1 **Object-Oriented Thinking**

CST141

2 **Interface vs. Implementation (Page 1)**

- The interface (the class documentation) consists of:
  - The name of the class and general description
  - A list of constructors and methods, as well as a description of the purpose of each constructor and method
  - The return values and parameters for constructors and methods
  - The constants and any other public fields

3 **Interface vs. Implementation (Page 2)**

- The interface does *not include* the class implementation:
  - The private data fields
  - Any public constructors and methods including the bodies (source code) for each method

4 **Documentation**

- "Document everything"
- *Write your comments first*
  - Before you write the method
  - If you do not know what to write, you probably do not understand fully what the method is supposed to do

5 **Writing Class Documentation**

- Your own classes can be documented the same way as are Java API library classes
  - Use your classes to *create* an interface, e.g. "library class"
- Others should be able to use your classes by reading the interface (documentation) without access to the implementation

6 **Elements of Documentation (Page 1)**

- Documentation for a *class* should include:
  - The class name (and inheritance hierarchy)
  - A comment describing the overall purpose, function, and characteristics of the class
  - A version number
  - The name of the author or authors

7 **Elements of Documentation (Page 2)**

- Documentation for *methods* (including constructors) and *public fields* (frequently constants) should include:
  - The method name (including constructors) or field name, as well as a comment describing the purpose and function of each
  - The parameter names and types, including a description
  - The return type, including a description

8 **The Javadoc Utility (Page 1)**

- Javadoc.exe is a standard, convenient tool to document Java code (part of Java JDK)
- Creates HTML files (web pages) that provide the ease of hyperlinks:
  – From one document to another …
  – As well as within each document

9 **The Javadoc Utility (Page 2)**

- Requires *special formatting* of comments
- The Javadoc utility reads the *formatted comments*, and automatically generates the HTML document based on those comments

10 **The Javadoc Utility (Page 3)**

- The two kinds of Javadoc comments:
  – Class-level comments—provides overall description of the classes
  – Member-level comments—describes the purpose(s) of the members (e.g. usually the methods and public fields)
- Both types of comments always start with the characters /** and end with */

11 **Class-Level Comments (Page 1)**

- Class-level comments provide an overall description of the class
- Placed just above *class header*
  – May not be followed by any other elements before the class header (e.g. import)
- Generally contain *author* and *version number* tags, and a description of the class

12 **Class-Level Comments (Page 2)**

- Example class-level comment:

```
/**
 * The Payee class calculates payroll
 * for regular and overtime workers.
 * Users update data fields by calling
 * the setHoursWorked() and setPayRate()
 * methods.
 *
 * @author Prof. Carl B. Struck
 * @version 1.0
*/
public class Payee
{
```

13 **Tags**

- Tags are formatting elements that start with ampersand (@) character and are formatted in the documentation by Javadoc.exe utility
- The @author tag describes the author(s)

```
@author Carl B. Struck
```

- The @version tag describes the version number or similar information
  @version 1.0

15 **Member-Level Comments (Page 1)**

- Member-level comments describe the public fields, methods, and constructors
- Placed directly above each *field* and/or *method header*

16 **Member-Level Comments (Page 2)**

- Member-level tags may include:
  – The @param tag which describes each of the method's required parameters
  – The @return tag describes the return value of a *non-void* method
  – The @throws tag describes exceptions which the method potentially throws (Chapter 14)

17 **Member-Level Comments (Page 3)**

- The @param tag describes each of the method's required parameters
  – There may be more than one @param for a method if it takes more than one parameter
  – First word always is the parameter variable name and it will be followed by a hyphen (-) in the generated documentation
  – Example:
    @param hoursWorked the employee number of hours worked

18 **Member-Level Comments (Page 4)**

- Member-level comment with a @param tag:

```
/**
 * Mutator method for the hours worked
 * data field. Validates that hours
 * worked is between 0.25 and 60.0.
 *
 * @param hoursWorked the employee number
 *                    of hours worked
 */
public void setHoursWorked(double hoursWorked)
{
```

20 **Member-Level Comments (Page 5)**

- The @return tag describes the return value of a *non-void* method
- Example:
  @return Employee number of hours worked as a double

21 **Member-Level Comments (Page 6)**

- A member-level comment with a @return tag:

```
/**
 * Accessor method for the hours worked
```

```
     * data field.
     *
     * @return Employee number of hours
     *         worked as a double
     */
    public double getHoursWorked()
    {
```

