JAVA™ PROGRAMMING

Chapter 7: Characters, Strings, and the `StringBuilder`

Objectives

- Identify string data problems
- Use `Character` class methods
- Declare and compare `String` objects
- Use other `String` methods
- Use the `StringBuilder` and `StringBuffer` classes

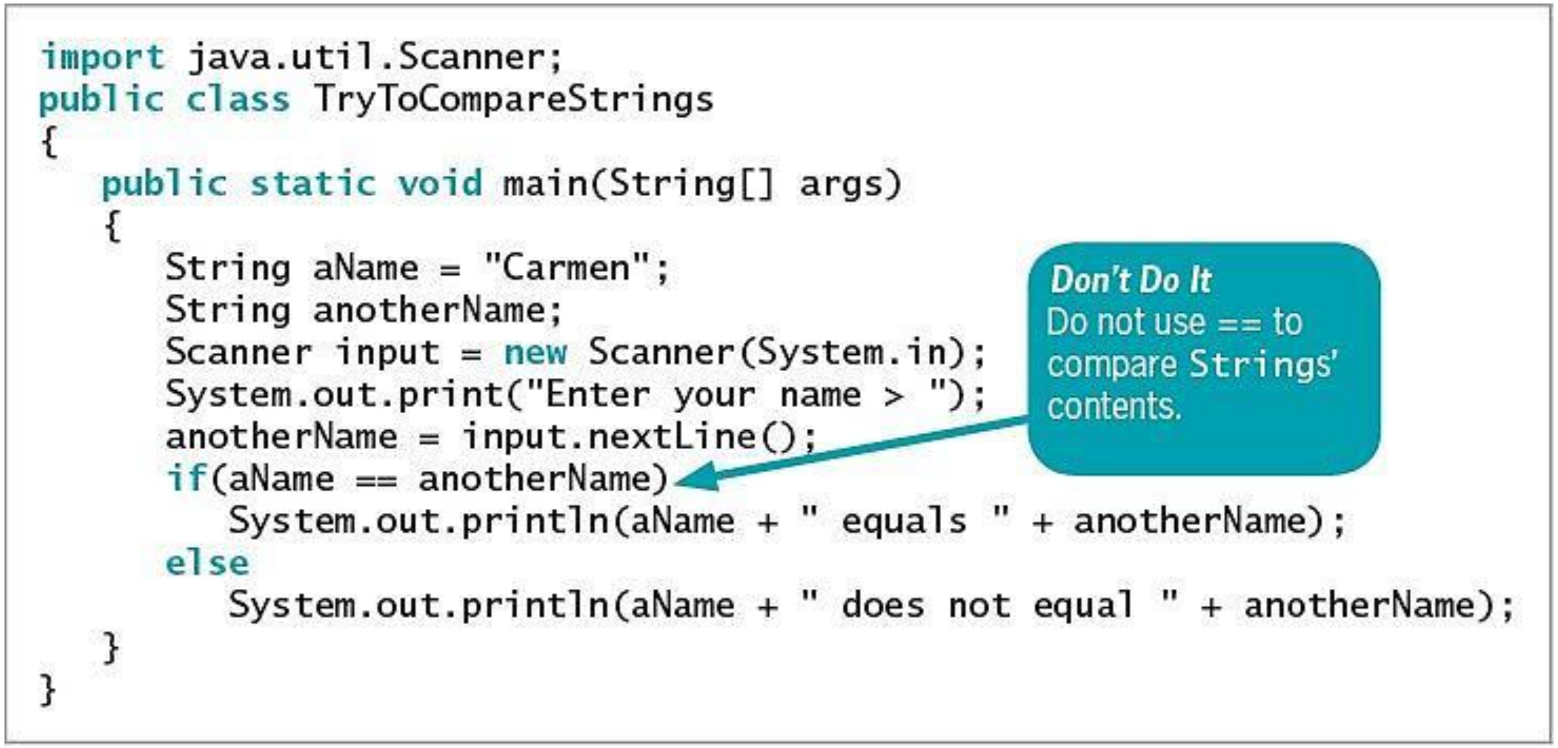
Understanding String Data Problems

- Manipulating characters and groups of characters provides some challenges for the beginning Java programmer
- A `String` is a class
  - Each created `String` is a class object
  - The `String` variable name is not a simple data type
  - **Reference**
    - A variable that holds a memory address

Understanding String Data Problems (cont'd.)

