CHAPTER

7

CHARACTERS, STRINGS, AND THE STRINGBUFFER

In this section, you will:

- ♦ Manipulate characters
- ♦ Declare a String object
- ♦ Compare String values
- ♦ Use other String methods
- ♦ Convert Strings to numbers
- ♦ Learn about the StringBuffer class

I can write an interactive program that accepts a character, but I really want to be able to manipulate characters and let users enter words or numbers into programs," you announce to Lynn Greenbrier, your Java mentor.

"You need to learn about the String," Lynn says. "The wide variety of String methods provided with the Java programming language will help you use words and phrases efficiently. You'll even be able to let your users input numbers."

Previewing a Guessing Game Program

To demonstrate the use of the String methods, you will test a simple guessing game, similar to Hangman. The user first will guess letters, and then guess the motto of Event Handlers Incorporated.

To preview the guessing game:

1. Open the **Chap7SecretPhrase.java** file in the Chapter.07 folder on your Student disk.
2. At the command prompt, compile the class with the command **`javac Chap7SecretPhrase.java`**.
3. Run the program with the command **`java Chap7SecretPhrase`**. Enter keyboard characters one at a time and guess Event Handlers' motto.

Manipulating Characters

You learned in Chapter 2 that the char data type is used to hold any single character. 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, such as letters or digits.

```
Java.lang.Object
  |
  +--java.lang.Character
```

Figure 7-1 Structure of the Character class

Table 7-1 Common methods of the Character class

Method	Description
isUpperCase()	Tests if character is uppercase
toUpperCase()	Changes a lowercase character to uppercase
isLowerCase()	Tests if character is lowercase
toLowerCase()	Changes an uppercase character to lowercase
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.)

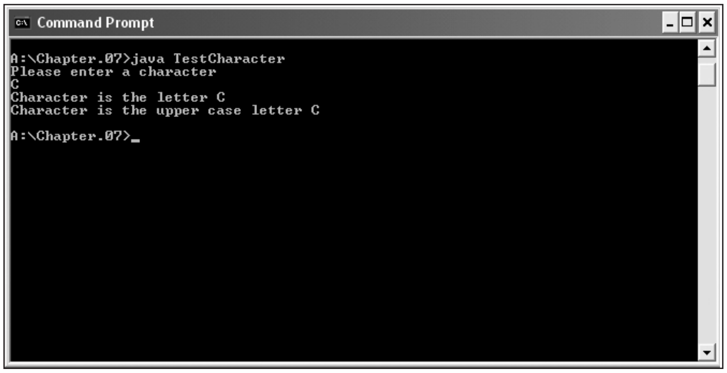
Figure 7-1 shows the Character class is defined in java.lang.Object and is automatically imported into every program you write. Commonly used methods available in the Character class are shown in Table 7-1.

Figure 7-2 contains a program that uses many of the methods shown in Table 7-1. The program declares a character variable named aChar and allows the user to supply a value for aChar by entering a single character from the keyboard. The program tests aChar using a series of `if` and `if…else` statements to determine whether the entered character is a digit, letter, or whitespace, and if it is a letter, whether it is lowercase or not.

```
public class TestCharacter
{
  public static void main(String[] args)throws Exception
  {
       char aChar;
       System.out.println("Please enter a character");
       aChar = (char)System.in.read();
       System.in.read();System.in.read();
       if(Character.isDigit(aChar))
              System.out.println(aChar+"is a number");
       else System.out.println(aChar+"is not a number");
       if(Character.isWhitespace(aChar))
              System.out.println(aChar+ "is a letter");
       else System.out.println(aChar+ "is not a letter");
       if(Character.isWhitespace(aChar))
              System.out.println
              ("Character is a whitespace character");
       else System.out.println
              ("Character is not a whitespace character");
       if(Character.isLetter(aChar))
              if(Character.isLowerCase(aChar))
                     System.out.println(aChar+
                     " is lowercase character");
              else
                     System.out.println(aChar+
                     " is not a lowercase character");
  }
}
```

Figure 7-2 The TestCharacter program

The output of the TestCharacter program for sample keyboard input of an uppercase character "C" is shown in Figure 7-3.



```
A:\Chapter.07>java TestCharacter
Please enter a character
C
Character is the letter C
Character is the upper case letter C

A:\Chapter.07>_
```

Figure 7-3 Sample output of the TestCharacter program

Declaring a String Object

You learned in Chapter 1 that a sequence of characters enclosed within double quotation marks is a literal string. You have used many literal strings, such as "First Java program," within println() statements.

