CHAPTER 7

Characters, Strings, and the StringBuilder

In this chapter, you will:

- Identify string data problems
- Manipulate characters
- Declare and compare `String` objects
- Use other `String` methods
- Convert `String` objects to numbers
- Use the `StringBuilder` and `StringBuffer` classes

Understanding String Data Problems

Manipulating characters and strings provides some challenges for the beginning Java programmer. For example, consider the TryToCompareStrings application shown in Figure 7-1. The main() method declares a String named aName and assigns "Carmen" to it. The user is then prompted to enter a name. The application compares the two names using the equivalency operator (==) and displays one of two messages indicating whether the strings are equivalent.


```
import java.util.Scanner;
public class TryToCompareStrings
{
   public static void main(String[] args)
   {
      String aName = "Carmen";
      String anotherName;
      Scanner input = new Scanner(System.in);
      System.out.print("Enter your name > ");
      anotherName = input.nextLine();
      if(aName == anotherName)
         System.out.println(aName + " equals " + anotherName);
      else
         System.out.println(aName + " does not equal " + anotherName);
   }
}
```

Don't Do It
Do not use == to compare Strings' contents.

Figure 7-1 The TryToCompareStrings application

Figure 7-2 shows the execution of the application. When the user types "Carmen" as the value for anotherName, the application concludes that the two names are not equal.

Figure 7-2 Execution of the TryToCompareStrings application

The application in Figure 7-1 seems to produce incorrect results. The problem stems from the fact that in Java, String is a class, and each created String is an object. As an object, a String variable name is not a simple data type—it is a **reference**; that is, a variable that holds a memory address. Therefore, when you compare two String objects using the == operator, you are comparing not their values but their computer memory locations.

Programmers want to compare the contents of memory locations (the values stored there) more frequently than they want to compare the locations themselves (the addresses). Fortunately, the creators of Java have provided three classes that you can use when working with text data; these classes provide you with many methods that make working with characters and strings easier:

- **`Character`**—A class whose instances can hold a single character value and whose methods manipulate and inspect single-character data
- **`String`**—A class for working with fixed-string data—that is, unchanging data composed of multiple characters
- `StringBuilder` and `StringBuffer`—Classes for storing and manipulating changeable data composed of multiple characters

TWO TRUTHS & A LIE

Understanding String Data Problems

1. A `String` is a simple data type that can hold text data.
2. Programmers want to compare the values of `String`s more frequently than they want to compare their memory addresses.
3. `Character`, `String`, and `StringBuilder` are useful built-in classes for working with text data.

The false statement is #1. A `String` variable name is a reference; that is, it holds a memory address.

Manipulating Characters

You learned in Chapter 2 that the `char` data type is used to hold any single character—for example, a letter, digit, or punctuation mark. In addition to the primitive data type `char`, Java offers a `Character` class. The `Character` class contains standard methods for testing the values of characters. Table 7-1 describes many of the `Character` class methods. The methods that begin with "is", such as `isUpperCase()`, return a Boolean value that can be used in comparison statements; the methods that begin with "to", such as `toUpperCase()`, return a character that has been converted to the stated format.

Method	Description
isUpperCase()	Tests if character is uppercase
toUpperCase()	Returns the uppercase equivalent of the argument; no change is made if the argument is not a lowercase letter
isLowerCase()	Tests if character is lowercase
toLowerCase()	Returns the lowercase equivalent of the argument; no change is made if the argument is not an uppercase letter
isDigit()	Returns true if the argument is a digit (0–9) and false otherwise
isLetter()	Returns true if the argument is a letter and false otherwise
isLetterOrDigit()	Returns true if the argument is a letter or digit and false otherwise
isWhitespace()	Returns true if the argument is whitespace and false otherwise; this includes the space, tab, newline, carriage return, and form feed

Table 7-1 Commonly used methods of the Character class

The Character class is defined in java.lang and is automatically imported into every program you write. The Character class inherits from java.lang.Object. You will learn more about the Object class when you study inheritance concepts in the chapter *Introduction to Inheritance*.

Figure 7-3 contains an application that uses many of the methods shown in Table 7-1. The application asks a user to enter a character. A String is accepted and the charAt() method is used to extract the first character in the user-entered String. (The charAt() method belongs to the String class; you will learn more about this class and method later in this chapter.) The application determines the attributes of the character and displays information about it.

```
import java.util.Scanner;
public class TestCharacter
{
   public static void main(String[] args)
   {
      char aChar;
      String aString;
      Scanner keyboard = new Scanner(System.in);
      System.out.print("Enter a character… ");
      aString = keyboard.nextLine();
      aChar = aString.charAt(0);
      System.out.println("The character is " + aChar);
```

Figure 7-3 The TestCharacter application *(continues)*

(continued)

```
      if(Character.isUpperCase(aChar))
         System.out.println(aChar + " is uppercase");
      else
         System.out.println(aChar + " is not uppercase");
      if(Character.isLowerCase(aChar))
         System.out.println(aChar + " is lowercase");
      else
         System.out.println(aChar + " is not lowercase");
      aChar = Character.toLowerCase(aChar);
      System.out.println("After toLowerCase(), aChar is " + aChar);
      aChar = Character.toUpperCase(aChar);
      System.out.println("After toUpperCase(), aChar is " + aChar);
      if(Character.isLetterOrDigit(aChar))
         System.out.println(aChar + " is a letter or digit");
      else
         System.out.println(aChar +
            " is neither a letter nor a digit");
      if(Character.isWhitespace(aChar))
         System.out.println(aChar + " is whitespace");
      else
         System.out.println(aChar + " is not whitespace");
   }
}
```

Figure 7-3 The `TestCharacter` application

You can tell that each of the `Character` class methods used in the `TestCharacter` application in Figure 7-3 is a static method because the method name is used without an object reference—you use only the class name, a dot, and the method name. You learned about the difference between static and instance methods in Chapter 3.

The output of three typical executions of the `TestCharacter` application is shown in Figure 7-4. For example, notice that when the character *C* is tested, you can see the following:

- The value returned by the `isUpperCase()` method is `true`.
- The value returned by the `isLowerCase()` method is `false`.
- The value returned by the `toLowerCase()` method is 'c'.
- The value returned by the `toUpperCase()` method is 'C'.
- The value returned by the `isLetterOrDigit()` method is `true`.
- The value returned by the `isWhitespace()` method is `false`.

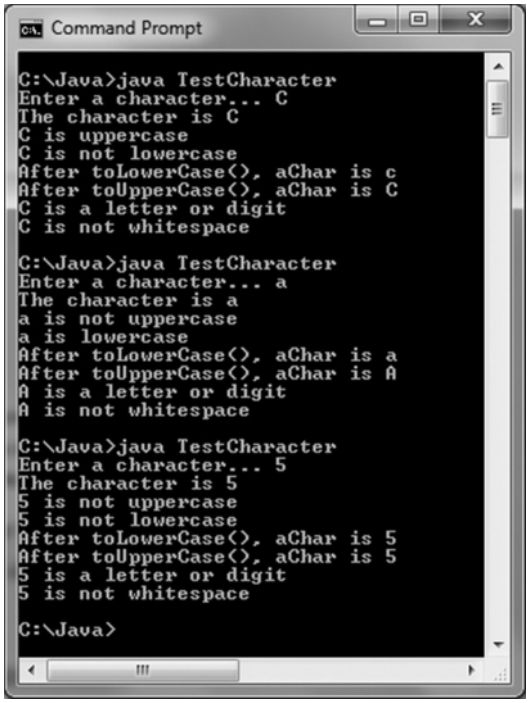

Figure 7-4 Three typical executions of the `TestCharacter` application

TWO TRUTHS & A LIE

Manipulating Characters

1. `Character` is a class, but `char` is a simple data type.
2. The `Character` class method `isLowerCase()` returns the lowercase version of any uppercase character.
3. If a `char` variable holds the Unicode value for the Tab key, `isWhitespace()` would be true and `isLetterOrDigit()` would be false.

The false statement is #2. The `Character` class method `isLowerCase()` returns `true` or `false`, as do all the `Character` class methods whose names use the `is` prefix.

You Do It

Retrieving and Testing a Character

In this section, you write a short program that extracts a character from a `String` and uses its lowercase equivalent.

1. Open a new file in your text editor, and start a program that accepts a response from a user using an input dialog box.

```
import javax.swing.JOptionPane;
public class YLoop
{
   public static void main(String[] args)
   {
```

2. Declare a constant character that holds the value that indicates a user wants to continue. Also include the `String` the user enters, a variable to hold the contents of the first character in that `String`, and a count of the number of iterations performed.

```
final char YES_OPTION = 'y';
String entryString;
char entryChar;
int count = 0;
```

3. Accept a `String` response from the user, and use the `charAt()` method to extract the first character in the `String`.

```
entryString = JOptionPane.showInputDialog(null,
   "Do you want to see a greeting?");
entryChar = entryString.charAt(0);
```

4. Write a loop that is controlled by comparing the lowercase equivalent of the entry `String`'s first character to `'y'`. Within the loop, `count` is incremented and then displayed along with a greeting and a prompt that asks the user whether an additional greeting should be displayed. The first character is extracted from the new `String` the user enters.

```
while(Character.toLowerCase(entryChar) == YES_OPTION)
{
   ++count;
   entryString = JOptionPane.showInputDialog(null,
      "Greeting #" + count +
      " Hello!\nDo you want to see another greeting?");
   entryChar = entryString.charAt(0);
}
```

(continues)

(continued)

5. Add closing curly braces for the `main()` method and for the class.
6. Save the file as **YLoop.java**, and then compile and execute it. Figure 7-5 shows the input dialog box that appears. Whether the user enters *Yes*, *yes*, *yeah*, *Yup*, *Y*, or *y*, the loop will continue. When the user enters anything that does not start with *y* or *Y*, the loop ends. (If the user clicks OK without entering anything in the dialog box, an error message is displayed and the program ends abruptly. You will learn to manage this type of error in the chapter *Exception Handling*.)