- Compare two `String`s using the `==` operator
  - Not comparing values
  - Comparing computer memory locations
- Compare contents of memory locations more frequently than memory locations themselves

Understanding String Data Problems (cont'd.)



```
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);
   }
}
```

Figure 7-1 The TryToCompareStrings application

Understanding String Data Problems (cont'd.)

- Classes to use when working with character data
  - **`Character`**
    - Instances hold a single character value
    - Defines methods that can manipulate or inspect single-character data
  - **`String`**
    - A class for working with fixed-string data
      - Unchanging data composed of multiple characters

Understanding String Data Problems (cont'd.)

- Classes to use when working with character data (cont'd.)
  - `StringBuilder` and `StringBuffer`
    - Classes for storing and manipulating changeable data composed of multiple characters

Using `Character` Class Methods

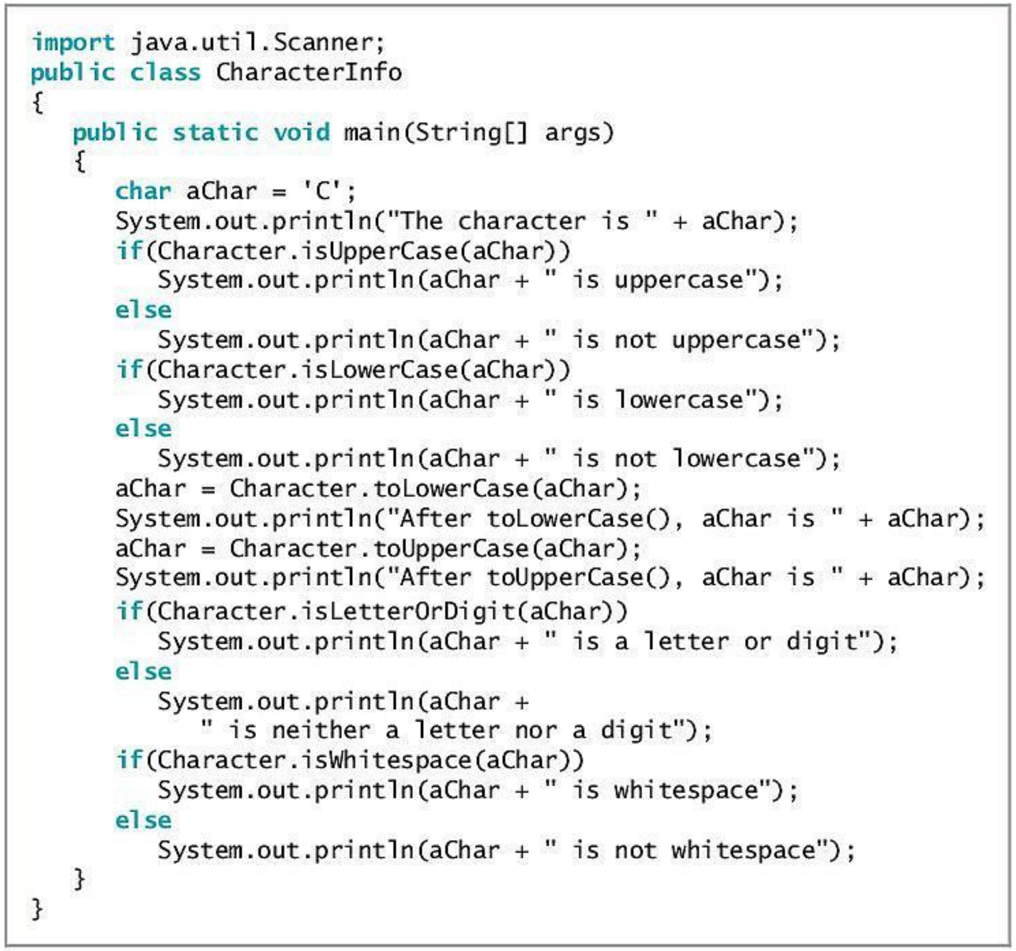
- `Character` class
  - Contains standard methods for testing the values of characters
  - Methods that begin with “is”
    - Such as `isUpperCase()`
    - Return a Boolean value that can be used in comparison statements
  - Methods that begin with “to”
    - Such as `toUpperCase()`
    - Return a character that has been converted to the stated format