Tip: You also use String in main() method headers.

A **String** variable is simply an object of the class String. The class String is defined in java.lang.String, which is automatically imported into every program you write. Figure 7-4 shows that the String class descends directly from the Object class. The String itself is distinct from the variable you use to refer to it. You create a String object by using the keyword `new` and the String constructor method, just as you would create an object of any other type. For example, `String aGreeting = new String("Hello");` is a statement that defines an object named aGreeting, declares it to be of type String, and assigns an initial value of "Hello" to the String. 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. Alternately, you can declare a String containing "Hello" with `String aGreeting = "Hello";`. Unlike other classes, the String class is special because you can create a String object without using the keyword `new` or calling the class constructor.

```
Java.lang.Object
  |
  +--java.lang.String
```

Figure 7-4 Structure of the String class

After declaring a String, you can display it in a print() or println() statement, just as you would for any other variable—for example, `System.out.println("The greeting is " + aGreeting);`.

Comparing String Values

In the Java programming language, String is a class, and each created String is a class 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, for example, `x = 45;`, then 45 replaces 10 at the assigned memory address. When you declare a String, such as `String aGreeting = "Hello";`, aGreeting holds a memory address where the characters "Hello" are stored. If you subsequently assign a new value to aGreeting, such as `aGreeting = "Bonjour";`, then the address held by aGreeting is altered; now aGreeting holds a new address where the characters "Bonjour" are stored. "Bonjour" is an entirely new object created with its own location. The "Hello" String is still in memory; it's just that aGreeting no longer holds its address. Eventually, a part of the Java system called the garbage collector will discard the "Hello" characters. Strings, therefore, are never actually changed; instead, new Strings are created and String variables hold the new addresses. Strings and other objects that can't be changed are known as **immutable**.

Because String variables hold memory addresses, you cannot make a simple comparison to determine whether two String objects are equivalent. For example, if you declare two Strings as `String aGreeting = "Hello";` and `String anotherGreeting = "Hello";`, Java will evaluate a comparison, such as `if(aGreeting == anotherGreeting)` as `false`. This is because when you compare aGreeting to anotherGreeting with the == operator, you are comparing their memory addresses, not their values.

Fortunately, the String class provides you with a number of useful methods. The **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. For example, Figure 7-5 shows two String objects and several comparisons. Each comparison in Figure 7-5 is true; each results in printing the line "Name's the same."

```
String aName = "Roger";
String anotherName = "Roger";
if(aName.equals(anotherName))
  System.out.println("Name's the same");
if(anotherName.equals(aName))
  System.out.println("Name's the same");
if(aName.equals("Roger");
  System.out.println("Name's the same");
```

Figure 7-5 String comparisons using the equals() method

Tip: The String class equals() method returns `true` only if two Strings are identical in content. Thus, a String holding "Roger " (with a space after the r) is not equivalent to a String holding "Roger" (with no space after the r).

Each String shown in Figure 7-5 (aName and anotherName) is an object of type String, so each String has access to the String class equals() method. The aName object can call equals() with `aName.equals()`, or the anotherName object can call equals() with `anotherName.equals()`. The equals() method can take either a variable String object or a literal string as its argument.

The **equalsIgnoreCase() method** is similar to the equals() method. As its name implies, it ignores case when determining if two Strings are equivalent. Thus, `aName.equals("roGER")` is `false`, but `aName.equalsIgnoreCase("roGER")` 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. The equalsIgnoreCase() method allows you to test entered data without regard to capitalization.

When the **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 hold 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 holds "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 will 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 holds the value "hamster," and anotherWord is a String variable that holds the value "iguana," then the comparison `if(aWord.compareTo(anotherWord)<0)` yields `true`.

