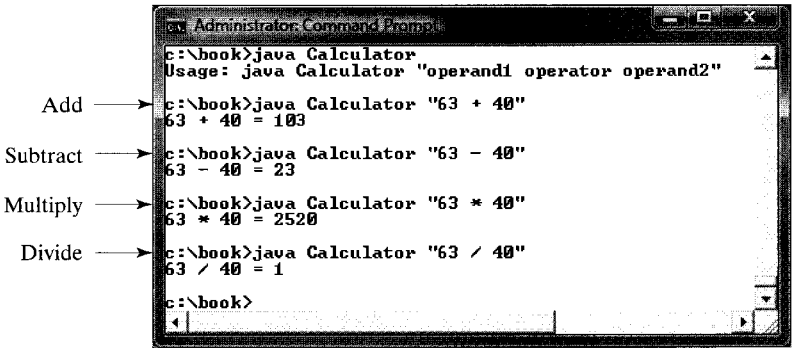


FIGURE 9.14 The program takes an expression in one argument (**operand1 operator operand2**) from the command line and displays the expression and the result of the arithmetic operation.

The program is shown in Listing 9.5.

LISTING 9.5 Calculator.java

```
1  public class Calculator {
2    /** Main method */
3    public static void main(String[] args) {
4      // Check number of strings passed
5      if (args.length != 1) {
6        System.out.println(
7          "Usage: java Calculator \"operand1 operator operand2\"");
8        System.exit(1);
9      }
10
11     // The result of the operation
12     int result = 0;
13
14     // The result of the operation
15     String[] tokens = args[0].split(" ");
16
17     // Determine the operator
18     switch (tokens[1].charAt(0) ) {
19       case '+': result = Integer.parseInt(tokens[0]) +
20                          Integer.parseInt(tokens[2]);
21                 break;
22       case '-': result = Integer.parseInt(tokens[0]) -
23                          Integer.parseInt(tokens[2]);
24                 break;
25       case '*': result = Integer.parseInt(tokens[0]) *
26                          Integer.parseInt(tokens[2]);
27                 break;
28       case '/': result = Integer.parseInt(tokens[0]) /
29                          Integer.parseInt(tokens[2]);
30     }
31
32     // Display result
33     System.out.println(tokens[0] + ' ' + tokens[1] + ' '
34       + tokens[2] + " = " + result);
35   }
36 }
```

check argument

split string

check operator

The expression is passed as a string in one argument and it is split into three parts—**tokens[0]**, **tokens[1]**, and **tokens[2]**—using the **split** method (line 15) with a space as a delimiter.

Integer.parseInt(tokens[0]) (line 19) converts a digital string into an integer. The string must consist of digits. If it doesn't, the program will terminate abnormally.

For this program to work, the expression must be entered in the form of "operand1 operator operand2". The operands and operator are separated by exactly one space. You can modify the program to accept the expressions in different forms (see Programming Exercise 9.28).

9.22 This book declares the **main** method as

MyProgrammingLab™

```
public static void main(String[] args)
```

Can it be replaced by one of the following lines?

```
public static void main(String args[])
public static void main(String[] x)
public static void main(String x[])
static void main(String x[])
```

9.23 Show the output of the following program when invoked using

1. **java Test I have a dream**
2. **java Test "1 2 3"**
3. **java Test**

```
public class Test {
  public static void main(String[] args) {
    System.out.println("Number of strings is " + args.length);
    for (int i = 0; i < args.length; i++)
      System.out.println(args[i]);
  }
}
```

KEY TERMS

interned string 337

wrapper class 350

CHAPTER SUMMARY

1. Strings are objects encapsulated in the **String** class. A string can be constructed using one of the 13 constructors or simply using a string literal. Java treats a string literal as a **String** object.

2. A **String** object is immutable; its contents cannot be changed. To improve efficiency and save memory, the JVM stores two literal strings that have the same character sequence in a unique object. This unique object is called an *interned string object*.

3. You can get the length of a string by invoking its **length()** method, retrieve a character at the specified **index** in the string using the **charAt(index)** method, and use the **indexOf** and **lastIndexOf** methods to find a character or a substring in a string.

4. You can use the **concat** method to concatenate two strings, or the plus (+) operator to concatenate two or more strings.

5. You can use the **substring** method to obtain a substring from the string.

6. You can use the **equals** and **compareTo** methods to compare strings. The **equals** method returns **true** if two strings are equal, and **false** if they are not equal. The **compareTo** method returns **0**, a positive integer, or a negative integer, depending on whether one string is equal to, greater than, or less than the other string.

7. 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.

8. The **Character** class is a wrapper class for a single character. The **Character** class provides useful static methods to determine whether a character is a letter (**isLetter(char)**), a digit (**isDigit(char)**), uppercase (**isUpperCase(char)**), or lowercase (**isLowerCase(char)**).