Manipulating Characters (cont'd.)

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

Manipulating Characters (cont'd.)



```java
import java.util.Scanner;
public class CharacterInfo
{
   public static void main(String[] args)
   {
      char aChar = 'C';
      System.out.println("The character is " + aChar);
      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 CharacterInfo application

Declaring and Comparing `String` Objects

- Literal string
  - A sequence of characters enclosed within double quotation marks
  - An unnamed object, or **anonymous object**, of the `String` class
- **`String` variable**
  - A named object of the `String` class
- Class `String`
  - Defined in `java.lang.String`
  - Automatically imported into every program

Declaring and Comparing `String` Objects (cont'd.)

- Declare a `String` variable
  - The `String` itself is distinct from the variable used to refer to it
- Create a `String` object

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

  - You can create a `String` object without:
    - Using the keyword `new`
    - Explicitly calling the class constructor

Comparing `String` Values

- `String` is a class
  - Each created `String` is a class object
- `String` variable name
  - A reference variable
  - Refers to a location in memory
    - Rather than to a particular value
- Assign a new value to a `String`
  - The address held by the `String` is altered

Comparing `String` Values (cont'd.)

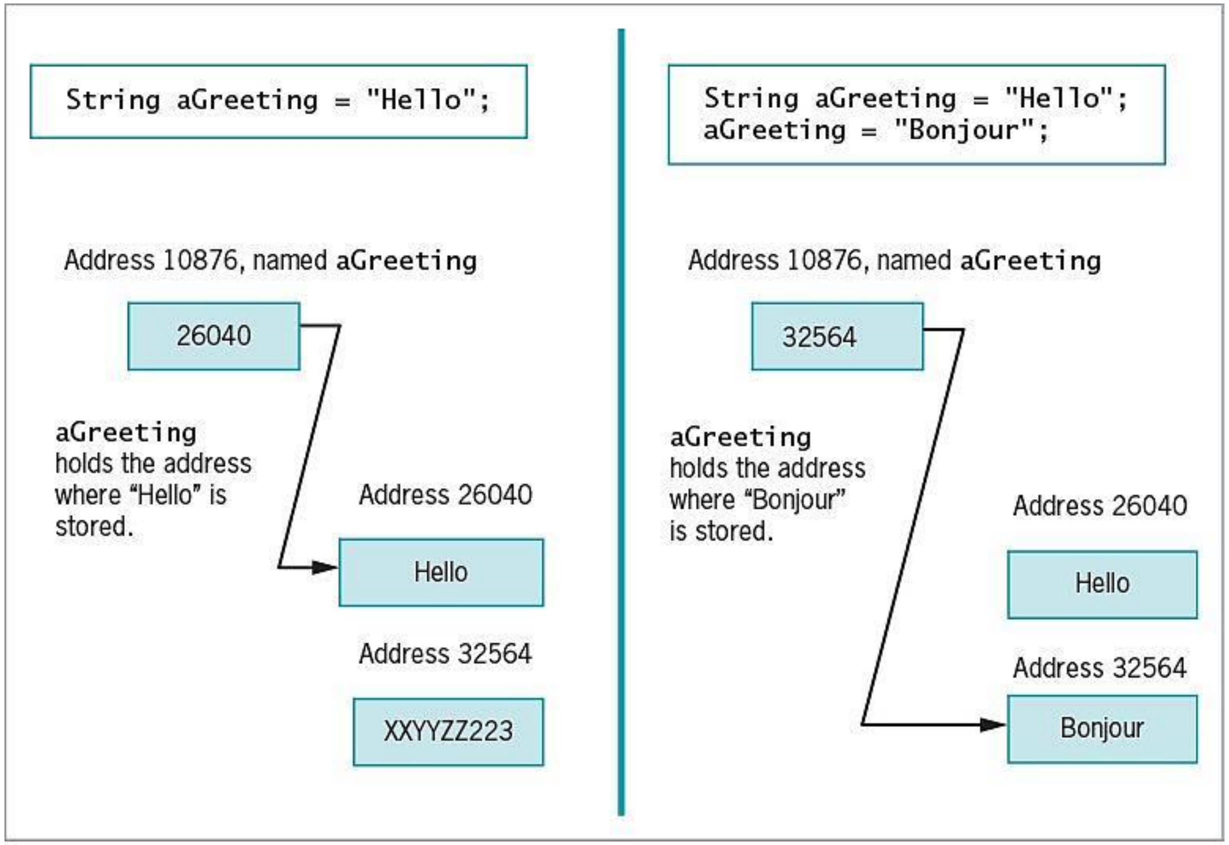

String aGreeting = "Hello";
Address 10876, named aGreeting
26040
aGreeting holds the address where "Hello" is stored.
Address 26040
Hello
Address 32564
XXYYZZ223
String aGreeting = "Hello";
aGreeting = "Bonjour";
Address 10876, named aGreeting
32564
aGreeting holds the address where "Bonjour" is stored.
Address 26040
Hello
Address 32564
Bonjour

Figure 7-5 Contents of `aGreeting` at declaration and after an assignment

Comparing `String` Values (cont'd.)

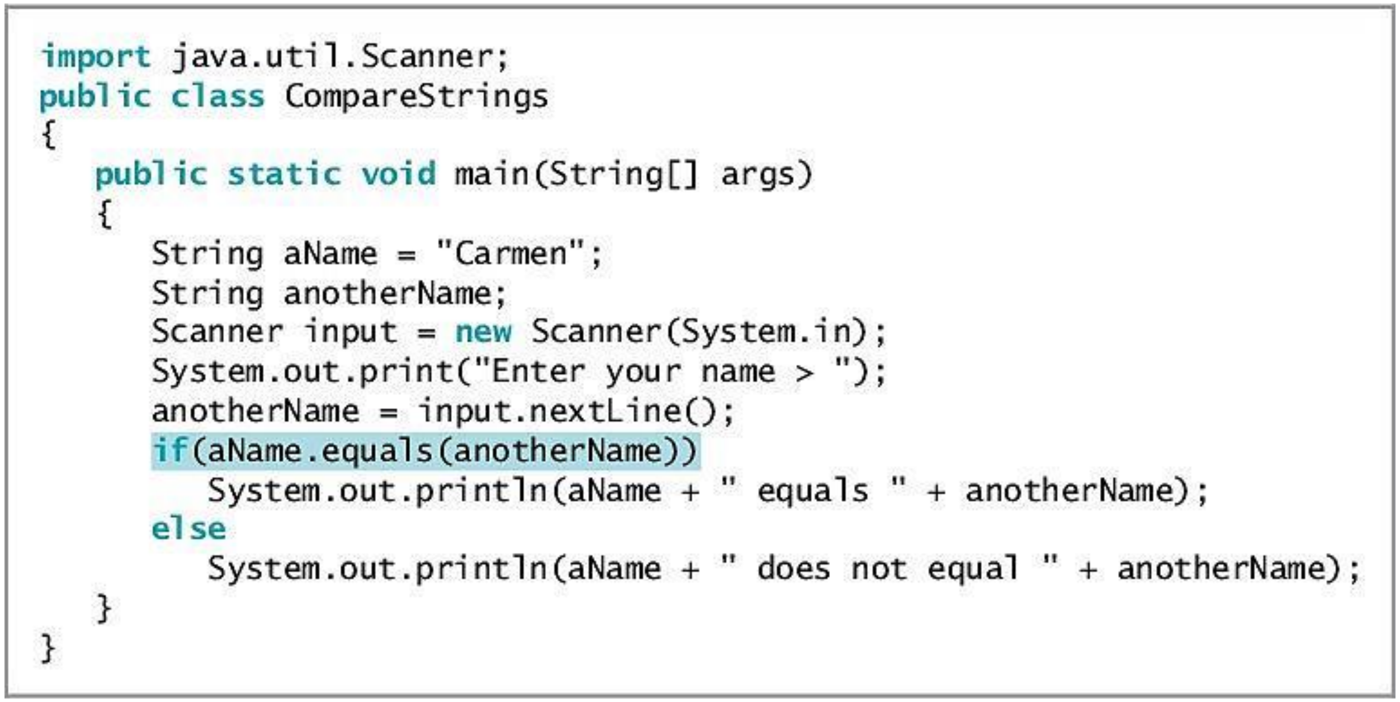
- **Immutable**
  - Objects that cannot be changed, such as a `String`
- Making simple comparisons between `String`s often produces misleading results
- Comparing `String`s using the `==` operator
  - Compares memory addresses, not values

Comparing `String` Values (cont'd.)

- **`equals()` method**
  - Evaluates the contents of two `String` objects to determine if they are equivalent
  - Returns `true` if objects have identical contents

    `public boolean equals(String s)`
- **`equalsIgnoreCase()` method**
  - Ignores case when determining if two `String`s are equivalent
  - Useful when users type responses to prompts in programs

Comparing `String` Values (cont'd.)