Using Other String Methods

A wide variety of additional String 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 `aWord = aWord.toUpperCase` assigns "SOMETHING" to aWord. Because aWord now is assigned "SOMETHING," `aWord = aWord.toLowerCase()` assigns "something" to aWord. 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 begins with zero rather than 1. 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('a')` is 2, and the value of `myName.indexOf('q')` is -1.

The **charAt() method** requires an integer argument which indicates the position of the character that the method returns. For example, if myName is a String holding "Stacy," then the value of `myName.charAt(0)` is 'S' and `myName.charAt(1)` is 't'.

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`.

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 **toString() method** converts any primitive type to a String. So, if you declare a String as `theString` and an integer as `int someInt = 4;`, then `theString = Integer.toString(someInt);` results in the String "4" being assigned to theString. If you declare another String as `aString` and a double as `double someDouble = 8.25`, then `aString = Double.toString(someDouble);` assigns the String "8.25" to aString.

Another method is available to convert any primitive type to a String. If you declare a String as `anotherString` and a float as `float someFloat = 12.34f`, then `anotherString = "" + someFloat`, assigns the String "12.34" to anotherString. The Java interpreter first converts the float 12.34f to a String "12.34," and adds it to the null String "". Joining Strings is called **concatenation**. The resulting string "12.34" is then assigned to anotherString.

Tip: The toString() method is not part of the String class; it is a method included in Java that you can use with any type of object. 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.

Tip: Because the toString() method takes arguments of any primitive type, including int, char, double, and so on, it is an overloaded method.

You already know that you can join Strings with other Strings or values by using a plus sign (+); you have used this approach in println() statements since Chapter 1. For example, you can print a firstName, a space, and a lastName with `System.out.println(firstName + " " + lastName);`. Additionally, 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 arguments—a start position and an end position—that are both based on the fact that a String's first position is position zero. For example, the program segment in Figure 7-6 shows the names Monday through Friday as Strings representing the days of the week. An abbreviation of a weekday, Monday for example, can be printed after using the substring method `System.out.println("The abbreviation for Monday is " + monday.substring(0,3));`. Here `monday.substring(0,3)` starts at the first character (index of 0), and extracts the first three letters from the string stored as "Monday." The output of the weekday abbreviations program is shown in Figure 7-7.

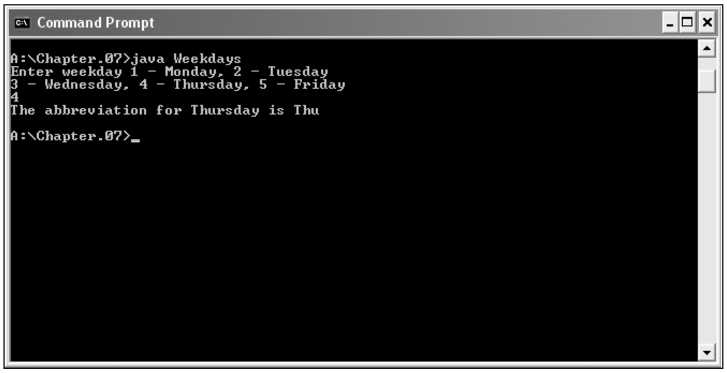
```
public class Weekdays
{
   public static void main(String() args)throws Exception
   {
      char weekday;

      String monday = "Monday";
      String tuesday = "Tuesday";
      String wednesday = "Wednesday";
      String thursday = "Thursday";
      String friday = "Friday";

      System.out.println("Enter weekday 1 - Monday, 2 - Tuesday");
      System.out.println("3 - Wednesday, 4 - Thursday, 5 - Friday");
      weekday = (char)System.in.read();
      System.in.read();System.in.read();

      if(weekday == '1')
        System.out.println("The abbreviation for Monday is " +
          monday.substring(0,3));
      else if(weekday == '2')
        System.out.println("The abbreviation for Tuesday is " +
          tuesday.substring(0,3));
      else if(weekday == '3')
        System.out.println("The abbreviation for Wednesday is " +
          wednesday.substring(0,3));
      else if(weekday == '4')
        System.out.println("The abbreviation for Thursday is " +
          thursday.substring(0,3));
      else
        System.out.println("The abbreviation for Friday is " +
          friday.substring(0,3));
   }
}
```

Figure 7-6 Program segment demonstrating the substring method and String concatenation



```
A:\Chapter.07>java Weekdays
Enter weekday 1 - Monday, 2 - Tuesday
3 - Wednesday, 4 - Thursday, 5 - Friday
4
The abbreviation for Thursday is Thu

A:\Chapter.07>_
```

Figure 7-7 Output of the substring method and String concatenation code segment

To demonstrate the use of the String methods, you will create a simple guessing game, similar to Hangman. The user first will guess letters, and then attempt to guess the motto of Event Handlers Incorporated.