28 **Encapsulation (Page 1)**

- Encapsulation is achieved by making instance variables private
  - Also called "information hiding"
  - Only *what* a class can do should be visible to the outside, not *how* it does it

29 **Encapsulation (Page 2)**

- Through a public interface the private data can be used by the client class without *corrupting* that data
  - Only the class' own methods may directly inspect or manipulate its data fields
  - Protects data from the client but still allows the client to access the data
  - Makes the class easier to maintain since the functionality is managed *in just one place*
  -

30 **Encapsulation (Page 3)**

- Encapsulation is achieved by:
  - Making data fields (instance and static variables) private, and …
  - Having public *accessor* and *mutator* methods that give access to the data fields (of which the client does not know how they function)

32 **Instance vs. Static (Page 2)**

- A variable or method that is dependent on a specific instance of the class should be an instance variable or method
  - The *opposite* of static

31 **Instance vs. Static (Page 1)**

- A variable that is *shared* (one RAM location) by all instances of a class should be static
  - Static variables usually should be handled by static methods
  - Reference static members with the class name, e.g. JOptionPane.showMessageDialog()
  - Do not pass parameters for static variables to constructors which always are used to create an instance; rather include a static set method

33 **The JOptionPane Class (Page 1)**

- Class from the Java API library providing simple to use *popup dialogs* to prompt users for a value or to display information
- A member of the javax.swing class:

– import javax.swing.JOptionPane;

34 **The JOptionPane Class (Page 2)**

- JOptionPane class can seem complex, but most methods are one-line calls to one of the four (4) static show*Xxx*Dialog methods
- Two of the methods are:
  - showInputDialog—prompts for some input
  - showMessageDialog—a message that tells the user about something that has happened

35 **The JOptionPane Class (Page 3)**

- These two methods showInputDialog and showMessageDialog are static:
  public static void showMessageDialog(Component *parent*, Object *message*)
- The syntax to call these methods uses the *class name*, not an object name), e.g.
  JOptionPane.showMessageDialog(null, pay1.toString());

36 **The JOptionPane Class (Page 4)**

- JOptionPane method calls *pause* program execution (blocks the caller until the user's interaction is complete)
- The Java API documentation for the class JOptionPane is located on-line at:
  - http://docs.oracle.com/javase/7/docs/api/javax/swing/JOptionPane.html

37 **The showMessageDialog() Method (Page 1)**

- Displays output in a *message dialog* window
- The showMessageDialog is a method of the predefined JOptionPane class contained in the Java API library
- Alternative to println method which instead allows GUI (graphical user interface) output

38 **The showMessageDialog() Method (Page 2)**

- Takes two required parameters:
  - The first is the keyword null
  - The second is the output *message* (String, etc.)
- Format:
  JOptionPane.showMessageDialog(null,
  message);
- Example:
  JOptionPane.showMessageDialog(null, pay1.toString());

39 **The showInputDialog() Method (Page 1)**

- Accepts a String typed input from users in a textbox within the dialog window
- The showInputDialog is a member of the JOptionPane class
- Alternative to Scanner object which instead allows for GUI input

40 **The showInputDialog() Method (Page 2)**

- The only required argument is a message

– A *prompt* that tells the user what value should be keyed into the textbox
- The return value of the method is a String that is usually *assigned* to a variable

41 **The showInputDialog() Method (Page 3)**

- Format:
  JOptionPane.showInputDialog(*message*);
- Example:
  String input = JOptionPane.showInputDialog("Enter hours worked");

42 **Wrapper Classes (Page 1)**

- Primitive types (byte, short, int, long, float, double, boolean and char) are not objects
- Wrapper classes, which allow primitives to be *treated like* objects, exist for every primitive:
  – Byte, Short, Integer, Long, Float, Double, Boolean and Character
- Located in the java.lang package so they do not need to be imported

43 **Wrapper Classes (Page 2)**

- All Java wrapper classes (except Character) have *parse* methods that can convert String format of a number to numeric value:
  Byte.parseByte(*string*)
  Short.parseShort(*string*)
  Integer.parseInt(*string*)
  Long.parseLong(*string*)
  Float.parseFloat(*string*)
  Double.parseDouble(*string*)
  Boolean.parseBoolean(*string*)