```
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 7-6 The CompareStrings application

Comparing `String` Values (cont'd.)

- **`compareTo()` method**
  - Compares two `String`s and returns:
    - Zero: If two `String`s refer to the same value
    - Negative number: If the calling object is “less than” the argument
    - Positive number: If the calling object is “more than” the argument

```
if (aWord.compareTo(anotherWord) < 0)
```

Empty and `null` `Strings`

- Empty `Strings`
- Reference a memory address with no characters
  - Can be used in `String` methods
- **`null Strings`**
  - Use the `null` Java keyword
  - `Strings` are set to `null` by default
  - Cannot be used in `String` methods

Using Other `String` Methods

- **`toUpperCase()`** and **`toLowerCase()`** methods
  - Convert any `String` to its uppercase or lowercase equivalent
- **`length()` method**
  - Returns the length of a `String`

Using Other `String` Methods (cont'd.)

- **`indexOf()` method**
  - Determines whether a specific character occurs within a `String`
  - 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`

Using Other `String` Methods (cont'd.)

- **`charAt()` method**
  - Requires an integer argument
  - Indicates the position of the character that the method returns
- **`endsWith()` method** and **`startsWith()` method**
  - Each takes a `String` argument
  - Return `true` or `false` if a `String` object does or does not end or start with the specified argument, respectively

Using Other `String` Methods (cont'd.)

- **`replace()` method**
  - Replaces all occurrences of some character within a `String`
- **`toString()` method**
  - Not part of the `String` class
  - Converts any object to a `String`
  - Converts primitive data types to `Strings`

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

Using Other `String` Methods (cont'd.)