To create the guessing game:

1. Open a new text file in your text editor. Enter the following first few lines of a SecretPhrase program. The program will contain the target phrase that the user will try to guess ("Plan With Us"), as well as a display phrase that is mostly asterisks (with a few hints).

   ```
   public class SecretPhrase
   {
     public static void main(String[] args) throws Exception
     {
       String targetPhrase = "Plan With Us";
       String displayPhrase = "P*** W*** U*";
   ```

2. Add the following variables that will hold the user's guess and the position of a guess that is found within the phrase:

   ```
   char guess;
   int position;
   ```

3. Next, add the following brief instruction:

   ```
   System.out.println("Play our game - guess our motto");
   ```

4. Enter the statement to display the hint phrase:

   ```
   System.out.println(displayPhrase);
   ```

5. Add the following loop that continues while asterisks remain in the displayPhrase. The user will enter a letter. You will use the indexOf() method to determine whether the guessed letter appears in the targetPhrase. If it does not, then ask the user to guess again. If the guessed letter appears in the phrase, you reconstruct the display phrase with the following:

   - The substring of characters in the display phrase that comes before the correct guess
   - The correct guess
   - The substring of characters in the display phrase that appears after the correct guess; in other words, the correct letter replaces the appropriate asterisk

   Add the following code:

   ```
   while(displayPhrase.indexOf('*') != -1)
   {
     System.out.println("Enter a letter");
     guess = (char)System.in.read();
   ```

```
    System.in.read(); System.in.read();
      // Absorbs Enter key
    position = targetPhrase.indexOf(guess);
      // Determines position of guess
    if(position == -1) // If guess is not in target phrase
      System.out.println("Sorry, guess again");
    else // If guess is in target phrase
    {
      displayPhrase = displayPhrase.substring(0,position) +
        guess + displayPhrase.substring
          (position+1,displayPhrase.length());
      System.out.println(displayPhrase);
    }
  }
```

6. The `while` loop will continue until all the asterisks in the targetPhrase are replaced by correct letters. Therefore, after the closing curly brace for the `while` loop, enter:

   `System.out.println("Congratulations!");`.

7. Type the closing curly braces for the main() method and for the SecretPhrase class.
8. Save the program as **SecretPhrase.java** in the Chapter.07 folder on your Student Disk, then compile and run the program. Make sure you understand how all the String methods contribute to the success of this program.

Converting Strings 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 use it like any other number. To convert a String to an integer, you use the **Integer class**, which is part of java.lang and automatically imported into programs you write. The **parseInt() method** is part of the Integer class, and takes a String argument and returns its integer value. For example, `int anInt = Integer.parseInt("649");` stores the numeric value 649 in the variable anInt. You can then use the integer value just as you would any other integer.

Tip

The word parse in English means "to resolve into component parts," as when you "parse a sentence." In Java, to parse a String means to break down its separate characters into a numeric format.

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 also imported into your programs automatically. A method of the Double class is parseDouble(), which takes a String argument and returns its double value. For example, `double doubleValue = Double.parseDouble("147.82");` stores the numeric value 147.82 in the variable doubleValue.

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

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

The 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 double that is stored in value.

Tip: The Double and Integer classes are examples of wrappers. A wrapper is a class or object that is "wrapped around" a simpler thing. You use the Double (uppercase D) class to make it convenient to work with primitive double (lowercase d) variables.

When planning an event, Event Handlers Incorporated must know how many guests to expect. Next you will prompt the user for the number of guests, read characters from the keyboard, store the characters in a String, and then convert the String to an integer.