44 **The Double.parseDouble Method**

- A method from wrapper class Double that converts String values to double type
  – May be necessary when an input method returns a String
- Format:
  double.parseDouble(*String*)
- Example:
  double hoursWorked = Double.parseDouble(stringHours);

45 **The Integer.parseInt Method**

- A method from wrapper class Integer that converts String values to int type
  – May be necessary when an input method (e.g. showInputDialog()) returns a String
- Format:
  Integer.parseInt(*string*)
- Example:
  int age = Integer.parseInt(stringAge);

46 **Return Values as Arguments to Another Method (Page 1)**

- When the return value (result) of one method will serve as an argument to the next

method ...
- Rather than storing the return value in a separate variable ...
- A common Java programmer practice is to insert the entire the first method call into the argument parentheses of the second method

47 **Return Values as Arguments to Another Method (Page 2)**

- Instead of:

```
String stringHours =
   JOptionPane.showInputDialog
      ("Enter hours worked");

double hoursWorked = Double.parseDouble(stringHours);
```

- Rather:

```
double hoursWorked = Double.parseDouble(
   JOptionPane.showInputDialog(
      "Enter hours worked") );
```

60 **char and String Variables**

- A char is a Java data type (a primitive numeric) that uses two bytes (16 bits) to store one text character ...
  - char literals enclosed in single quotes
  - E.g char anyLetter = 'L';
- A String (object or reference) is a series of characters treated as a unit ...
  - String literals enclosed in double quotes
  - E.g String firstName = "Charles";

61 **Character Representation**

- All characters (whether in a char or a String) are represented as a binary integer value between zero (0) and 65,535
- Requires two bytes (16 bits) of storage in RAM or on a disk ...
  - The highest *16 digit* binary number is 11111111 11111111 or 65,535
- The integer storage values are know as Unicode (formerly ANSI—one byte)

62 **The Unicode Table**

- Complete Unicode specification is found at:
  - http://www.ssec.wisc.edu/~tomw/java/unicode.html
  - The letter "A" is:
    - 65 in decimal
    - 0000 0000 0100 0001 in Unicode binary
  - The letter "a" is:
    - 97 in decimal
    - 0000 0000 0110 0001 in Unicode binary

63 **The String Class (Page 1)**

- String variables are reference variables (objects of the String class) …
  – "Points to" *multiple* locations in RAM
- The String class is located in the java.lang package so it *does not* need to be imported

64 **The String Class (Page 2)**

- Additionally String objects contain a series of methods used for manipulating them …
  – Java methods for processing strings include techniques for finding/comparing characters, extracting substrings, modifying upper/lower case, and many other methods

65 **Instantiating Strings (Page 1)**

- Java Strings may be *declared* using the same format as primitive variables (declares an un-instantiated object):
  – Format:
  String *stringVariable*;

66 **Instantiating Strings (Page 2)**

- Or a string may be *instantiated* formally using *object-oriented* notation with a constructor call:
  – Format:
  String *stringObject* = new String();

67 **Instantiating Strings (Page 3)**

- There are thirteen (13) *constructor* methods for instantiating String objects
- Example with no arguments:
  String middleName = new String();

68 **Instantiating Strings (Page 4)**

- Examples with String (either String constant or String variable) arguments:
  String lastName = new String("Jenson");
  – Equivalent of: String lastName = "Jenson";
  String lastName = new String(s1);
  – Equivalent of: String lastName = s1;
- Other String constructors accept char arrays, byte arrays, StringBuffers and StringBuilders

69 **The Scanner Class (Page 1)**

- A simple text scanner which is able to parse primitive types (int, double, boolean, etc.) and strings using regular expressions
- When the variable System.in (the "standard input stream") is passed to the constructor, reads text from the console, e.g the keyboard
- Found in the java.util package
  import java.util.Scanner;

70 **The Scanner Class (Page 2)**

- Format:

Scanner *scannerObject* = new Scanner(*inputStream*);

- Example:

  Scanner reader = new Scanner(System.in);

71 **The System.in Variable**

- The variable (field) "in" is a member of class System and is defined as the "standard input stream"
- Typically corresponds to *keyboard* input from the terminal (or sometimes another input source specified by the host environment or user)
- This stream is already open and ready to supply input data