9. The **StringBuilder** and **StringBuffer** classes can be used to replace the **String** class. The **String** object is immutable, but you can add, insert, or append new contents into **StringBuilder** and **StringBuffer** objects. Use **String** if the string contents do not require any change, and use **StringBuilder** or **StringBuffer** if they might change.

10. You can pass strings to the **main** method from the command line. Strings passed to the **main** program are stored in **args**, which is an array of strings. The first string is represented by **args[0]**, and **args.length** is the number of strings passed.

Test Questions

Do the test questions for this chapter online at www.cs.armstrong.edu/liang/intro9e/test.html.

MyProgrammingLab™

Programming Exercises

Sections 9.2–9.3

***9.1** (*Check SSN*) Write a program that prompts the user to enter a Social Security number in the format DDD-DD-DDDD, where D is a digit. The program displays **Valid SSN** for a correct Social Security number and **Invalid SSN** otherwise.

****9.2** (*Check substrings*) You can check whether a string is a substring of another string by using the **indexOf** method in the **String** class. Write your own method for this function. Write a program that prompts the user to enter two strings, and checks whether the first string is a substring of the second.

****9.3** (*Check password*) Some websites impose certain rules for passwords. Write a method that checks whether a string is a valid password. Suppose the password rules are as follows:

- A password must have at least eight characters.
- A password consists of only letters and digits.
- A password must contain at least two digits.

Write a program that prompts the user to enter a password and displays **Valid Password** if the rules are followed or **Invalid Password** otherwise.

9.4 (*Occurrences of a specified character*) Write a method that finds the number of occurrences of a specified character in a string using the following header:

```
public static int count(String str, char a)
```

For example, `count("Welcome", 'e')` returns `2`. Write a test program that prompts the user to enter a string followed by a character and displays the number of occurrences of the character in the string.

****9.5** (*Occurrences of each digit in a string*) Write a method that counts the occurrences of each digit in a string using the following header:

```
public static int[] count(String s)
```

The method counts how many times a digit appears in the string. The return value is an array of ten elements, each of which holds the count for a digit. For example, after executing `int[] counts = count("12203AB3")`, `counts[0]` is `1`, `counts[1]` is `1`, `counts[2]` is `2`, and `counts[3]` is `2`.

Write a test program that prompts the user to enter a string and displays the number of occurrences of each digit in the string.

***9.6** (*Count the letters in a string*) Write a method that counts the number of letters in a string using the following header:

```
public static int countLetters(String s)
```

Write a test program that prompts the user to enter a string and displays the number of letters in the string.

***9.7** (*Phone keypads*) The international standard letter/number mapping found on the telephone is:

1	2 ABC	3 DEF
4 GHI	5 JKL	6 MNO
7 PQRS	8 TUV	9 WXYZ
	0	

Write a method that returns a number, given an uppercase letter, as follows:

```
public static int getNumber(char uppercaseLetter)
```

Write a test program that prompts the user to enter a phone number as a string. The input number may contain letters. The program translates a letter (upper- or lowercase) to a digit and leaves all other characters intact. Here is a sample run of the program:

```
Enter a string: 1-800-Flowers Enter
1-800-3569377
```

```
Enter a string: 1800flowers
18003569377
```

*9.8 (*Binary to decimal*) Write a method that parses a binary number as a string into a decimal integer. The method header is:

```
public static int binaryToDecimal(String binaryString)
```

For example, binary string 10001 is 17 ($1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2 + 1 = 17$). Therefore, **binaryToDecimal("10001")** returns **17**. Note that **Integer.parseInt("10001", 2)** parses a binary string to a decimal value. Do not use this method in this exercise.

Write a test program that prompts the user to enter a binary string and displays the corresponding decimal integer value.

Section 9.4

**9.9 (*Binary to hex*) Write a method that parses a binary number into a hex number. The method header is:

```
public static String binaryToHex(String binaryValue)
```

Write a test program that prompts the user to enter a binary number and displays the corresponding hexadecimal value.

VideoNote
Number conversion

**9.10 (*Decimal to binary*) Write a method that parses a decimal number into a binary number as a string. The method header is:

```
public static String decimalToBinary(int value)
```

Write a test program that prompts the user to enter a decimal integer value and displays the corresponding binary value.

**9.11 (*Sort characters in a string*) Write a method that returns a sorted string using the following header:

```
public static String sort(String s)
```

For example, **sort("acb")** returns **abc**.

Write a test program that prompts the user to enter a string and displays the sorted string.