To create a program that accepts integer input:

1. Open a new text file in your text editor. Type the statement **`import javax.swing.*;`**, press **[Enter]**, and then enter the following first few lines of a DialogInput class that will accept string input:

```
public class NumInput
{
  public static void main(String[] args) throws Exception
  {
```

2. Declare the following variables to hold the input String and the resulting integer:

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

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

```
inputString = JOptionPane.showInputDialog(null,
  "Enter the number of guests at your event");
```

4. Use the following Integer.parseInt() method to convert the input String to an integer. Then use the integer in a numeric decision that displays a message dialog box when the number of guests entered is greater than 100:

```
inputNumber = Integer.parseInt(inputString);
if(inputNumber > 100)
  JOptionPane.showMessageDialog(null,
    "A surcharge will  apply!);
```

5. Enter the closing statement `System.exit(0);` press **[Enter]**, and then enter the final two closing curly braces for the program.
6. Save the program as **NumInput.java** in the Chapter.07 folder on your Student Disk, then compile and test the program.

Learning About the StringBuffer Class

A String class limitation is that the value of a String is fixed after the String is created; Strings are immutable. When you write `someString = "Hello";` and follow it with `someString = "Goodbye";`, you have neither changed the contents of computer memory at someString, nor have you 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.

To circumvent these limitations, you can use the StringBuffer class. **StringBuffer** is an alternative to the String class, and can usually be used anywhere you would use a String. The structure of the StringBuffer class is shown in Figure 7-8. Like the String class, the StringBuffer class is part of the java.lang package and is automatically imported into every program.

```
Java.lang.Object
  |
  +--java.lang.StringBuffer
```

Figure 7-8 Structure of the StringBuffer class

You can create a StringBuffer object that contains a given String with the statement `StringBuffer eventString = new StringBuffer("Event Handlers Incorporated");`. When you initialize a `StringBuffer` object you must use the keyword `new` and provide an initializing value between parentheses. You can create the StringBuffer variable using syntax similar to the syntax for creating a String variable, such as `StringBuffer philosophyString = null;`. The variable does not refer to anything until you initialize it with a defined `StringBuffer` object. For example, you could write `philosophyString = new StringBuffer("Dedicated to making your event a most memorable one");`. You can also initialize the StringBuffer variable philosophyString with an existing StringBuffer object using `philosophyString =  eventString;`.

Generally when you create a String object, sufficient memory is allocated to accommodate the number of Unicode characters in the string. A StringBuffer object, however, contains a memory block called a **buffer** which might not contain a string. Even if it does contain a String, the String might not occupy all of the 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 referred to as the **capacity** of the StringBuffer object. It is generally more efficient to make the StringBuffer capacity sufficient for the needs of your program, rather than modify a String that has been originally stored with a small capacity.

You can change the length of a String in a StringBuffer object with the setLength() method. The length is a property of the String held by the StringBuffer. When you increase a StringBuffer 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 StringBuffer object, you use the capacity() method. For example, the EventStringBuffer program in Figure 7-9 shows the creation of the eventString object as `StringBuffer eventString = new StringBuffer("Event Handlers Incorporated");`. The capacity of the StringBuffer object is obtained as `int aCapacity = eventString.capacity();` and printed using `System.out.println("The capacity is " + aCapacity);`. Figure 7-10 shows the StringBuffer capacity is 43. Note that the capacity of 43 is 16 characters larger than the length of the string "Event Handlers Incorporated," which contains 27.

Tip

In general, when a StringBuffer object is created from a String, the capacity will be the length of the string plus 16.

In Figure 7-9 the philosophyString variable is created as `StringBuffer philosophyString = null;`. The variable does not refer to anything until it is initialized with the defined StringBuffer object `philosophyString = new StringBuffer("Dedicated to making your event a most memorable one");`. The capacity of philosophyString is shown in Figure 7-10 as the length of the string plus 16 or 67.

In the program shown in Figure 7-9, the length of the eventString is changed with the statement `eventString.setLength(40);`. When the value of the length prints, as shown in Figure 7-10, the extra characters show as blanks before the String "end" is printed. Also in Figure 7-10, the philosophyString length is shortened to a length of 30 in the statement `philosophyString.setLength(30);`. The shortened 30-character output string is shown as "Dedicated to making your event" in Figure 7-10.