- **Concatenation**
  - Join a simple variable to a `String`

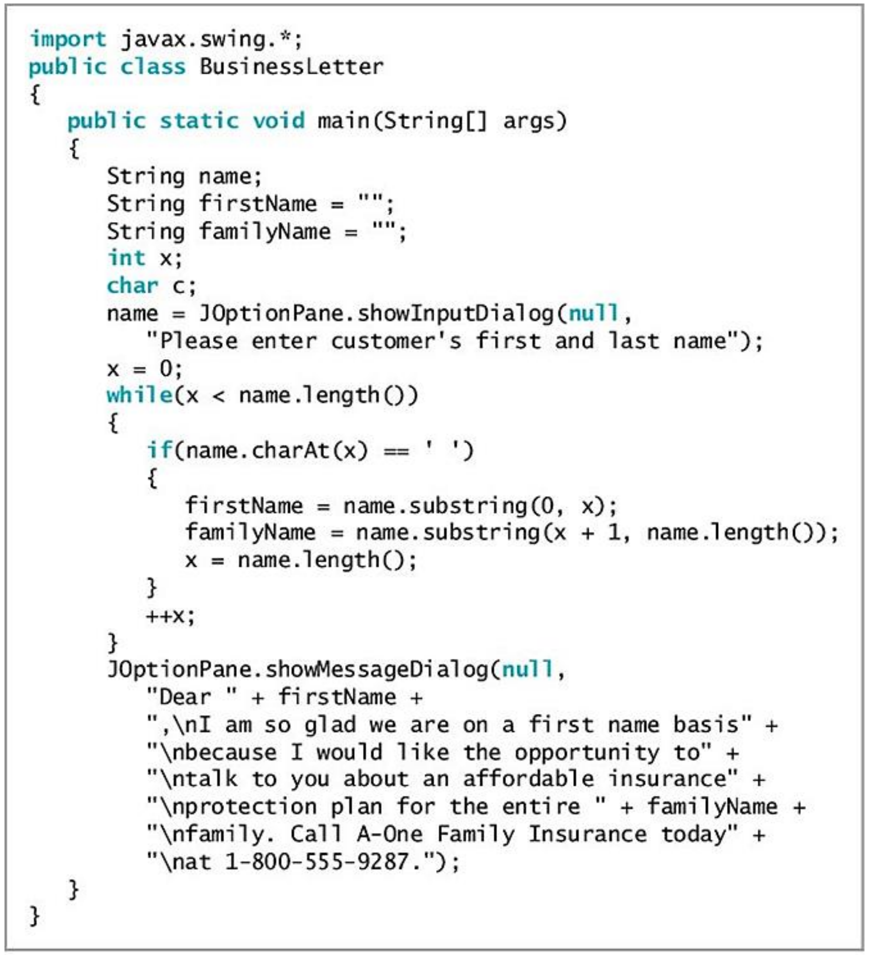
```
String aString = "My age is " + myAge;
```

  - Use the + operator

Using Other `String` Methods (cont'd.)

- **`substring()` method**
  - Extracts part of a `String`
  - Takes two integer arguments
    - Start position
    - End position
  - The length of the extracted substring is the difference between the second integer and the first integer