9.12 (*Anagrams*) Write a method that checks whether two words are anagrams. Two words are anagrams if they contain the same letters in any order. For example, **silent and **listen** are anagrams. The header of the method is:

```
public static boolean isAnagram(String s1, String s2)
```

Write a test program that prompts the user to enter two strings and, if they are anagrams, displays **two strings are anagrams**, and displays **two strings are not anagrams** if they are not anagrams.

Section 9.5

*9.13 (*Pass a string to check palindromes*) Rewrite Listing 9.1 by passing the string as a command-line argument.

*9.14 (*Sum integers*) Write two programs. The first program passes an unspecified number of integers as separate strings to the **main** method and displays their total. The

second program passes an unspecified number of integers delimited by one space in a string to the **main** method and displays their total. Name the two programs **Exercise9_14a** and **Exercise9_14b**, as shown in Figure 9.15.

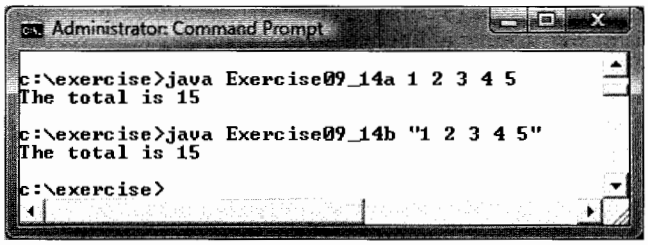


FIGURE 9.15 The program adds all the numbers passed from the command line.

***9.15** (*Find the number of uppercase letters in a string*) Write a program that passes a string to the **main** method and displays the number of uppercase letters in the string.

Comprehensive

****9.16** (*Implement the **String** class*) The **String** class is provided in the Java library. Provide your own implementation for the following methods (name the new class **MyString1**):

```
public MyString1(char[] chars);
public char charAt(int index);
public int length();
public MyString1 substring(int begin, int end);
public MyString1 toLowerCase();
public boolean equals(MyString1 s);
public static MyString1 valueOf(int i);
```

****9.17** (*Guess the capitals*) Write a program that repeatedly prompts the user to enter a capital for a state. Upon receiving the user input, the program reports whether the answer is correct. Assume that **50** states and their capitals are stored in a two-dimensional array, as shown in Figure 9.16. The program prompts the user to answer all states' capitals and displays the total correct count. The user's answer is not case-sensitive.

Alabama	Montgomery
Alaska	Juneau
Arizona	Phoenix
...	...
...	...

FIGURE 9.16 A two-dimensional array stores states and their capitals.

Here is a sample run:

```
What is the capital of Alabama? Montogomery Enter
The correct answer should be Montgomery
What is the capital of Alaska? Juneau Enter
Your answer is correct
What is the capital of Arizona? ...
...
The correct count is 35
```

****9.18** (*Implement the* ***String*** *class*) The **String** class is provided in the Java library. Provide your own implementation for the following methods (name the new class **MyString2**):

```
public MyString2(String s);
public int compare(String s);
public MyString2 substring(int begin);
public MyString2 toUpperCase();
public char[] toChars();
public static MyString2 valueOf(boolean b);
```

***9.19** (*Common prefix*) Write a method that returns the longest common prefix of two strings. For example, the longest common prefix of **distance** and **disinfection** is **dis**. The header of the method is:

```
public static String prefix(String s1, String s2)
```

If the two strings don't have a common prefix, the method returns an empty string.

Write a **main** method that prompts the user to enter two strings and displays their longest common prefix.

9.20 (*Implement the* ***Character*** *class*) The **Character** class is provided in the Java library. Provide your own implementation for this class. Name the new class **MyCharacter**.

****9.21** (*New string* ***split*** *method*) The **split** method in the **String** class returns an array of strings consisting of the substrings split by the delimiters. However, the delimiters are not returned. Implement the following new method that returns an array of strings consisting of the substrings split by the matching delimiters, including the matching delimiters.

```
public static String[] split(String s, String regex)
```

For example, **split("ab#12#453", "#")** returns **ab**, **#**, **12**, **#**, **453** in an array of **String**, and **split("a?b?gf#e", "[?#]")** returns **a**, **b**, **?**, **b**, **gf**, **#**, and **e** in an array of **String**.