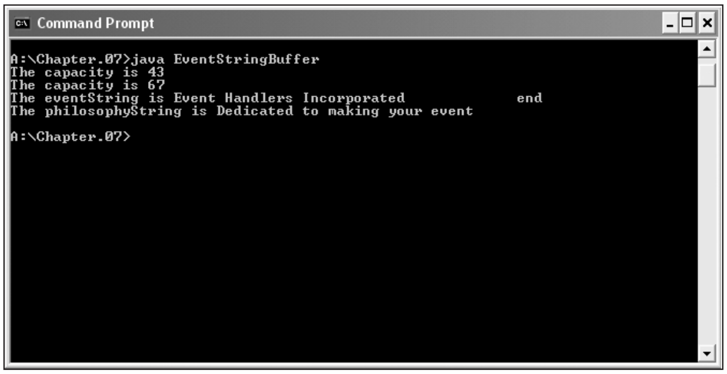
```
public class EventStringBuffer
{
  public static void main(String[] args)
  {
  StringBuffer eventString =
        new StringBuffer("Event Handlers Incorporated");
     int aCapacity = eventString.capacity();
     System.out.println("The capacity is " + aCapacity);

     StringBuffer philosophyString = null;
     philosophyString =
        new StringBuffer("Dedicated to making your event a most memorable one");
     int bCapacity = philosophyString.capacity();
     System.out.println("The capacity is " + bCapacity);

     eventString.setLength(50);
     System.out.println("The eventString is " + eventString + "end");

     philosophyString.setLength(30)
        System.out.println("The philosophyString is " + philosophyString);
  }
}
```

Figure 7-9 EventStringBuffer program



```
A:\Chapter.07>java EventStringBuffer
The capacity is 43
The capacity is 67
The eventString is Event Handlers Incorporated                end
The philosophyString is Dedicated to making your event

A:\Chapter.07>
```

Figure 7-10 Output of the EventStringBuffer program

Using StringBuffer objects provides more flexibility than String objects because you can insert or append new contents into a StringBuffer. The StringBuffer class provides you with three constructors:

- `public StringBuffer()` constructs a StringBuffer with no characters and a default size of 16 characters
- `public StringBuffer(int length)` constructs a StringBuffer with no characters, and a capacity specified by length

- `public StringBuffer(String s)` contains the same characters as those stored in the String object s (The capacity of the StringBuffer is the length of the String argument you provide, plus 16 additional characters.)

The **append() method** lets you add characters to the end of a StringBuffer object. For example, if a StringBuffer object is declared as `StringBuffer someBuffer = new StringBuffer("Happy");`, then the statement `someBuffer.append (" birthday")` alters someBuffer to hold "Happy birthday."

The **insert() method** lets you add characters at a specific location within a StringBuffer object. For example, if someBuffer holds "Happy birthday," then `someBuffer.insert(6, "30th ");` alters the StringBuffer to contain "Happy 30th birthday." The first character in the StringBuffer object occupies position zero. To alter just one character in a StringBuffer, you can use the **setCharAt() method**. This method requires two arguments, an integer position, and a character. If someBuffer holds "Happy 30th birthday," then `someBuffer.setCharAt(6,'4');` changes the someBuffer value into a 40th birthday greeting.

Next you will use StringBuffer methods.

To use StringBuffer methods:

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

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

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

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

3. Enter the following append() method to add characters to the existing StringBuffer and print it again:

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

4. Enter the following insert() method to insert characters, print, insert additional characters, and print the StringBuffer again:

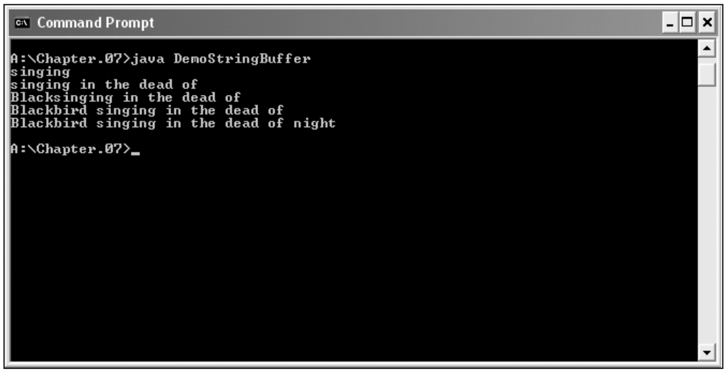
```
str.insert(0, "Black");
print(str);
str.insert(5, "bird ");
print(str);
```