72 **The nextLine() Method**

- A member of class Scanner that reads the next characters from the input device up to a carriage return as type String
  - Input device is the terminal keyboard if input stream for the Scanner object is System.in
- Format:

  scannerObject.nextLine()
- Example:

  String gender = reader.nextLine();

73 **The nextInt() Method**

- A member of class Scanner that reads and parses next characters from the input device up to a carriage return as type int
  - Input device is the terminal keyboard if input stream for the Scanner object is System.in
- Format:

  scannerObject.nextInt()
- Example:

  int inputAge = reader.nextInt();

74 **The nextDouble() Method**

- A member of class Scanner that reads and parses next characters from the input device up to a carriage return as type double
  - Input device is the terminal keyboard if input stream for the Scanner object is System.in
- Format:

  scannerObject.nextDouble()
- Example:

  double inputHours = reader.nextDouble();

76 **Immutable Strings (Page 1)**

- All object variables "reference" the object—the variable actually stores the *RAM address* where the object is located
- Strings are immutable objects—their contents *cannot be changed* after they are

instantiated

77 **Immutable Strings (Page 2)**

- When a string variable is updated, the variable references a new address where the new value is stored
  - The old string still is in RAM memory but can no longer be accessed

78 **Methods of the String Class**

- Used to perform manipulations with or upon a string or string variable
- Formats:
  stringVariable.method([*arg1*, *arg2*, ...])
  "*string*".*method*([*arg1*, *arg2*, ...])
- Some examples:
  int stringLength = s1.length();
  if (s1.equals("Java")) ...
  int indexLocation = "hello".indexOf(s5);
  String subStr1 = s1.substring(12);

79 **The equals() and equalsIgnoreCase() Methods (Page 1)**

- Two boolean methods of class String that compare their string objects to another string to see if they are identical
  - Returns value either true or false
- The equals() method is case sensitive
  - E.g. "H" does *not* equal "h"
- The equalsIgnoreCase() method ignores the *upper/lower case* of the letters compared, e.g. "H" *does* equal "h"

80 **The equals() and equalsIgnoreCase() Methods (Page 2)**

- Formats:
  stringObject.equals(*String*)
  stringObject.equalsIgnoreCase(*String*)
  - The *String* argument is the "*string*" or *stringVariable* to which the *stringObject* is compared
- Examples:
  if (s1.equals("Java")) ...
  - Equivalent but *invalid*: if (s1 == "Java")
  if (s2.equalsIgnoreCase(s3)) ...

81 **The equals() and equalsIgnoreCase() Methods (Page 3)**

- Why is it not possible to just use the "is equal to" operator (==) with Strings?
- String is a class and so Strings are *objects*
- When used with two objects the "is equal to" operator asks if the two objects are identical, that is do they share the same address in memory
- The following (compares addresses) really means "are these two objects the same String?":

```
if (s1 == "Java")
```

82 **String Comparison Processing**

- Made character by character, from *left* to *right*, in accordance with the computer's collating sequence
  - Unicode (ANSI, ASCII) , EBCDIC or some other code
- The binary value of the leftmost character of *one factor* is compared to the binary value of the leftmost character of *the other*
- If they are equal, the comparisons continue with each succeeding character position

83 **String Comparison Examples**

- Example 1:
  - "java"
    - Binary: 0110 1010 (106) / 0110 0001 (97) …
  - "jello"
    - Binary: 0110 1010 (106) / 0110 0101 (101) …
- Example 2:
  - "hello"
    - Binary: 0110 1000 (104) / 0110 0101 (101) …
  - "Hello"
    - Binary: 0100 1000 (72) / 0110 0101 (101) …

84 **Escape Sequences**

- Special character sequences within a string:
  - Begin with a backslash (\) and …
  - Modify the *format* of printed output
- Some common escape sequences:
  - \n New line (carriage return and line feed)
  - \t Tab
  - \\ Backslash (to print \ character)
  - \" Double quote (to print the " character)
  - \u*nnnn* A Unicode character as a hexadecimal value

86 **The compareTo() and compareToIgnoreCase() Methods (Page 1)**

- Methods of String class that compares the string object to another string to see if the object is:
  - *Equal* to string argument to which it is compared
  - *Greater* or *lesser* than the string argument to which it is compared
- Method compareTo() is *case sensitive*; method compareToIgnoreCase() is *not*