Figure 7-5 Dialog box displayed by the YLoop application

Examining the `Character` *Class at the Java Web Site*

1. Using a Web browser, go to the Java Web site at ***www.oracle.com/technetwork/java/index.html***, and select **Java APIs** and **Java SE 7**. Using the alphabetical list of classes, find the **`Character`** class and select it.
2. Examine the extensive list of methods for the `Character` class. Find one with which you are familiar, such as `toLowerCase()`. Notice that there are two overloaded versions of the method. The one you used in the `YLoop` application accepts a `char` and returns a `char`. The other version that accepts and returns an `int` uses Unicode values. Appendix B of this book provides more information on Unicode.

Declaring and Comparing String Objects

You learned in Chapter 1 that a sequence of characters enclosed within double quotation marks is a literal string. (Programmers might also call it a "string literal.") You have used many literal strings, such as "First Java application", and you have assigned values to String objects and used them within methods, such as println() and showMessageDialog(). A literal string is an unnamed object, or **anonymous object**, of the String class, and a **String variable** is simply a named object of the same class. The class String is defined in java.lang.String, which is automatically imported into every program you write.

You have declared a String array named args in every main() method header that you have written. You will learn about arrays in the next chapter.

When you declare a String object, the String itself—that is, the series of characters contained in the String—is distinct from the identifier you use to refer to it. You can create a String object by using the keyword new and the String constructor, just as you would create an object of any other type. For example, the following statement defines an object named aGreeting, declares it to be of type String, and assigns an initial value of "Hello" to the String:

```
String aGreeting = new String("Hello");
```

The variable aGreeting stores a reference to a String object—it keeps track of where the String object is stored in memory. When you declare and initialize aGreeting, it links to the initializing String value. Because Strings are declared so routinely in programs, Java provides a shortcut, so you can declare a String containing "Hello" with the following statement that omits the keyword new and does not explicitly call the class constructor:

```
String aGreeting = "Hello";
```

Comparing String Values

In Java, String is a class, and each created String is an object. A String variable name is a reference; that is, a String variable name refers to a location in memory, rather than to a particular value.

The distinction is subtle, but when you declare a variable of a basic, primitive type, such as int x = 10;, the memory address where x is located holds the value 10. If you later assign a new value to x, the new value replaces the old one at the assigned memory address. For example, if you code x = 45;, then 45 replaces 10 at the address of x.

In contrast, when you declare a String, such as String aGreeting = "Hello";, aGreeting does not hold the characters "Hello"; instead it holds a memory address where the characters are stored.

The left side of Figure 7-6 shows a diagram of computer memory if aGreeting happens to be stored at memory address 10876 and the String "Hello" happens to be stored at memory address 26040. You cannot choose the memory address where a value is stored. Addresses such as 10876 and 26040 are chosen by the operating system.

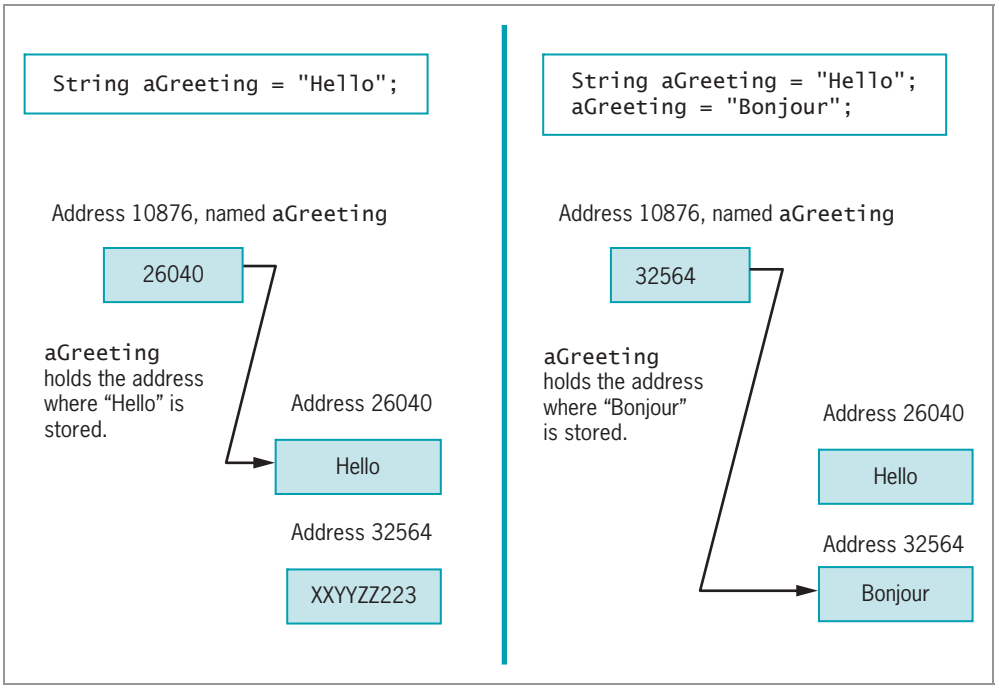

Figure 7-6 Contents of `aGreeting` at declaration and after an assignment

When you refer to `aGreeting`, you actually are accessing the address of the characters you want to use. (In the example on the left side of Figure 7-6, the memory location beginning at address 32564 has not yet been used and holds garbage values.)

If you subsequently assign a new value to `aGreeting`, such as `aGreeting = "Bonjour";`, the address held by `aGreeting` is altered; now, `aGreeting` holds a new address where the characters "Bonjour" are stored. As shown on the right side of Figure 7-6, "Bonjour" is an entirely new object created with its own location. The "Hello" `String` is still in memory, but `aGreeting` no longer holds its address. Eventually, a part of the Java system called the garbage collector discards the "Hello" characters. `String`s, therefore, are never actually changed; instead, new `String`s are created and `String` references hold the new addresses. `String`s and other objects that can't be changed are **immutable**.

The creators of Java made `String`s immutable for several reasons. For example, in environments where multiple programs (or parts of programs, called *threads of execution*) run concurrently, one logical path cannot change a `String` being used by another path. The compiler can also be made to execute more efficiently with immutable `String` objects. In simple programs, you don't care much about these features. However, immutability leads to performance problems. Later in this chapter, you will learn that if you want to use a mutable object to hold strings of characters, you can use the `StringBuilder` class.

Because String references hold memory addresses, making simple comparisons between them often produces misleading results. For example, recall the TryToCompareStrings application in Figure 7-1. In this example, Java evaluates the String variables aName and anotherName as not equal because even though the variables contain the same series of characters, one set is assigned directly and the other is entered from the keyboard and stored in a different area of memory. When you compare Strings with the == operator, you are comparing their memory addresses, not their values. Furthermore, when you try to compare Strings using the less-than (<) or greater-than (>) operator, the program will not even compile.

If you declare two String objects and initialize both to the same value, the value is stored only once in memory and the two object references hold the same memory address. Because the String is stored just once, memory is saved. Consider the following example in which the same value is assigned to two Strings (as opposed to getting one from user input). The reason for the output in the following example is misleading. When you write the following code, the output is *Strings are the same.*

```
String firstString = "abc";
String secondString = "abc";
if(firstString == secondString)
   System.out.println("Strings are the same");
```

The output is *Strings are the same* because the memory addresses held by firstString and secondString are the same, not because their contents are the same.

Fortunately, the String class provides you with a number of useful methods that compare Strings in the way you usually intend. The String class **equals() method** evaluates the contents of two String objects to determine if they are equivalent. The method returns true if the objects have identical contents, no matter how the contents were assigned. For example, Figure 7-7 shows a CompareStrings application, which is identical to the TryToCompareStrings application in Figure 7-1 except for the shaded comparison.

```
import java.util.Scanner;
public class CompareStrings
{
   public static void main(String[] args)
   {
      String aName = "Carmen";
      String anotherName;
      Scanner input = new Scanner(System.in);
      System.out.print("Enter your name > ");
      anotherName = input.nextLine();
      if(aName.equals(anotherName))
         System.out.println(aName + " equals " + anotherName);
      else
         System.out.println(aName + " does not equal " + anotherName);
   }
}
```

Figure 7-7 The CompareStrings application

When a user runs the CompareStrings application and enters "Carmen" for the name, the output appears as shown in Figure 7-8; the contents of the Strings are equal. The String class equals() method returns true only if two Strings are identical in content. Thus, a String that refers to "Carmen " (with a space after the *n*) is not equivalent to a String that refers to "Carmen" (with no space after the *n*).

Figure 7-8 Output of the CompareStrings application

Technically, the equals() method does not perform an alphabetical comparison with Strings; it performs a **lexicographical comparison**—a comparison based on the integer Unicode values of the characters.