Using Other `String` Methods (cont'd.)



```
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 7-8 The `BusinessLetter` application

Using Other `String` Methods (cont'd.)

- `regionMatches()` method
  - Two variants that can be used to test if two `String` regions are equal
- A substring of the specified `String` object is compared to a substring of the other
  - If the substrings contain the same character sequence, then the expression is `true`
  - Otherwise, the expression is `false`
- A second version uses an additional `boolean` argument
  - Determines whether case is ignored when comparing characters

Converting `String` Objects to Numbers

- **`Integer` class**
  - Part of `java.lang`
  - Automatically imported into programs
  - Converts a `String` to an integer
  - **`parseInt()` method**
    - Takes a `String` argument
    - Returns its integer value
- **Wrapper**
  - A class or an object “wrapped around” a simpler element

Converting `String` Objects to Numbers (cont'd.)

- `Integer` class `valueOf()` method
  - Converts a `String` to an `Integer` class object
- `Integer` class `intValue()` method
  - Extracts the simple integer from its wrapper class
- **`Double` class**
  - A wrapper class
  - Imported into programs automatically
  - **`parseDouble()` method**
    - Takes a `String` argument and returns its `double` value

Learning About the `StringBuilder` and `StringBuffer` Classes

- The value of a `String` is fixed
  - After a `String` is created, it is immutable
- **`StringBuilder`** and **`StringBuffer`** classes
  - An alternative to the `String` class
  - Used when a `String` will be modified
  - Can use anywhere you would use a `String`
  - Part of the `java.lang` package
  - Automatically imported into every program

Learning About the `StringBuilder` and `StringBuffer` Classes (cont'd.)

- `StringBuilder`
  - More efficient
- `StringBuffer`
  - Thread safe
  - Use in multithreaded programs

Learning About the `StringBuilder` and `StringBuffer` Classes (cont'd.)

- Create a `StringBuilder` object

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

  - Must use:
    - The keyword `new`
    - The constructor name
    - An initializing value between the constructor's parentheses

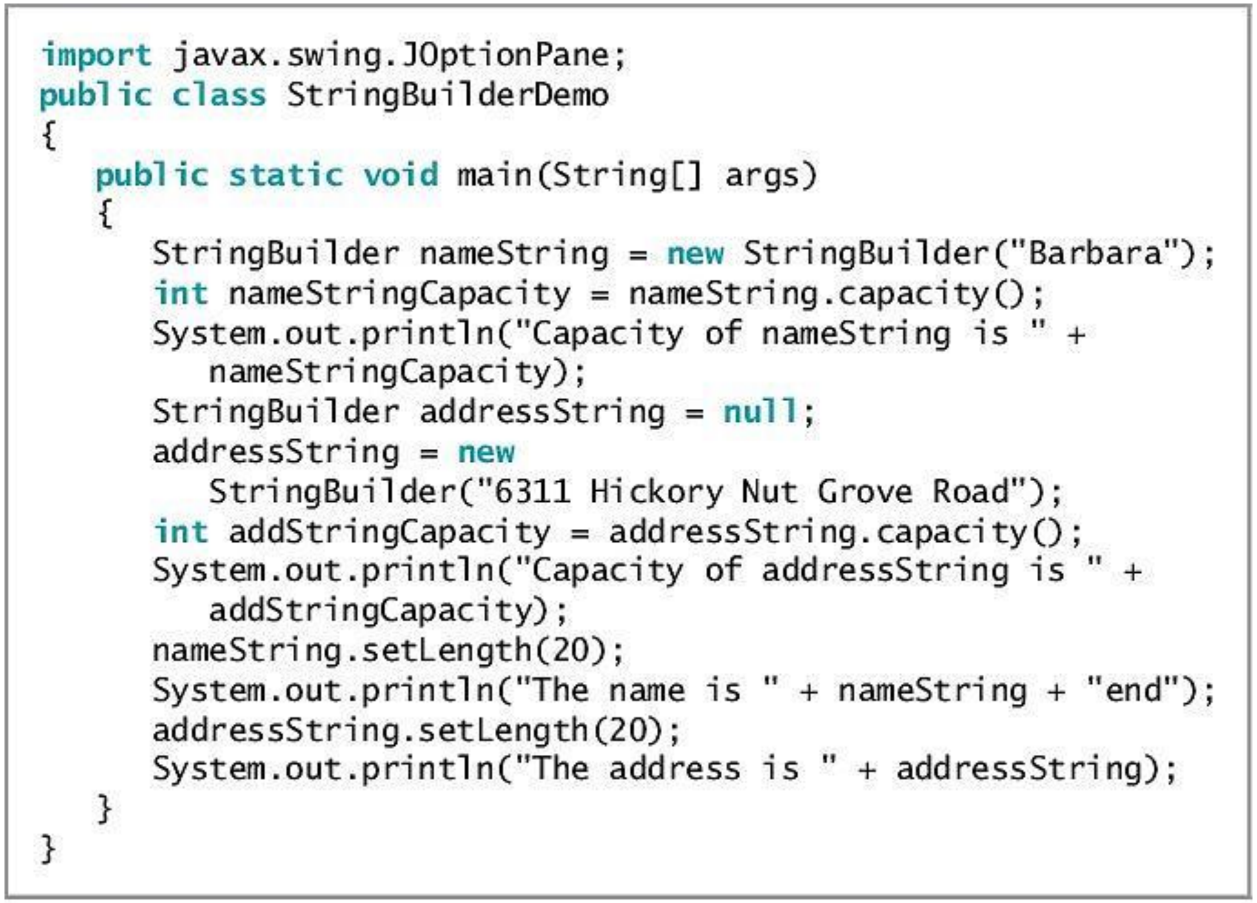
Learning About the `StringBuilder` and `StringBuffer` Classes (cont'd.)

- **Buffer**
  - A memory block
  - Might or might not contain a `String`
  - The `String` might not occupy the entire buffer
    - The length of a `String` can be different from the length of the buffer
  - **Capacity**
    - The actual length of the buffer

Learning About the `StringBuilder` and `StringBuffer` Classes (cont'd.)

- **`setLength()` method**
  - Changes the length of a `String` in a `StringBuilder` object
- `length` property
  - An attribute of the `StringBuilder` class
  - Identifies the number of characters in the `String` contained in the `StringBuilder`
- **`capacity()` method**
  - Finds the capacity of a `StringBuilder` object

Learning About the StringBuilder and StringBuffer Classes (cont'd.)