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 prints StringBuffer objects:
```
public static void print(StringBuffer s)
{
  System.out.println(s);
}
```
8. Type the closing curly brace for the class, and then save the file as **DemoStringBuffer.java** in the Chapter.07 folder on your Student Disk. Compile and execute, and then compare your output to Figure 7-11.

You can extract characters from a StringBuffer object using the charAt()and getChars() methods. The **charAt() method** accepts an argument that is the offset of the character position from the beginning of a String. If you declare `StringBuffer text = new StringBuffer("Java  Programming");`, then `text.charAt(5)` refers to the character 'P.'



```
A:\Chapter.07>java DemoStringBuffer
singing
singing in the dead of
Blacksinging in the dead of
Blackbird singing in the dead of
Blackbird singing in the dead of night

A:\Chapter.07>_
```

Figure 7-11 Output of the DemoStringBuffer program

Tip: If you try to use an index less than 0 or greater than the index of the last position in the StringBuffer object, you will cause an exception to be thrown and your program will terminate.

Finally, you can change a single character in a StringBuffer object using the setCharAt() method with two arguments. The first argument indicates the index position of the character to be changed; the second argument specifies the replacement character. If you declare `StringBuffer text = "java";`, the statement `text.setCharAt(0,'J')` sets the first character to 'J'.

Chapter Summary

- A sequence of characters enclosed within double quotation marks is a literal string.
- You create a String object by using the keyword `new` and the String constructor method.
- Each String is a class object, and a String variable name is actually a reference. Strings, therefore, are never changed; they are immutable.
- The equals() method evaluates the contents of two String objects to determine whether they are equivalent, and then returns a Boolean value.
- The equalsIgnoreCase() method determines if two Strings are equivalent without considering case.
- The compareTo() method returns zero if two String objects hold the same value. 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 "greater than" the argument.
- The methods toUpperCase() and toLowerCase() convert any String to its uppercase or lowercase equivalent.
- The indexOf() method determines whether a specific character occurs within a String. If it does, the method returns the position of the character. The return value is -1 if the character does not exist in the String.
- The endsWith() and startsWith() methods each take a String argument and return `true` or `false`, depending on whether or not a String object ends with or starts with the specified argument.
- The replace() method allows you to replace all occurrences of some character within a String.
- The toString() method converts any primitive type 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, which takes two arguments, and a start and end position, both of which are based on the fact that a String's first position is position zero.
- If a String contains all numbers, you can convert it to a number.
- The parseInt() method takes a String argument and returns its integer value.
- The Double.valueOf() method converts a String to a Double object; the doubleValue() method converts a Double object to a double variable.
- To circumvent some limitations of the String class, you can use the StringBuffer class. You can insert or append new contents into a StringBuffer.

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 ________.
   a. `object`
   b. `create`
   c. `char`
   d. `new`
3. A String variable name is a ________.
   a. reference
   b. value
   c. constant
   d. literal
4. Objects that cannot be changed are ________.
   a. irrevocable
   b. nonvolatile
   c. immutable
   d. stable
5. If you declare two String objects as `String word1 = new String("happy");` and `String word2 = new String("happy");`, then 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");` and `String word2 = new String("happy");`, then 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 using `String name1 = new String("Jordan");` and `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");`, then 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");`, then 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 any ________________ to a String.
    a. character
    b. integer
    c. float
    d. all of the above
14. Joining Strings is called ________________.
    a. chaining
    b. joining
    c. linking
    d. concatenation
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 substring() method requires ________________ arguments.
    a. no
    b. one
    c. two
    d. three
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; they both are primitive types
    d. nonexistent; both are classes
19. For an alternative to the String class, you can use ________________.
    a. char
    b. StringHolder
    c. StringBuffer
    d. StringMerger