Each String declared in Figure 7-7 (aName and anotherName) is an object of type String, so each String has access to the String class equals() method. If you analyze how the equals() method is used in the application in Figure 7-7, you can tell quite a bit about how the method was written by Java's creators:

- Because you use the equals() method with a String object and the method uses the unique contents of that object to make a comparison, you can tell that it is not a static method.
- Because the call to the equals() method can be used in an if statement, you can tell that it returns a Boolean value.
- Because you see a String used between the parentheses in the method call, you can tell that the equals() method takes a String argument.

So, the method header of the equals() method within the String class must be similar to the following:

```
public boolean equals(String s)
```

The only thing you do not know about the method header is the local name used for the String argument—it might be s, or it might be any other legal Java identifier. When you use a prewritten method such as equals(), you do not know how the code looks inside it. For example, you do not know whether the equals() method compares the characters in the Strings from left to right or from right to left. All you know is that the method returns true if the two Strings are completely equivalent and false if they are not.

Because both aName and anotherName are Strings in the application in Figure 7-7, the aName object can call equals() with aName.equals(anotherName) as shown, or the anotherName object could call equals() with anotherName.equals(aName). The equals() method can take either a variable String object or a literal string as its argument.

The String class **equalsIgnoreCase() method** is similar to the equals() method. As its name implies, this method ignores case when determining if two Strings are equivalent. Thus, if you declare a String as String aName = "Carmen";, then aName.equals("caRMen") is false, but aName.equalsIgnoreCase("caRMen") is true. This method is useful when users type responses to prompts in your programs. You cannot predict when a user might use the Shift key or the Caps Lock key during data entry.

When the String class **compareTo() method** is used to compare two Strings, it provides additional information to the user in the form of an integer value. When you use compareTo() to compare two String objects, the method returns zero only if the two Strings refer to the same value. If there is any difference between the Strings, a negative number is returned if the calling object is "less than" the argument, and a positive number is returned if the calling object is "more than" the argument. Strings are considered "less than" or "more than" each other based on their Unicode values; thus, "a" is less than "b", and "b" is less than "c". For example, if aName refers to "Roger", then aName.compareTo("Robert"); returns a 5. The number is positive, indicating that "Roger" is more than "Robert". This does not mean that "Roger" has more characters than "Robert"; it means that "Roger" is alphabetically "more" than "Robert". The comparison proceeds as follows:

- The *R* in "Roger" and the *R* in "Robert" are compared, and found to be equal.
- The *o* in "Roger" and the *o* in "Robert" are compared, and found to be equal.
- The *g* in "Roger" and the *b* in "Robert" are compared; they are different. The numeric value of *g* minus the numeric value of *b* is 5 (because *g* is five letters after *b* in the alphabet), so the compareTo() method returns the value 5.

Often, you won't care what the specific return value of compareTo() is; you simply want to determine if it is positive or negative. For example, you can use a test such as if(aWord.compareTo(anotherWord) < 0) to determine whether aWord is alphabetically less than anotherWord. If aWord is a String variable that refers to the value "hamster", and anotherWord is a String variable that refers to the value "iguana", the comparison if(aWord.compareTo(anotherWord) < 0) yields true.

Empty and null Strings

Programmers are often confused by the difference between empty Strings and null Strings. You can create an empty String named word1 and two null Strings named word2 and word3 with the following statements:

```
String word1 = "";
String word2 = null;
String word3;
```

The empty `String word1` references a memory address where no characters are stored. The **`null String`** `word2` uses the Java keyword `null` so that `word2` does not yet hold a memory address. The unassigned `String word3` is also a `null String` by default. A significant difference between these declarations is that `word1` can be used with the `String` methods described in this chapter, but `word2` and `word3` cannot. For example, assuming a `String` named `someOtherString` has been assigned a value, then the comparison `word1.equals(someOtherString)` is valid, but `word2.equals(someOtherString)` causes an error.

Because `String`s are set to `null` by default, some programmers think explicitly setting a `String` to `null` is redundant. Other programmers feel that explicitly using the keyword `null` makes your intentions clearer to those reading your program. You should use the style your organization recommends.

Watch the video *Comparing Strings*.

TWO TRUTHS & A LIE

Declaring and Comparing `String` Objects

1. To create a `String` object, you must use the keyword `new` and explicitly call the class constructor.
2. When you compare `String`s with the `==` operator, you are comparing their memory addresses, not their values.
3. When you compare `String`s with the `equals()` method, you are comparing their values, not their memory addresses.

The false statement is #1. You can create a `String` object with or without the keyword `new` and without explicitly calling the `String` constructor.

You Do It

Examining the `String` Class at the Java Web Site

In this section, you learn more about the `String` class.

1. Using a Web browser, go to the Java Web site, and select **Java APIs** and **Java SE 7**. Using the alphabetical list of classes, find the **`String`** class and select it.

(continues)

(continued)

2. Examine the `equals()` method. In the last section you saw this method used in expressions such as `aName.equals(anotherName)`. Because `equals()` is used with the object `aName`, you could predict that the `equals()` method is not `static`. When you look at the documentation for the `equals()` method, you can see this is true. You also can see that it returns a `boolean` value. What you might have predicted is that the `equals()` method takes a `String` argument, because `anotherName` is a `String`. However, the documentation shows that the `equals()` method accepts an `Object` argument. You will learn more about the `Object` class in the chapter *Advanced Inheritance Concepts*, but for now understand that a `String` is a type of `Object`. `Object` is a class from which all other classes stem. In Java, every class is a type of `Object`.

Using Other `String` Methods

A wide variety of additional methods are available with the `String` class. The methods **`toUpperCase()`** and **`toLowerCase()`** convert any `String` to its uppercase or lowercase equivalent. For example, if you declare a `String` as `String aWord = "something";`, then the string "something" is created in memory and its address is assigned to `aWord`. The statement `aWord = aWord.toUpperCase()` creates "SOMETHING" in memory and assigns its address to `aWord`. Because `aWord` now refers to "SOMETHING," `aWord = aWord.toLowerCase()` alters `aWord` to refer to "something".

The **`length()` method** is an accessor method that returns the length of a `String`. For example, the following statements result in the variable `len` that holds the value 5.

```
String greeting = "Hello";
int len = greeting.length();
```

In Chapter 2, you learned that your own accessor methods often start with the prefix *get*. The creators of Java did not follow this convention when naming the `length()` method.

When you must determine whether a `String` is empty, it is more efficient to compare its length to 0 than it is to use the `equals()` method.

The **`indexOf()` method** determines whether a specific character occurs within a `String`. If it does, the method returns the position of the character; the first position of a `String` is zero. The return value is –1 if the character does not exist in the `String`. For example, in `String myName = "Stacy";`, the value of `myName.indexOf('S')` is "0", the value of `myName.indexOf('a')` is "2", and the value of `myName.indexOf('q')` is "-1".

The **charAt() method** requires an integer argument that indicates the position of the character that the method returns, starting with 0. For example, if myName is a String that refers to "Stacy", the value of myName.charAt(0) is "S" and the value of myName.charAt(4) is "y". An error occurs if you use an argument that is negative, or greater than or equal to the length of the calling String . Instead of using a constant argument with charAt(), frequently you will want to use a variable argument to examine every character in a loop. For example, to count the number of spaces in the String mySentence, you might write a loop like the following:

```
for(int x = 0; x < myName.length(); ++x)
   if(mySentence.charAt(x) == ' ')
      ++countOfSpaces;
```

The **endsWith() method** and the **startsWith() method** each take a String argument and return true or false if a String object does or does not end or start with the specified argument. For example, if String myName = "Stacy";, then myName.startsWith("Sta") is true, and myName.endsWith("z") is false. These methods are case sensitive, so if String myName = "Stacy";, then myName.startsWith("sta") is false.

The **replace() method** allows you to replace all occurrences of some character within a String. For example, if String yourName = "Annette";, then String goofyName = yourName.replace('n', 'X'); assigns "AXXette" to goofyName. The statement goofyName = yourName.replace('p', 'X'); would assign "Annette" to goofyName without any changes because 'p' is not found in yourName. The replace() method is case sensitive, so if String yourName = "Annette";, then String goofyName = yourName.replace('N', 'X'); results in no change.

Although not part of the String class, the **toString() method** is useful when working with String objects. It converts any object to a String. In particular, it is useful when you want to convert primitive data types to Strings. So, if you declare theString and someInt = 4;, as follows, then after the following statements, theString refers to "4":

```
String theString;
int someInt = 4;
theString = Integer.toString(someInt);
```

If you declare another String and a double as follows, then after the following statements, aString refers to "8.25":

```
String aString;
double someDouble = 8.25;
aString = Double.toString(someDouble);
```

You also can use **concatenation** to convert any primitive type to a String. You can join a simple variable to a String, creating a longer String using the + operator. For example, if you declare a variable as int myAge = 25;, the following statement results in aString that refers to "My age is 25":

```
String aString = "My age is " + myAge;
```

Similarly, if you write the following, then anotherString refers to "12.34".

```
String anotherString;
float someFloat = 12.34f;
anotherString = "" + someFloat;
```

The Java interpreter first converts the float 12.34f to a String "12.34" and adds it to the empty String "".

The toString() method does not originate in the String class; it is a method included in Java that you can use with any type of object. In the chapter *Advanced Inheritance Concepts*, you will learn how to construct versions of the method for your own classes and that toString() originates in the Object class. You have been using toString() throughout this book without knowing it. When you use print() and println(), their arguments are automatically converted to Strings if necessary. You don't need import statements to use toString() because it is part of java.lang, which is imported automatically. Because the toString() method you use with println() takes arguments of any primitive type, including int, char, double, and so on, it is a working example of polymorphism.