```
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 7-12 The StringBuilderDemo application

Learning About the `StringBuilder` and `StringBuffer` Classes (cont'd.)

- Using `StringBuilder` objects
  - Provides improved computer performance over `String` objects
  - Can insert or append new contents into `StringBuilder`
- `StringBuilder` constructors

```
public StringBuilder ()
public StringBuilder (int capacity)
public StringBuilder (String s)
```

Learning About the StringBuilder and StringBuffer Classes (cont'd.)

- **append() method**
  - Adds characters to the end of a StringBuilder object
- **insert() method**
  - Adds characters at a specific location within a StringBuilder object
- **setCharAt() method**
  - Changes a character at a specified position within a StringBuilder object

Learning About the `StringBuilder` and `StringBuffer` Classes (cont'd.)

- **`charAt()` method**
  - Accepts an argument that is the offset of the character position from the beginning of a `String`
  - Returns the character at that position

Learning About the StringBuilder and StringBuffer Classes (cont'd.)

```
import java.time.*;
public class ConcatenationTimeComparison
{
   public static void main(String[] args)
   {
      long startTime, endTime;
      final int TIMES = 200_000;
      final int FACTOR = 1_000_000;
      int x;
      StringBuilder string1 = new StringBuilder("");
      StringBuilder string2 = new StringBuilder(TIMES * 4);
      LocalDateTime now;
      now = LocalDateTime.now();
      startTime = now.getNano();
      for(x = 0; x < TIMES; ++x)
         string1.append("Java");
      now = LocalDateTime.now();
      endTime = now.getNano();
      System.out.println("Time with empty StringBuilder: " +
         ((endTime - startTime) / FACTOR + " milliseconds"));
      now = LocalDateTime.now();
      startTime = now.getNano();
      for(x = 0; x < TIMES; ++x)
         string2.append("Java");
      now = LocalDateTime.now();
      endTime = now.getNano();
      System.out.println("Time with empty StringBuilder: " +
         ((endTime - startTime) / FACTOR + " milliseconds"));
   }
}
```

Figure 7-14 The ConcatenationTimeComparison application

You Do It

- Testing Characters
- Examining the `String` Class at the Java Web Site
- Using `String` Methods
- Converting a `String` to an Integer
- Using `StringBuilder` Methods

Don’t Do It

- Don’t attempt to compare `String`s using the standard comparison operators
- Don’t forget that `startsWith()`, `endsWith()`, and `replace()` are case sensitive
- Don’t forget to use the `new` operator and the constructor when declaring initialized `StringBuilder` objects
- Don’t use `StringBuilder` or `StringBuffer` if the `String` class will work as well

Summary

- `String` variables
  - References
- `Character` class
  - Instances can hold a single character value
- Each `String` class object
  - Is immutable
  - `equals()` method
- `toString()` method
  - Converts any object to a `String`

Summary (cont'd.)

- `Integer.parseInt()` method
  - Takes a `String` argument and returns an integer value
- `Double.parseDouble()` method
  - Takes a `String` argument and returns a `double` value
- `StringBuilder` or `StringBuffer` class
  - Improves performance when a string's contents must change