87 **The compareTo() and compareToIgnoreCase() Methods (Page 2)**

- The return value is an int as follows:
  - *Zero* (0) if the string object is equal to the "compare to" string argument
  - A *positive* integer if the string object is greater than the "compare to" string

argument
– A *negative* integer if the string object is less than the "compare to" string argument

88 **The compareTo() and compareToIgnoreCase() Methods (Page 3)**

- Formats:
  stringObject.compareTo(*string*)
  stringObject.compareToIgnoreCase(*string*)
  – The *string* argument is the "*string*" or *stringVariable* to which the *stringObject* is compared
- Examples:
  if (s1.compareTo("Java") > 0) …
  – Equivalent but *invalid*: if (s1 > "Java")
  if (s2.compareToIgnoreCase(s3) < 0) …

90 **The length() Method**

- A method of the String class that returns an int which is the count of the *number of characters* within a String
- Format:
  stringObject.length()
  – The length() method takes *no arguments*
- Examples:
  int stringLength1 = s1.length();
  int stringLength2 = "hello".length();
  – The second example returns the integer 5

91 **The charAt() Method (Page 1)**

- A method of class String that returns a char (*one character*) from a specific location within the string and *converts* it to a char
- Format:
  stringObject.charAt(*index*)
  – *index* is an int which is the position within the *stringObject* from where the character is returned

92 **The charAt() Method (Page 2)**

- Examples:
  char letter1 = s1.charAt(7);
  char letter2 = "hello".charAt(1);
  – The second example returns the character 'e'
- Using the charAt() method with an index value of zero (0) is the most common way to convert a String to a char, e.g.
  string.charAt(0);

94 **The substring() Method (Page 1)**

- Returns a String which is the subset of characters from within a string beginning at specified *start* location

- If an optional *end* location is designated, characters are returned only up to that location; otherwise …
- *All* characters to the end of the string object are returned

• Although the characters are returned, the *original* string object is *unchanged*

95 **The substring() Method (Page 2)**

• Format:

stringObject.substring(*beginIndex*[, *endIndex*])

- *beginIndex* is an int which is the location in *stringObject* where the subset of the returned characters *begins*
- *endIndex* (optional) is an int which is the location where the subset of returned characters *ends* (only those characters up to but not including it)

96 **The substring() Method (Page 3)**

• Examples:

String s2 = s1.substring(12);

– Returns all characters from index position 12 to end of string (the 13th character)

String s4 = s3.substring(12, 16);

– Returns all characters from index position 12 up to *excluding* index position 16

98 **The toLowerCase() Method (Page 1)**

• Returns a string with all the alphabetic characters in the string object converted to *lower case* …

- Effects *only* alphabetic characters

• Although the lower case characters are returned, the original string object is *unchanged*

99 **The toLowerCase() Method (Page 2)**

• Format:

stringObject.toLowerCase()

- There are *no arguments* to the method

• Example:

String s2 = s1.toLowerCase();

String s3 = "Hello".toLowerCase();

– Variable s3 will be assigned "hello"

100 **The toUpperCase() Method (Page 1)**

• Returns a string with all the alphabetic characters in the string object converted to *upper case*

- Effects *only* alphabetic characters

• Although the upper case characters are returned, the original string object is *unchanged*

101 **The toUpperCase() Method (Page 2)**

• Format:

stringObject.toUpperCase()
– There are *no arguments* to the method
- Example:
String s2 = s1.toUpperCase();
String s3 = "Hello".toUpperCase();
– Variable s3 will be assigned "HELLO"

103 **The split() Method (Page 1)**

- Splits a String object into tokens
  – Tokens are a series of substrings or a collection of string objects (like an array)
- For example:
  – In the string:
    • "Tokens are sets of characters"
  – The tokens are:
    • "Tokens", "are", "sets", "of", "characters"
  – Assuming that the blank space is the delimiter

104 **The split() Method (Page 2)**

- Format:
stringObject.split(*regEx*)
  – *regex* is a regular expression—the string which is the delimiter (separator) between the tokens
- Example:
String[] t1 = s1.split(" ");

106 **The StringBuilder Class (Page 1)**

- A class that provides functionality for building and concatenating strings into a single string
- StringBuilder class is located in the java.lang package (does *not* need to be imported)