You already know that you can concatenate Strings with other Strings or values by using a plus sign (+); you have used this approach in methods such as println() and showMessageDialog() since Chapter 1. For example, you can display a firstName, a space, and a lastName with the following statement:

```
System.out.println(firstName + " " + lastName);
```

In addition, you can extract part of a String with the **substring() method** and use it alone or concatenate it with another String. The substring() method takes two integer arguments—a start position and an end position—that are both based on the fact that a String's first position is position zero. The length of the extracted substring is the difference between the second integer and the first integer; if you call the method without a second integer argument, the substring extends to the end of the original string.

For example, the application in Figure 7-9 prompts the user for a customer's first and last names. The application then extracts these names so that a friendly business letter can be constructed. After the application prompts the user to enter a name, a loop control variable is initialized to 0. While the variable remains less than the length of the entered name, each character is compared to the space character. When a space is found, two new strings are created. The first, firstName, is the substring of the original entry from position 0 to the location where the space was found. The second, familyName, is the substring of the original entry from the position after the space to the end of the string. Once the first and last names have been created, the loop control variable is set to the length of the original string so the loop will exit and proceed to the display of the friendly business letter. Figure 7-10 shows the data entry screen as well as the output letter created.

```
import javax.swing.*;
public class BusinessLetter
{
   public static void main(String[] args)
   {
      String name;
      String firstName = "";
      String familyName = "";
      int x;
      char c;
      name = JOptionPane.showInputDialog(null,
         "Please enter customer's first and last name");
      x = 0;
      while(x < name.length())
      {
         if(name.charAt(x) == ' ')
         {
            firstName = name.substring(0, x);
            familyName = name.substring(x + 1, name.length());
            x = name.length();
         }
         ++x;
      }
      JOptionPane.showMessageDialog(null,
         "Dear " + firstName +
         ",\nI am so glad we are on a first name basis" +
         "\nbecause I would like the opportunity to" +
         "\ntalk to you about an affordable insurance" +
         "\nprotection plan for the entire " + familyName +
         "\nfamily. Call A-One Family Insurance today" +
         "\nat 1-800-555-9287.");
   }
}
```

Figure 7-9 The `BusinessLetter` application

To keep the example simple, the `BusinessLetter` application in Figure 7-9 displays a letter for just one customer. An actual business application would most likely allow a clerk to enter dozens or even hundreds of customer names and store them in a data file for future use. You will learn to store data permanently in files in the chapter *File Input and Output*. For now, just concentrate on the string-handling capabilities of the application.

Figure 7-10 Typical execution of the `BusinessLetter` application

The **`regionMatches()` method** can be used to test whether two `String` regions are the same. One version of the `regionMatches()` method takes four arguments—the position at which to start in the calling `String`, the other `String` being compared, the position to start in the other `String`, and the length of the comparison. For example, suppose that you have declared two `String` objects as follows:

```
String firstString = "abcde";
String secondString = "xxbcdef";
```

Then, the expression `firstString.regionMatches(1, secondString, 2, 4)` is `true` because the four-character substring starting at position 1 in `firstString` is `"bcde"` and the four-character substring starting at position 2 in `secondString` is also `"bcde"`. The expression `firstString.regionMatches(0, secondString, 3, 2)` is `false` because the two-character substring starting at position 0 in `firstString` is `"ab"` and the two-character substring starting at position 3 in `secondString` is `"cd"`.

A second version of the `regionMatches()` method takes an additional `boolean` argument as the first argument. This argument represents whether case should be ignored in deciding whether regions match. For example, suppose that you have declared two `String`s as follows:

```
String thirdString = "123 Maple Drive";
String fourthString = "a maple tree";
```

Then the following expression is `true` because the substring of `thirdString` that starts at position 4 and continues for five characters is `"Maple"`, the substring of `fourthString` that starts at position 2 and continues for five characters is `"maple"`, and the argument that ignores case has been set to `true`:

```
thirdString.regionMatches(true, 4, fourthString, 2, 5)
```

TWO TRUTHS & A LIE

Using Other String Methods

1. Assume that myName is a String defined as "molly". The value of myName.toUpperCase() is "Molly".
2. Assume that myName is a String defined as "molly". The value of myName.length() is "5".
3. Assume that myName is a String defined as "molly". The value of myName.indexOf('M') is –1.

The false statement is #1. If myName is "molly", then myName.toUpperCase() is "MOLLY".

You Do It

Using String *Class Methods*

To demonstrate the use of the String methods, in this section you create an application that asks a user for a name and then "fixes" the name so that the first letter of each new word is uppercase, whether or not the user entered the name that way.

1. Open a new text file in your text editor. Enter the following first few lines of a RepairName program. The program declares several variables, including two strings that will refer to a name: one will be "repaired" with correct capitalization; the other will be saved as the user entered it so it can be displayed in its original form at the end of the program. After declaring the variables, prompt the user for a name:

```
import javax.swing.*;
public class RepairName
{
   public static void main(String[] args)
   {
      String name, saveOriginalName;
      int stringLength;
      int i;
      char c;
      name = JOptionPane.showInputDialog(null,
         "Please enter your first and last name");
```

(continues)

(continued)

2. Store the name entered in the `saveOriginalName` variable. Next, calculate the length of the name the user entered, then begin a loop that will examine every character in the name. The first character of a name is always capitalized, so when the loop control variable `i` is 0, the character in that position in the name string is extracted and converted to its uppercase equivalent. Then the name is replaced with the uppercase character appended to the remainder of the existing name.

```
saveOriginalName = name;
stringLength = name.length();
for(i=0; i < stringLength; i++)
{
   c = name.charAt(i);
   if(i == 0)
   {
      c = Character.toUpperCase(c);
      name = c + name.substring(1, stringLength);
   }
```

3. After the first character in the name is converted, the program looks through the rest of the name, testing for spaces and capitalizing every character that follows a space. When a space is found at position `i`, `i` is increased, the next character is extracted from the name, the character is converted to its uppercase version, and a new name string is created using the old string up to the current position, the newly capitalized letter, and the remainder of the name string. The `if…else` ends and the `for` loop ends.

```
  else
     if(name.charAt(i) == ' ')
     {
        ++i;
        c = name.charAt(i);
        c = Character.toUpperCase(c);
        name = name.substring(0, i) + c +
           name.substring(i + 1, stringLength);
     }
}
```

4. After every character has been examined, display the original and repaired names, and add closing braces for the `main()` method and the class.

```
      JOptionPane.showMessageDialog(null, "Original name was " +
         saveOriginalName + "\nRepaired name is " + name);
   }
}
```

(continues)

(continued)

5. Save the application as **RepairName.java**, and then compile and run the program. Figure 7-11 shows a typical program execution. Make certain you understand how all the `String` methods contribute to the success of this program.

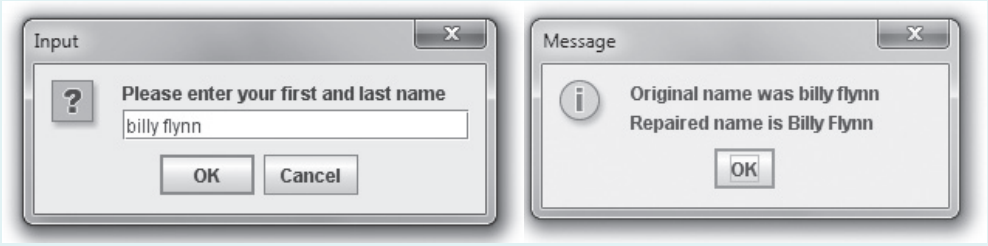

Figure 7-11 Typical execution of the `RepairName` application

Converting `String` Objects to Numbers

If a `String` contains all numbers, as in "649", you can convert it from a `String` to a number so you can use it for arithmetic or you can use it like any other number. For example, suppose you ask a user to enter a salary in an input dialog box. When you accept input using `showInputDialog()`, the accepted value is always a `String`. To be able to use the value in arithmetic statements, you must convert the `String` to a number.

When you use any of the methods described in this section to attempt to convert a `String` to a number, but the `String` does not represent a valid number (for example, if it contains letters), or the `String` represents the wrong kind of number (for example, it contains a decimal point but is being converted to an integer), an error called a `NumberFormatException` occurs. You will learn about exceptions in the chapter *Exception Handling*.

To convert a `String` to an integer, you use the **`Integer` class**, which is part of `java.lang` and is automatically imported into programs you write. The `Integer` class is an example of a wrapper. A **wrapper** is a class or object that is "wrapped around" a simpler element; the `Integer` wrapper class contains a simple integer and useful methods to manipulate it. You have already used the **`parseInt()` method**, which is part of the `Integer` class; it takes a `String` argument and returns its integer value. For example, the following statement stores the numeric value 649 in the variable `anInt`:

```
int anInt = Integer.parseInt("649");
```

You can then use the integer value just as you would any other integer. You can tell that `parseInt()` is a `static` method because you use it with the class name and not with an object.