20. The default capacity for a StringBuffer object is ______________ characters.
    a. zero
    b. two
    c. 16
    d. 32

EXERCISES

1. Write a program that concatenates the three Strings: "Event Handlers is dedicated", "to making your event", and "a most memorable one." Print each String and the concatenated String. Save the program as **JoinStrings.java** in your Chapter.07 folder on your Student Disk.
2. Write a program that calculates the total number of vowels contained in the String "Event Handlers is dedicated to making your event a most memorable one." Save the program name **StringVowels.java** in your Chapter.07 folder on your Student Disk.
3. Write a program that calculates the total number of letters contained in the String "Event Handlers Incorporated, 8900 U.S. Highway 14, Crystal Lake, IL 60014". Save the program name as **StringLetters.java** in your Chapter.07 folder on your Student Disk.
4. Write a program that calculates the total number of whitespaces contained in the String "[TAB][TAB]   ", which represents two tabs and three spaces. Save the program as **StringWhite.java** in your Chapter.07 folder on your Student Disk.
5. Write a program that converts the variables someInt and someDouble in the statements `int someInt = 21;` and `someDouble = 128.04;` to strings using the classes Integer and Double. Save the program as **ToString.java** in your Chapter.07 folder on your Student Disk.
6. Write a program that converts the variables someInt, someDouble, and someFloat in the statements `int someInt = 567;`, `double someDouble = 48.25;`, and `float someFloat = 443.21f;` to strings. Do not use the Integer, Double, and Float classes to make the conversions. Save the program as **ToString2.java** in your Chapter.07 folder on your Student Disk.
7. a. Write a program that demonstrates that when two identical names are compared and the case differs, the equals method will return `false` when making a comparison. Save the program as **Comparison.java** in your Chapter.07 folder on your Student Disk.
   b. Demonstrate that the equalIgnoreCase() method will change the comparison in question 7a. from `false` to `true`.

8. Write a program to demonstrate that the compareTo() method returns either a positive number, a negative number, or a zero when used to compare two Strings. Save the program as **Compare.java** in your Chapter.07 folder on your Student Disk.
9. Write a program to demonstrate the following, based on the statement `String dedicate = "Dedicated to making your event a most memorable one":`
   a. index of 'D'
   b. char at (15)
   c. endsWith(one)
   d. eplace('a', 'A').

   Save the program as **Demonstrate.java** in your Chapter.07 folder on your Student Disk.
10. Create a class that holds three initialized StringBuffer objects: your first name, middle name, and last name. Create three new StringBuffer objects as follows:
    - An object named EntireName that holds your three names, separated by spaces
    - An object named LastFirst that holds your last name, a comma, a space, and your first name, in that order
    - An object named Signature that holds your first name, a space, your middle initial (not the entire name), a period, a space, and your last name

    Display all three objects. Save the program as **Buffer.java** in the Chapter.07 folder on your Student Disk.
11. Each of the following files in the Chapter.07 folder on your Student Disk has syntax and/or logical 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

CASE PROJECT

Case Project

The Tax Advantage Company provides free tax services to residents who cannot calculate their personal taxes or pay for calculation by a public tax service. You have been asked to write a Java program that will calculate an estimated tax for either a single or married taxpayer, given a keyboard income entry. After the income is entered, a code status "S" for single or "M" for married is entered.

The necessary classes are a Tax class with a main() method for keyboard entry of income and code status values, and a TaxReturn class to calculate the tax based on the input. The input values should be passed from the Tax class to the TaxReturn class through the instantiation of a TaxReturn object. The instantiation could take a form such as `TaxReturn aTaxReturn = new TaxReturn(income, status);`.

For taxpayers, two tax rates are needed, 15 percent and 30 percent. For single taxpayers, the cutoff rate of 15 percent is $10,000, and 30 percent above $10,000. For married taxpayers, the cutoff rate is 15 percent for amounts below $20,000, and 30 percent for amounts of $20,000 and above.

Program output should show the code status, income, and the amount of tax. Save the program files as **Tax.jav**a and **TaxReturn.java** in the Chapter.07 folder on your Student Disk.