107 **The StringBuilder Class (Page 2)**

- The primary methods of class StringBuilder are:
  – append—concatenates string (or some other type converted to String) *to the end* of the StringBuilder object
  – insert—inserts string (or some other data type converted to String) *within* the StringBuilder object

108 **The StringBuilder Constructor (Page 1)**

- There are four constructors including:
StringBuilder *stringBuilderObject* = new StringBuilder();
  – Creates a string builder object with a capacity of 16 elements (initially empty)
StringBuilder *stringBuilderObject* = new StringBuilder(*initialCapacity*);
  – Creates an empty string builder with a capacity specified by the int parameter *initialCapacity*

109 **The StringBuilder Constructor (Page 2)**

- There are four constructors including (con.):
  StringBuilder *stringBuilderObject* = new StringBuilder(*stringObject*);
  – Creates a string builder of the initial value of the specified *stringObject* plus 16 additional empty elements

110 **The length() Method**

- Like the String class, class StringBuilder has a length() method
- Returns an int which is the number of characters in the builder
- Format:
  stringBuilderObject.length()

111 **The capacity() Method**

- The capacity, which is an int returned by the capacity() method, is always greater than (usually) or equal to the length
- Automatically expands as necessary to accommodate additions to the string builder
- Format:
  stringBuilderObject.capacity()

112 **The append() Method (Page 1)**

- StringBuilder method that concatenates its argument to the end of string builder object
- The data is converted to a string before the append operation takes place
  – Therefore the argument *type* may be String or any of the following:
    • boolean, char, char[] (array), float, double, short, int, long, or Object

113 **The append() Method (Page 2)**

- Format:
  stringBuilderObject.append(*argument*);
- Examples:
  output.append("The char is " + c1);
  --or—
  output.append("The char is ");
  output.append(c1);

115 **The delete() Method**

- StringBuilder method that deletes subsequence from start to end - 1 (inclusive) in the string builder's char sequence
- Format:
  stringBuilderObject.delete(*start*, *end*);
- Example:
  output.delete(12, output.length());
  – This example deletes from the 13th char to the end of the string builder object

116 **The deleteCharAt() Method**

- StringBuilder method that deletes the char located at index in string builder object
- Format:
  stringBuilderObject.deleteCharAt(*index*);
- Example:
  output.deleteCharAt(0);
  – This example deletes the 1st char of the string builder object

117 **The insert() Method (Page 1)**

- StringBuilder method of that inserts the second argument into string builder object
- The first int argument indicates the index before which the data is to be inserted
- Like append(), the data is converted to a string before the insert operation takes place
  – Therefore the argument *type* may be String, but also may be boolean, char, char[] (char array), float, double, short, int, long, or Object

118 **The insert() Method (Page 2)**

- Format:
  stringBuilderObject.insert(*index*, *object*);
  – The *object* could be a primitive (int, double, char, etc.), a String or any other object
- Example:
  output.insert(6, "Hello");
  – This example inserts the string "Hello" before the 7th character of the string builder object

119 **The replace() Method**

- StringBuilder method that replaces the specified characters in string builder object
- Format:
  stringBuilderObject.replace(*start*, *end*, *stringObject*);
- Example:
  output.replace(2, 4, "Hello");
  – This example replaces the 3rd through the 5th char's of the string builder object with the string "Hello"

120 **The reverse() Method**

- StringBuilder method that reverses sequence of characters in the string builder object
- Format:
  stringBuilderObject.reverse();

121 **The setCharAt() Method**

- StringBuilder method that replaces a single character in the string builder object
- Format:
  stringBuilderObject.setCharAt(*index*, *char*);
- Example:
  output.setCharAt(8, 'G');

– This example replaces the 9th char of the string builder object with the character 'G'

122 **The toString() Method (Page 1)**

- StringBuilder has a toString() method that overrides that of Object and returns a string representation of the object
  – Effectively the character sequence within the string builder object

123 **The toString() Method (Page 2)**

- Format:
  stringBuilderObject.toString()
- Examples:
  String s2 = output.toString();
  – Return type of method is String
  System.out.println(output.toString());
  JOptionPane.showMessageDialog(null, output);

124 **The StringBuffer Class**

- The StringBuffer class is similar to StringBuilder and often the two can be used interchangeably
  – The methods of both allow strings and other characters to be added, inserted and/or appended into their objects
- Use StringBuffer if the object might be accessed by *multiple tasks* concurrently; otherwise use StringBuilder