Alternatively, you can use the `Integer` class `valueOf()` method to convert a `String` to an `Integer` class object, and then use the `Integer` class `intValue()` method to extract the simple integer from its wrapper class. The `ConvertStringToInteger` application in Figure 7-12 shows how you can accomplish the conversion. When the user enters a `String` in the `showInputDialog()` method, the `String` is stored in `stringHours`. The application then uses the `valueOf()` method to convert the `String` to an `Integer` object and uses the `intValue()` method to extract the integer. When the user enters "37" as the `String`, it is converted to a number that can be used in a mathematical statement, and the output appears as expected; this output is shown in Figure 7-13.

```
import javax.swing.JOptionPane;
public class ConvertStringToInteger
{
   public static void main(String[] args)
   {
      String stringHours;
      int hours;
      Integer integerHours;
      final double PAY_RATE = 12.25;
      stringHours = JOptionPane.showInputDialog(null,
        "How many hours did you work this week?");
      integerHours = Integer.valueOf(stringHours);
      hours = integerHours.intValue();
      JOptionPane.showMessageDialog(null, "You worked " +
         hours + " hours at $" + PAY_RATE + " per hour" +
         "\nThat's $" + (hours * PAY_RATE));
   }
}
```

Figure 7-12 The `ConvertStringToInteger` application

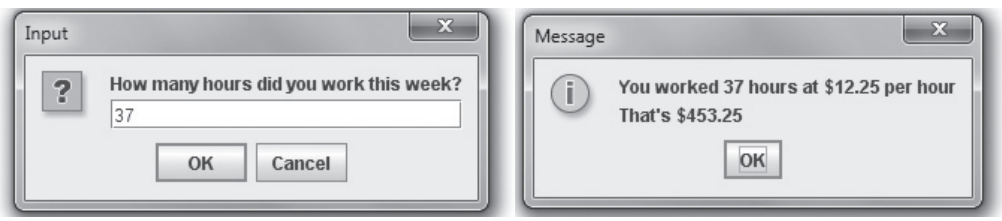

Figure 7-13 Typical execution of the `ConvertStringToInteger` application

It is also easy to convert a `String` object to a `double` value. You must use the **`Double` class**, which, like the `Integer` class, is a wrapper class and is imported into your programs automatically. The `Double` class **`parseDouble()` method** takes a `String` argument and returns its `double` value. For example, the following statement stores the numeric value 147.82 in the variable `doubleValue`.

```
double doubleValue = Double.parseDouble("147.82");
```

To convert a `String` containing "147.82" to a `double`, you also can use the following code:

```
String stringValue = new String("147.82");
Double tempValue = Double.valueOf(stringValue);
double value = tempValue.doubleValue();
```

In this example, `stringValue` is passed to the `Double.valueOf()` method, which returns a `Double` object. The `doubleValue()` method is used with the `tempValue` object; this method returns a simple `double` that is stored in `value`.

The methods `parseInt()` and `parseDouble()` are newer than the `valueOf()` methods, and many programmers prefer to use them when writing new applications.

Besides `Double` and `Integer`, other wrapper classes such as `Float` and `Long` also provide `valueOf()` methods that convert `String`s to the wrapper types. Additionally, the classes provide `parseFloat()` and `parseLong()` methods, respectively.

Watch the video *String Methods*.

TWO TRUTHS & A LIE

Converting `String` Objects to Numbers

1. The `Integer` and `Double` classes are wrapper classes.
2. The value of `Integer.parseInt("22.22")` is 22.
3. The value of `Double.parseDouble("22.22")` is 22.22.

The false statement is #2. `Integer.parseInt("22.22")` does not work because the `String` argument to the `parseInt()` method cannot be converted to an integer.

You Do It

Converting a `String` *to an Integer*

In the next steps, you write a program that prompts the user for a number, reads characters from the keyboard, stores the characters in a `String`, and then converts the `String` to an integer that can be used in arithmetic statements.

1. Open a new text file in your text editor. Type the first few lines of a `NumberInput` class that will accept string input:

```
import javax.swing.*;
public class NumberInput
{
   public static void main(String[] args)
   {
```

2. Declare the following variables for the input `String`, the integer to which it is converted, and the result:

```
String inputString;
int inputNumber;
int result;
```

3. Declare a constant that holds a multiplier factor. This program will multiply the user's input by 10:

```
final int FACTOR = 10;
```

4. Enter the following input dialog box statement that stores the user keyboard input in the `String` variable `inputString`:

```
inputString = JOptionPane.showInputDialog(null,
   "Enter a number");
```

5. Use the following `Integer.parseInt()` method to convert the input `String` to an integer. Then multiply the integer by 10 and display the result:

```
inputNumber = Integer.parseInt(inputString);
result = inputNumber * FACTOR;
JOptionPane.showMessageDialog(null,
      inputNumber + " * " + FACTOR + " = " + result);
```

6. Add the final two closing curly braces for the program, then save the program as **NumberInput.java** and compile and test the program. Figure 7-14 shows a typical execution. Even though the user enters a `String`, it can be used successfully in an arithmetic statement because it was converted using the `parseInt()` method.

(continues)

(continued)

Figure 7-14 Typical execution of the `NumberInput` program

Examining the `parseInt()` *Method at the Java Web Site*

1. Using a Web browser, go to the Java Web site, and select **Java APIs** and **Java SE 7**. Using the alphabetical list of classes, find the `Integer` class and select it.
2. Find the `parseInt()` method that accepts a `String` parameter and examine it. You can see that the method is `static`, which is why you use it with the class name `Integer` and not with an object. You also see that it returns an `int`. You have used this method since the earliest chapters of this book, but now that you understand classes, objects, and methods, you can more easily interpret the Java documentation.

Learning About the `StringBuilder` and `StringBuffer` Classes

In Java, the value of a `String` is fixed after the `String` is created; `String`s are immutable, or unchangeable. When you write `someString = "Hello";` and follow it with `someString = "Goodbye";`, you have neither changed the contents of computer memory at the address represented by `someString` nor eliminated the characters "Hello". Instead, you have stored "Goodbye" at a new computer memory location and stored the new address in the `someString` variable. If you want to modify `someString` from "Goodbye" to "Goodbye Everybody", you cannot add a space and "Everybody" to the `someString` that contains "Goodbye". Instead, you must create an entirely new `String`, "Goodbye Everybody", and assign it to the `someString` address. If you perform many such operations with `String`s, you end up creating many different `String` objects in memory, which takes time and resources.

To circumvent these limitations, you can use either the **`StringBuilder`** or **`StringBuffer`** class. You use one of these classes, which are alternatives to the `String` class, when you know a `String` will be modified; usually, you can use a `StringBuilder` or `StringBuffer` object

anywhere you would use a String. Like the String class, these two classes are part of the java.lang package and are automatically imported into every program. The classes are identical except for the following:

- StringBuilder is more efficient.
- StringBuffer is thread safe. This means you should use it in applications that run multiple **threads of execution**, which are units of processing that are scheduled by an operating system and that can be used to create multiple paths of control during program execution. Because most programs you write (and all the programs you will write using this book) contain a single thread, usually you should use StringBuilder.

The rest of this section discusses StringBuilder, but every statement is also true of StringBuffer.

You can create a StringBuilder object that contains a String with a statement such as the following:

```
StringBuilder message = new StringBuilder("Hello there");
```

When you use the nextLine() method with a Scanner object for console input or a JOptionPane.showInputDialog() method for GUI input, user input almost always comes into your program as a String. If you want to work with the input as a StringBuilder object, you can convert the String using the StringBuilder constructor. For example, the following statement gets a user's input using a Scanner object named keyboard and then stores it in the StringBuilder name:

```
StringBuilder name = new StringBuilder(keyboard.nextLine());
```

When you create a String, you have the option of omitting the keyword new, but when you initialize a StringBuilder object you must use the keyword new, the constructor name, and an initializing value between the constructor's parentheses. You can create an empty StringBuilder variable using a statement such as the following:

```
StringBuilder uninitializedString = null;
```

The variable does not refer to anything until you initialize it with a defined StringBuilder object. Generally, when you create a String object, sufficient memory is allocated to accommodate the number of Unicode characters in the string. A StringBuilder object, however, contains a memory block called a **buffer**, which might or might not contain a string. Even if it does contain a string, the string might not occupy the entire buffer. In other words, the length of a string can be different from the length of the buffer. The actual length of the buffer is the **capacity** of the StringBuilder object.

You can change the length of a string in a StringBuilder object with the **setLength() method**. The length of a StringBuilder object equals the number of characters in the String contained in the StringBuilder. When you increase a StringBuilder object's length to be longer than the String it holds, the extra characters contain '\u0000'. If you use the setLength() method to specify a length shorter than its String, the string is truncated.

To find the capacity of a `StringBuilder` object, you use the **`capacity()` method**. The `StringBuilderDemo` application in Figure 7-15 demonstrates this method. The application creates a `nameString` object containing the seven characters "Barbara". The capacity of the `StringBuilder` object is obtained and stored in an integer variable named `nameStringCapacity` and displayed.

```
import javax.swing.JOptionPane;
public class StringBuilderDemo
{
   public static void main(String[] args)
   {
      StringBuilder nameString = new StringBuilder("Barbara");
      int nameStringCapacity = nameString.capacity();
      System.out.println("Capacity of nameString is " +
         nameStringCapacity);
      StringBuilder addressString = null;
      addressString = new
         StringBuilder("6311 Hickory Nut Grove Road");
      int addStringCapacity = addressString.capacity();
      System.out.println("Capacity of addressString is " +
         addStringCapacity);
      nameString.setLength(20);
      System.out.println("The name is " + nameString + "end");
      addressString.setLength(20);
      System.out.println("The address is " + addressString);
   }
}
```

Figure 7-15 The `StringBuilderDemo` application

Figure 7-16 shows the `StringBuilder` capacity is 23, which is 16 characters more than the length of the string "Barbara". Whenever you create a `StringBuilder` object, its capacity is the length of the `String` contained in `StringBuilder`, plus 16. The "extra" 16 positions allow for reasonable modification of the `StringBuilder` object after creation without allocating any new memory locations.