****9.22** (*Implement the* ***StringBuilder*** *class*) The **StringBuilder** class is provided in the Java library. Provide your own implementation for the following methods (name the new class **MyStringBuilder1**):

```
public MyStringBuilder1(String s);
public MyStringBuilder1 append(MyStringBuilder1 s);
public MyStringBuilder1 append(int i);
public int length();
public char charAt(int index);
public MyStringBuilder1 toLowerCase();
public MyStringBuilder1 substring(int begin, int end);
public String toString();
```

****9.23** (*Financial: credit card number validation*) Rewrite Programming Exercise 5.31 using a string input for the credit card number. Redesign the program using the following methods:

```
/** Return true if the card number is valid */
public static boolean isValid(String cardNumber)

/** Get the result from Step 2 */
public static int sumOfDoubleEvenPlace(String cardNumber)
```

```
/** Return this number if it is a single digit; otherwise,
 * return the sum of the two digits */
public static int getDigit(int number)

/** Return sum of odd-place digits in number */
public static int sumOfOddPlace(String cardNumber)
```

9.24 (*Implement the* ***StringBuilder* *class*) The **StringBuilder** class is provided in the Java library. Provide your own implementation for the following methods (name the new class **MyStringBuilder2**):

```
public MyStringBuilder2();
public MyStringBuilder2(char[] chars);
public MyStringBuilder2(String s);
public MyStringBuilder2 insert(int offset, MyStringBuilder2 s);
public MyStringBuilder2 reverse();
public MyStringBuilder2 substring(int begin);
public MyStringBuilder2 toUpperCase();
```

***9.25 (*Game: hangman*) Write a hangman game that randomly generates a word and prompts the user to guess one letter at a time, as shown in the sample run. Each letter in the word is displayed as an asterisk. When the user makes a correct guess, the actual letter is then displayed. When the user finishes a word, display the number of misses and ask the user whether to continue to play with another word. Declare an array to store words, as follows:

```
// Add any words you wish in this array
String[] words = {"write", "that", ...};
```

```
(Guess) Enter a letter in word ******* > p Enter
(Guess) Enter a letter in word p****** > r Enter
(Guess) Enter a letter in word pr**r** > p Enter
     p is already in the word
(Guess) Enter a letter in word pr**r** > o Enter
(Guess) Enter a letter in word pro*r** > g Enter
(Guess) Enter a letter in word progr** > n Enter
     n is not in the word
(Guess) Enter a letter in word progr** > m Enter
(Guess) Enter a letter in word progr*m > a Enter
The word is program. You missed 1 time
Do you want to guess another word? Enter y or n>
```

**9.26 (*Check ISBN-10*) Use string operations to simplify Programming Exercise 3.9. Enter the first 9 digits of an ISBN number as a string.

VideoNote
Check ISBN-10

*9.27 (*Bioinformatics: find genes*) Biologists use a sequence of the letters A, C, T, and G to model a *genome*. A *gene* is a substring of a genome that starts after a triplet ATG and ends before a triplet TAG, TAA, or TGA. Furthermore, the length of a gene string is a multiple of 3, and the gene does not contain any of the triplets ATG, TAG, TAA, or TGA. Write a program that prompts the user to enter a genome and displays all genes in the genome. If no gene is found in the input sequence, display "no gene is found". Here are the sample runs:

```
Enter a genome string: TTATGTTTTAAGGATGGGGCGTTAGTT Enter
TTT
GGGCGT
```

```
Enter a genome string: TGTGTGTATAT ↵Enter
no gene is found
```

***9.28** (*Calculator*) Revise Listing 9.5, Calculator.java, to accept an expression in which the operands and operator are separated by zero or more spaces. For example, **3+4** and **3 + 4** are acceptable expressions.

***9.29** (*Business: check ISBN-13*) **ISBN-13** is a new standard for identifying books. It uses the 13 digits $d_1d_2d_3d_4d_5d_6d_7d_8d_9d_{10}d_{11}d_{12}d_{13}$. The last digit, d_{13}, is a checksum, which is calculated from the other digits using the following formula:

$$10 - (d_1 + 3d_2 + d_3 + 3d_4 + d_5 + 3d_6 + d_7 + 3d_8 + d_9 + 3d_{10} + d_{11} + 3d_{12}) \% 10$$

If the checksum is **10**, replace it with **0**. Your program should read the input as a string. Here are sample runs:

```
Enter the first 12 digits of an ISBN-13 as a string:
  978013213080 ↵Enter
The ISBN-13 number is 9780132130806
```

```
Enter the first 12 digits of an ISBN-13 as a string:
  978013213079 ↵Enter
The ISBN-13 number is 9780132130790
```

***9.30** (*Capitalize first letter of each word*) Write the following method that returns a new string in which the first letter in each word is capitalized.

```
public static void title(String s)
```

Write a test program that prompts the user to enter a string and invokes this method, and displays the return value from this method. Here is a sample run:

```
Enter a string: london   england 2015 ↵Enter
The new string is: London   England 2015
```

Note that words may be separated by multiple blank spaces.

***9.31** (*Swap case*) Write the following method that returns a new string in which the uppercase letters are changed to lowercase and lowercase letters are changed to uppercase.

```
public static String swapCase(String s)
```

Write a test program that prompts the user to enter a string and invokes this method, and displays the return value from this method. Here is a sample run:

```
Enter a string: I'm here ↵Enter
The new string is: i'M HERE
```