Figure 7-16 Output of the StringBuilderDemo application

The creators of Java chose 16 characters as the "extra" length for a StringBuilder object because 16 characters fully occupy four bytes of memory. As you work more with computers in general and programming in particular, you will notice that storage capacities are almost always created in exponential values of 2—for example, 4, 8, 16, 32, 64, and so on.

In the application in Figure 7-15, the addressString variable is created as StringBuilder addressString = null;. The variable does not refer to anything until it is initialized with the defined StringBuilder object in the following statement:

```
addressString = new StringBuilder("6311 Hickory Nut Grove Road");
```

The capacity of this new StringBuilder object is shown in Figure 7-16 as the length of the string plus 16, or 43.

In the application shown in Figure 7-15, the length of each of the Strings is changed to 20 using the setLength() method. The application displays the expanded nameString and "end", so you can see in the output that there are 13 extra spaces at the end of the String. The application also displays the truncated addressString so that you can see the effect of reducing its length to 20.

Using StringBuilder objects provides improved computer performance over String objects because you can insert or append new contents into a StringBuilder. In other words, unlike immutable Strings, the ability of StringBuilder objects to be modified makes them more efficient when you know string contents will change.

Although the equals() method compares String object contents, when you use it with StringBuilder objects, it compares references. You can compare the contents of two StringBuilder objects named obj1 and obj2 by converting them to Strings with an expression such as the following:

```
obj1.toString().equals(obj2.toString())
```

The StringBuilder class provides you with four constructors as follows:

- public StringBuilder() constructs a StringBuilder with no characters and a default size of 16 characters.
- public StringBuilder(int capacity) constructs a StringBuilder with no characters and a capacity specified by the parameter.

- `public StringBuilder(String s)` contains the same characters as those stored in the `String` object `s`. The capacity of the `StringBuilder` is the length of the `String` argument you provide, plus 16 additional characters.
- The fourth `StringBuilder` constructor uses an argument of type `CharSequence`. `CharSequence` is another Java class; it is an interface that holds a sequence of `char` values. You will learn to create interfaces in the chapter *Advanced Inheritance Concepts*.

The **`append()` method** lets you add characters to the end of a `StringBuilder` object. For example, the following two statements together declare `phrase` to hold "Happy" and alter the phrase to hold "Happy birthday":

```
StringBuilder phrase = new StringBuilder("Happy");
phrase.append(" birthday");
```

The **`insert()` method** lets you add characters at a specific location within a `StringBuilder` object. For example, if `phrase` refers to "Happy birthday", then `phrase.insert(6, "30th ");` alters the `StringBuilder` to contain "Happy 30th birthday". The first character in the `StringBuilder` object occupies position zero.

To alter just one character in a `StringBuilder`, you can use the **`setCharAt()` method**, which allows you to change a character at a specified position within a `StringBuilder` object. This method requires two arguments: an integer position and a character. If `phrase` refers to "Happy 30th birthday", then `phrase.setCharAt(6,'4');` changes the value into a 40th birthday greeting.

One way you can extract a character from a `StringBuilder` object is to use the `charAt()` method. The **`charAt()` method** accepts an argument that is the offset of the character position from the beginning of a `String` and returns the character at that position. The following statements assign the character 'P' to the variable `letter`:

```
StringBuilder text = new StringBuilder("Java Programming");
char letter = text.charAt(5);
```

If you try to use an index that is less than 0 or greater than the index of the last position in the `StringBuilder` object, you cause an error known as an exception and your program terminates.

When you can approximate the eventual size needed for a `StringBuilder` object, assigning sufficient capacity can improve program performance. For example, the program in Figure 7-17 compares the time needed to append "Java" 20,000 times to two `StringBuilder` objects—one that has the initial default size of 16 characters and another that has an initial size of 80,000 characters. Figure 7-18 shows the execution. The extra time needed for the loop that uses the shorter `StringBuilder` is the result of repeatedly assigning new memory as the object grows in size.

```
public class CompareConcatenationTimes
{
   public static void main(String[] args)
   {
      long startTime1, startTime2,
         endTime1, endTime2;
      final int TIMES = 20000;
      int x;
      StringBuilder string1 = new StringBuilder("");
      StringBuilder string2 = new StringBuilder(TIMES * 4);
      startTime1 = System.currentTimeMillis();
      for(x = 0; x < TIMES; ++x)
         string1.append("Java");
      endTime1 = System.currentTimeMillis();
      System.out.println("Time for empty StringBuilder : "
         + (endTime1 - startTime1)+ " milliseconds");
      startTime2 = System.currentTimeMillis();
      for(x = 0; x < TIMES; ++x)
         string2.append("Java");
      endTime2 = System.currentTimeMillis();
      System.out.println("Time for large StringBuilder : "
         + (endTime2 - startTime2)+ " milliseconds");
   }
}
```

Figure 7-17 The CompareConcatenationTimes application

Figure 7-18 Output of the CompareConcatenationTimes program

You saw a demonstration of the currentTimeMillis() method in Chapter 6.

Watch the video *StringBuilder*.

TWO TRUTHS & A LIE

Learning About the StringBuilder and StringBuffer Classes

1. When you create a `String`, you have the option of omitting the keyword `new`, but when you initialize a `StringBuilder` object, you must use the keyword `new`, the constructor name, and an initializing value between the constructor's parentheses.
2. When you create a `StringBuilder` object with an initial value of "Juan", its capacity is 16.
3. If a `StringBuilder` named `myAddress` contains "817", then `myAddress.append(" Maple Lane");` alters `myAddress` to contain "817 Maple Lane".

The false statement is #2. When you create a `StringBuilder` object with an initial value of "Juan", its capacity is the length of the `String` contained in `StringBuilder`, 4, plus 16 more, for a total of 20.

You Do It

Using `StringBuilder` *Methods*

In these steps, you write a program that demonstrates the `StringBuilder` class.

1. Open a new text file, and type the following first lines of a `DemoStringBuilder` class:

```
public class DemoStringBuilder
{
   public static void main(String[] args)
   {
```

2. Use the following code to create a `StringBuilder` object, and then call a `print()` method (that you will create in Step 7) to display the `StringBuilder`:

(continues)

(continued)

```
StringBuilder str = new StringBuilder("singing");
print(str);
```

3. Enter the following `append()` method to add characters to the existing `StringBuilder` and display it again:

```
str.append(" in the dead of ");
print(str);
```

4. Enter the following `insert()` method to insert characters. Then display the `StringBuilder`, insert additional characters, and display it again:

```
str.insert(0, "Black");
print(str);
str.insert(5, "bird ");
print(str);
```

5. Add one more `append()` and `print()` combination:

```
str.append("night");
print(str);
```

6. Add a closing curly brace for the `main()` method.

7. Enter the following `print()` method that displays `StringBuilder` objects:

```
public static void print(StringBuilder s)
{
   System.out.println(s);
}
```

8. Type the closing curly brace for the class, and then save the file as **DemoStringBuilder.java**. Compile and execute, and then compare your output to Figure 7-19.

Figure 7-19 Output of the `DemoStringBuilder` application

Don't Do It

- Don't attempt to compare `String` objects using the standard comparison operators. The == operator will compare only the addresses of `String`s, and the < and > operators will not work.
- Don't forget that `startsWith()`, `endsWith()`, and `replace()` are case sensitive, so you might want to convert participating `String`s to the same case before using them.
- Don't forget to use the `new` operator and the constructor when declaring initialized `StringBuilder` or `StringBuffer` objects.

Key Terms

A **reference** is a variable that holds a memory address.

The **`Character` class** is one whose instances can hold a single character value. This class also defines methods that can manipulate or inspect single-character data.

The **`String` class** is for working with fixed-string data—that is, unchanging data composed of multiple characters.

A **`String` variable** is a named object of the `String` class.

An **anonymous object** is an unnamed object.

Immutable objects cannot be changed.

The `String` class **`equals()` method** evaluates the contents of two `String` objects to determine if they are equivalent.

A **lexicographical comparison** is based on the integer Unicode values of characters.

The `String` class **`equalsIgnoreCase()` method** is similar to the `equals()` method. As its name implies, it ignores case when determining if two `String`s are equivalent.

The `String` class **`compareTo()` method** is used to compare two `String`s; the method returns zero only if the two `String`s refer to the same value. If there is any difference between the `String`s, a negative number is returned if the calling object is "less than" the argument, and a positive number is returned if the calling object is "more than" the argument.

A **`null String`** does not hold a memory address.

The `String` class **`toUpperCase()` method** converts any `String` to its uppercase equivalent.

The `String` class **`toLowerCase()` method** converts any `String` to its lowercase equivalent.

The `String` class **`length()` method** returns the length of a `String`.

The `String` class **`indexOf()` method** determines whether a specific character occurs within a `String`. If it does, the method returns the position of the character; the first position of a `String` begins with zero. The return value is –1 if the character does not exist in the `String`.

The `String` class **`charAt()` method** requires an integer argument that indicates the position of the character that the method returns.

The `String` class **`endsWith()` method** takes a `String` argument and returns `true` or `false` if a `String` object does or does not end with the specified argument.

The `String` class **`startsWith()` method** takes a `String` argument and returns `true` or `false` if a `String` object does or does not start with the specified argument.

The `String` class **`replace()` method** replaces all occurrences of some character within a `String`.

The **`toString()` method** converts any object to a `String`.

Concatenation is the process of joining a variable to a string to create a longer string.

The **`substring()` method** extracts part of a `String`.

The **`regionMatches()` method** tests whether two `String` regions are the same.

The **`Integer` class** is a wrapper class that contains a simple integer and useful methods to manipulate it.

A **wrapper** is a class or object that is "wrapped around" a simpler element.

The `Integer` class **`parseInt()` method** takes a `String` argument and returns its integer value.

The **`Double` class** is a wrapper class that contains a simple `double` and useful methods to manipulate it.

The `Double` class **`parseDouble()` method** takes a `String` argument and returns its `double` value.

The **`StringBuilder`** class is used as an alternative to the `String` class because it is more efficient if a `String`'s contents will change.

The **`StringBuffer`** class is an alternative to the `String` and `StringBuilder` classes because it is efficient and thread safe.

Threads of execution are units of processing that are scheduled by an operating system and that can be used to create multiple paths of control during program execution.

A **buffer** is a block of memory.

The **capacity** of a `StringBuilder` object is the actual length of the buffer, as opposed to that of the string contained in the buffer.

The StringBuilder class **setLength() method** changes the length of the string in a StringBuilder object.

The StringBuilder class **capacity() method** returns the actual length, or capacity, of the StringBuilder object.

The StringBuilder class **append() method** lets you add characters to the end of a StringBuilder object.

The StringBuilder class **insert() method** lets you add characters at a specific location within a StringBuilder object.

The StringBuilder class **setCharAt() method** allows you to change a character at a specified position within a StringBuilder object.

The StringBuilder class **charAt() method** accepts an argument that is the offset of the character position from the beginning of a String and returns the character at that position.

Chapter Summary

- String variables are references, so they require special techniques for making comparisons.
- The Character class is one whose instances can hold a single character value. This class also defines methods that can manipulate or inspect single-character data.
- A sequence of characters enclosed within double quotation marks is a literal string. You can create a String object by using the keyword new and the String constructor. Unlike other classes, you also can create a String object without using the keyword new or explicitly calling the class constructor. Strings are immutable. Useful String class methods include equals(), equalsIgnoreCase(), and compareTo().
- Additional useful String methods include toUpperCase(), toLowerCase(), length(), indexOf(), charAt(), endsWith(), startsWith(), and replace(). The toString() method converts any object to a String. You can join Strings with other Strings or values by using a plus sign (+); this process is called concatenation. You can extract part of a String with the substring() method.
- If a String contains appropriate characters, you can convert it to a number with the help of the following methods: Integer.parseInt(), Integer.valueOf(), intValue(), Double.parseDouble(), Double.valueOf(), and doubleValue().
- You can use the StringBuilder or StringBuffer class to improve performance when a string's contents must change.

Review Questions

1. A sequence of characters enclosed within double quotation marks is a _____.

   a. symbolic string
   b. literal string
   c. prompt
   d. command

2. To create a `String` object, you can use the keyword _____ before the constructor call, but you are not required to use this format.

   a. `object`
   b. `create`
   c. `char`
   d. `new`

3. A `String` variable name is a _____.

   a. reference
   b. value
   c. constant
   d. literal

4. The term that programmers use to describe objects that cannot be changed is _____.

   a. irrevocable
   b. nonvolatile
   c. immutable
   d. stable

5. Suppose that you declare two `String` objects as:

```
String word1 = new String("happy");
String word2;
```

   When you ask a user to enter a value for `word2`, if the user types "happy", the value of `word1 == word2` is _____.

   a. `true`
   b. `false`
   c. illegal
   d. unknown

6. If you declare two `String` objects as:

```
String word1 = new String("happy");
String word2 = new String("happy");
```

   the value of `word1.equals(word2)` is _____.

   a. `true`
   b. `false`
   c. illegal
   d. unknown

7. The method that determines whether two `String` objects are equivalent, regardless of case, is _____.

   a. `equalsNoCase()`
   b. `toUpperCase()`
   c. `equalsIgnoreCase()`
   d. `equals()`

8. If a `String` is declared as:

```
String aStr = new String("lima bean");
```

then `aStr.equals("Lima Bean")` is _____.

a. `true`
b. `false`
c. illegal
d. unknown

9. If you create two `String` objects:

```
String name1 = new String("Jordan");
String name2 = new String("Jore");
```

then `name1.compareTo(name2)` has a value of _____.

a. `true`
b. `false`
c. –1
d. 1

10. If `String myFriend = new String("Ginny");`, which of the following has the value 1?

a. `myFriend.compareTo("Gabby");`
b. `myFriend.compareTo("Gabriella");`
c. `myFriend.compareTo("Ghazala");`
d. `myFriend.compareTo("Hammie");`

11. If `String movie = new String("West Side Story");`, the value of `movie.indexOf('s')` is _____.

a. `true`
b. `false`
c. 2
d. 3

12. The `String` class `replace()` method replaces _____.

a. a `String` with a character
b. one `String` with another `String`
c. one character in a `String` with another character
d. every occurrence of a character in a `String` with another character

13. The `toString()` method converts a(n) _____ to a `String`.

a. `char`
b. `int`
c. `float`
d. all of the above

14. Joining `String`s with a plus sign is called _____.

a. chaining
b. concatenation
c. parsing
d. linking

15. The first position in a `String` _____.
    a. must be alphabetic
    b. must be uppercase
    c. is position zero
    d. is ignored by the `compareTo()` method

16. The method that extracts a string from within another string is _____.
    a. `extract()`
    b. `parseString()`
    c. `substring()`
    d. `append()`

17. The method `parseInt()` converts a(n) _____.
    a. integer to a `String`
    b. integer to a `Double`
    c. `Double` to a `String`
    d. `String` to an integer

18. The difference between `int` and `Integer` is _____.
    a. `int` is a primitive type; `Integer` is a class
    b. `int` is a class; `Integer` is a primitive type
    c. nonexistent; both are primitive types
    d. nonexistent; both are classes

19. For an alternative to the `String` class, and so that you can change a `String`'s contents, you can use _____.
    a. `char`
    b. `StringHolder`
    c. `StringBuilder`
    d. `StringMerger`

20. Unlike when you create a `String`, when you create a `StringBuilder`, you must use the keyword _____.
    a. `buffer`
    b. `new`
    c. `null`
    d. `class`

Exercises

Programming Exercises

1. Write an application that prompts the user for three first names and concatenates them in every possible two-name combination so that new parents can easily compare them to find the most pleasing baby name. Save the file as **BabyNameComparison.java**.

2. a. Write an application that counts the total number of spaces contained in the `String` "The person who says something is impossible should not interrupt the person who is doing it." Save the file as **CountSpaces.java**.

   b. Write an application that counts the total number of spaces contained in a `String` entered by the user. Save the file as **CountSpaces2.java**.

3. Write an application that prompts the user for a `String` that contains at least five letters and at least five digits. Continuously reprompt the user until a valid `String` is entered. Display a message indicating whether the user was successful or did not enter enough digits, letters, or both. Save the file as **FiveLettersAndFiveDigits.java**.

4. Write an application that allows a user to enter two `String`s. Output the number of characters in the first `String` that also appear in the second `String`, and output those characters. Figure 7-20 shows two typical executions. Save the file as **CountMatches.java**.

Figure 7-20 Two typical executions of the `CountMatches` application

5. Write an application that counts the words in a `String` entered by a user. Words are separated by any combination of spaces, periods, commas, semicolons, question marks, exclamation points, or dashes. Figure 7-21 shows two typical executions. Save the file as **CountWords.java**.

Figure 7-21 Two typical executions of the `CountWords` application

6. a. Write an application that accepts three `String`s from the user and displays one of two messages depending on whether the user entered the `String`s in alphabetical order without regard to case. Save the file as **Alphabetize.java**.

   b. Write an application that accepts three `String`s from the user and displays them in alphabetical order without regard to case. Save the file as **Alphabetize2.java**.

7. Write an application that demonstrates each of the following methods based on the following quote:

   ```
   "You can never cross the ocean until you have the courage to lose sight
   of the shore." – Christopher Columbus
   ```

   - `indexOf('v')`
   - `indexOf('x')`
   - `charAt(16)`
   - `endsWith("bus")`
   - `endsWith("car")`
   - `replace('c', 'C')`

   Save the file as **DemonstratingStringMethods.java**.

8. Three-letter acronyms are common in the business world. For example, in Java you use the IDE (Integrated Development Environment) in the JDK (Java Development Kit) to write programs used by the JVM (Java Virtual Machine) that you might send over a LAN (local area network). Programmers even use the acronym TLA to stand for *three-letter acronym*. Write a program that allows a user to enter three words, and display the appropriate three-letter acronym in all uppercase letters. If the user enters more than three words, ignore the extra words. Figure 7-22 shows a typical execution. Save the file as **ThreeLetterAcronym.java**.

Figure 7-22 Typical execution of the `ThreeLetterAcronym` program

9. Prompt a user to enter a series of integers separated by spaces and accept the input as a `String`. Display the list of integers and their sum. Save the file as **SumIntegersInString.java**.

10. Write an application that determines whether a phrase entered by the user is a palindrome. A palindrome is a phrase that reads the same backward and forward without regarding capitalization or punctuation. For example, "Dot saw I was Tod", "Was it a car or a cat I saw?", and "Madam, I'm Adam" are palindromes. Save the file as **Palindrome.java**.

11. Write an application that prompts a user for a full name and street address and constructs an ID from the user's initials and numeric part of the address. For example, the user William Henry Harrison who lives at 34 Elm would have an ID of WHH34, whereas user Addison Mitchell who lives at 1778 Monroe would have an ID of AM1778. Save the file as **ConstructID.java**.

12. Write an application that accepts a user's password from the keyboard. When the entered password has fewer than six characters, more than 10 characters, or does not contain at least one letter and one digit, prompt the user again. When the user's entry meets all the password requirements, prompt the user to reenter the password, and do not let the user continue until the second password matches the first one. Save the file as **Password.java**.

13. Create a `TaxReturn` class with fields that hold a taxpayer's Social Security number, last name, first name, street address, city, state, zip code, annual income, marital status, and tax liability. Include a constructor that requires arguments that provide values for all the fields other than the tax liability. The constructor calculates the tax liability based on annual income and the percentages in the following table.

	Marital status	
Income ($)	**Single**	**Married**
0–20,000	15%	14%
20,001–50,000	22%	20%
50,001 and over	30%	28%

In the `TaxReturn` class, also include a display method that displays all the `TaxReturn` data. Save the file as **TaxReturn.java**.

Create an application that prompts a user for the data needed to create a `TaxReturn`. Continue to prompt the user for data as long as any of the following are true:

- The Social Security number is not in the correct format, with digits and dashes in the appropriate positions; for example, 999-99-9999.
- The zip code is not five digits.
- The marital status does not begin with one of the following: "S", "s", "M", or "m".
- The annual income is negative.

After all the input data is correct, create a `TaxReturn` object and then display its values. Save the file as **PrepareTax.java**.

Debugging Exercises

1. Each of the following files in the Chapter07 folder of your downloadable student files has syntax and/or logic errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with *Fix*. For example, DebugSeven1.java will become FixDebugSeven1.java.

   a. DebugSeven1.java
   b. DebugSeven2.java
   c. DebugSeven3.java
   d. DebugSeven4.java

Game Zone

1. a. In Chapter 3, you designed a `Card` class. The class holds fields that contain a `Card`'s value and suit. Currently, the suit is represented by a single character (s, h, d, or c). Modify the class so that the suit is a string ("Spades", "Hearts", "Diamonds", or "Clubs"). Also, add a new field to the class to hold the string representation of a `Card`'s rank based on its value. Within the `Card` class `setValue()` method, besides setting the numeric value, also set the string rank value as follows.

Numeric value	String value for rank
1	"Ace"
2 through 10	"2" through "10"
11	"Jack"
12	"Queen"
13	"King"

b. In Chapter 5, you created a War `Card` game that randomly selects two cards (one for the player and one for the computer) and declares a winner (or a tie). Modify the game to set each `Card`'s suit as the appropriate string, then execute the game using the newly modified `Card` class. Figure 7-23 shows four typical executions. Recall that in this version of War, you assume that the Ace is the lowest-valued card. Save the game as **War2.java**.

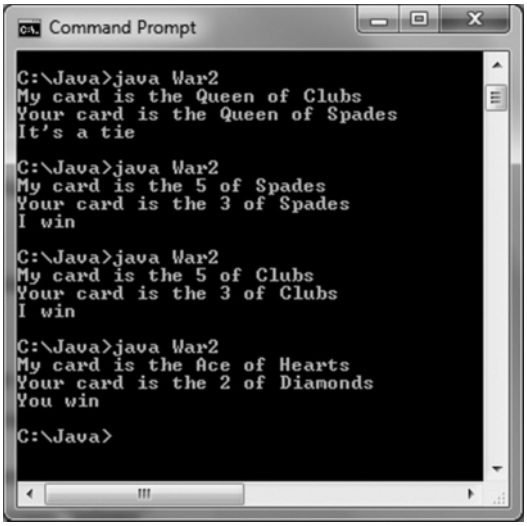

Figure 7-23 Four typical executions of the `War2` game

2. In Chapter 5, you created a Rock Paper Scissors game. In the game, a player entered a number to represent one of the three choices. Make the following improvements to the game:
   - Allow the user to enter a string ("rock", "paper", or "scissors") instead of a digit.
   - Make sure the game works correctly whether the player enters a choice in uppercase or lowercase letters or a combination of the two.

- To allow for player misspellings, accept the player's entry as long as the first two letters are correct. (In other words, if a player types "scixxrs", you will accept it as "scissors" because at least the first two letters are correct.)
- When the player does not type at least the first two letters of the choice correctly, reprompt the player and continue to do so until the player's entry contains at least the first two letters of one of the options.
- Allow 10 complete rounds of the game. At the end, display counts of the number of times the player won, the number of times the computer won, and the number of tie games.

Save the file as **RockPaperScissors2.java**.

3. Create a simple guessing game, similar to Hangman, in which the user guesses letters and then attempts to guess a partially hidden phrase. Display a phrase in which some of the letters are replaced by asterisks; for example, "G* T***" (for "Go Team"). Each time the user guesses a letter, either place the letter in the correct spot (or spots) in the phrase and display it again, or tell the user the guessed letter is not in the phrase. Display a congratulatory message when the entire correct phrase has been deduced. Save the game as **SecretPhrase.java**. In the next chapter, you will modify this program so that instead of presenting the user with the same phrase every time the game is played, the program randomly selects the phrase from a list of phrases.
4. Eliza is a famous 1966 computer program written by Joseph Weizenbaum. It imitates a psychologist (more specifically, a Rogerian therapist) by rephrasing many of a patient's statements as questions and posing them to the patient. This type of therapy (sometimes called nondirectional) is often parodied in movies and television shows, in which the therapist does not even have to listen to the patient, but gives "canned" responses that lead the patient from statement to statement. For example, when the patient says, "I am having trouble with my brother," the therapist might say, "Tell me more about your brother." If the patient says, "I dislike school," the therapist might say, "Why do you say you dislike school?" Eliza became a milestone in the history of computers because it was the first time a computer programmer attempted to create the illusion of human-to-human interaction.

   Create a simple version of Eliza by allowing the user to enter statements continually until the user quits by typing "Goodbye". After each statement, have the computer make one of the following responses:

   - If the user entered the word "my" (for example, "I am having trouble with my brother"), respond with "Tell me more about your" and insert the noun in question—for example, "Tell me more about your brother". When you search for a word in the user's entry, make sure it is the entire word and not just letters within another word. For example, when searching for *my*, make sure it is not part of another word such as *dummy* or *mystic*.
   - If the user entered a strong word, such as "love" or "hate", respond with, "You seem to have strong feelings about that".

- Add a few other appropriate responses of your choosing.
- In the absence of any of the preceding inputs, respond with a random phrase from the following: "Please go on", "Tell me more", or "Continue".

Save the file as **Eliza.java**.

Case Problems

1. Carly's Catering provides meals for parties and special events. In previous chapters, you have developed a class that holds catering event information and an application that tests the methods using four objects of the class. Now modify the `Event` and `EventDemo` classes as follows:

   - Modify the method that sets the event number in the `Event` class so that if the argument passed to the method is not a four-character `String` that starts with a letter followed by three digits, then the event number is forced to "A000". If the initial letter in the event number is not uppercase, force it to be so.
   - Add a contact phone number field to the `Event` class.
   - Add a set method for the contact phone number field in the `Event` class. Whether the user enters all digits or any combination of digits, spaces, dashes, dots, or parentheses for a phone number, store it as all digits. For example, if the user enters *(920) 872-9182*, store the phone number as *9208729182*. If the user enters a number with fewer or more than 10 digits, store the number as *0000000000*.
   - Add a get method for the phone number field. The get method returns the phone number as a `String` constructed as follows: parentheses surround a three-digit area code, followed by a space, followed by the three-digit phone exchange, followed by a hyphen, followed by the last four digits of the phone number.
   - Modify the `EventDemo` program so that besides the event number and guests, the program also prompts the user for and retrieves a contact phone number for each of the sample objects. Display the phone number along with the other `Event` details. Test the `EventDemo` application to make sure it works correctly with valid and invalid event and phone numbers.

   Save the files as **Event.java** and **EventDemo.java**.

2. Sammy's Seashore Supplies rents beach equipment to tourists. In previous chapters, you have developed a class that holds equipment rental information and an application that tests the methods using four objects of the class. Now modify the `Rental` and `RentalDemo` classes as follows:

   - Modify the method that sets the contract number in the `Rental` class so that if the argument passed to the method is not a four-character `String` that starts

with a letter followed by three digits, then the contract number is forced to "A000". If the initial letter in the contract number is not uppercase, force it to be so.

- Add a contact phone number field to the `Rental` class.
- Add a set method for the contact phone number field in the `Rental` class. Whether the user enters all digits or any combination of digits, spaces, dashes, dots, or parentheses for a phone number, store it as all digits. For example, if the user enters *(920) 872-9182*, store the phone number as *9208729182*. If the user enters a number with fewer or more than 10 digits, store the number as *0000000000*.
- Add a get method for the phone number field. The get method returns the phone number as a `String` constructed as follows: parentheses surround a three-digit area code, followed by a space, followed by the three-digit phone exchange, followed by a hyphen, followed by the last four digits of the phone number.
- Modify the `RentalDemo` program so that besides the contract number and minutes, the program also prompts the user for and retrieves a contact phone number for each of the sample objects. Display the phone number along with the other `Rental` details. Test the `RentalDemo` application to make sure it works correctly with valid and invalid contract and phone numbers.

Save the files as **Rental.java** and **RentalDemo.java**.