Object-Oriented Thinking (Part 2)

Introduction to Programming and Computational Problem Solving - 2
CSE 8B
Lecture 9
Announcements

• Assignment 4 is due Nov 4, 11:59 PM
• Quiz 4 is Nov 6
• Assignment 5 will be released Nov 4
  – Due Nov 12, 11:59 PM
• Educational research study
  – Nov 6, weekly reflection
• Reading
  – Chapter 10
Object-oriented thinking

• The advantages of object-oriented programming over procedural programming
• Classes provide more flexibility and modularity for building reusable software
• How to solve problems using the object-oriented paradigm
• Class design
Thinking in objects

• Procedural programming focuses on designing methods

• Object-oriented programming
  – Couples data and methods together into objects
  – Focuses on designing objects and operations on objects

• Object-orientated programming combines the power of procedural programming with an additional component that integrates data with operations into objects
Procedural programming vs object-oriented programming

• Procedural programming
  – Data and operations on data are separate
  – Requires passing data to methods

• Object-oriented programming
  – Data and operations on data are in an object
  – Organizes programs like the real world
    • All objects are associated with both attributes and activities
  – Using objects improves software reusability and makes programs easier to both develop and maintain
Class abstraction and encapsulation

• A class is designed for use by many different users (or customers or clients)
• To be useful in a wide range of applications, a class should provide a variety of ways for customization through properties, and constructors and methods
Class design and development

• Use a UML class diagram to design the class
• Write a test program that uses the class
  – Developing a class and using a class are two separate tasks
  – It is easier to implement a class if you must use the class
• Implement the class
• Use Javadoc to document the class (contract)
Primitive data type values as objects

• A primitive data type is not an object
• But it can be wrapped in an object using a Java API wrapper class
  
  Boolean
  Character
  Short
  Byte
  Integer
  Long
  Float
  Double

Notes
• The wrapper classes do not have no-arg constructors
• The instances of all wrapper classes are immutable (i.e., their internal values cannot be changed once the objects are created)
# Integer and Double wrapper classes

<table>
<thead>
<tr>
<th>java.lang.Integer</th>
<th>java.lang.Double</th>
</tr>
</thead>
<tbody>
<tr>
<td>-value: int</td>
<td>-value: double</td>
</tr>
<tr>
<td>+MAX_VALUE: int</td>
<td>+MAX_VALUE: double</td>
</tr>
<tr>
<td>+MIN_VALUE: int</td>
<td>+MIN_VALUE: double</td>
</tr>
<tr>
<td>+Integer(value: int)</td>
<td>+Double(value: double)</td>
</tr>
<tr>
<td>+Integer(s: String)</td>
<td>+Double(s: String)</td>
</tr>
<tr>
<td>+byteValue(): byte</td>
<td>+byteValue(): byte</td>
</tr>
<tr>
<td>+shortValue(): short</td>
<td>+shortValue(): short</td>
</tr>
<tr>
<td>+intValue(): int</td>
<td>+intValue(): int</td>
</tr>
<tr>
<td>+longValue(): long</td>
<td>+longValue(): long</td>
</tr>
<tr>
<td>+floatValue(): float</td>
<td>+floatValue(): float</td>
</tr>
<tr>
<td>+doubleValue():double</td>
<td>+doubleValue():double</td>
</tr>
<tr>
<td>+compareTo(o: Integer): int</td>
<td>+compareTo(o: Double): int</td>
</tr>
<tr>
<td>+toString(): String</td>
<td>+toString(): String</td>
</tr>
<tr>
<td>+valueOf(s: String): Integer</td>
<td>+valueOf(s: String): Double</td>
</tr>
<tr>
<td>+valueOf(s: String, radix: int): Integer</td>
<td>+valueOf(s: String, radix: int): Double</td>
</tr>
<tr>
<td>+parseInt(s: String): int</td>
<td>+parseInt(s: String): int</td>
</tr>
<tr>
<td>+parseInt(s: String, radix: int): int</td>
<td>+parseInt(s: String, radix: int): int</td>
</tr>
<tr>
<td>+parseDouble(s: String): double</td>
<td>+parseDouble(s: String): double</td>
</tr>
<tr>
<td>+parseDouble(s: String, radix: int): double</td>
<td>+parseDouble(s: String, radix: int): double</td>
</tr>
</tbody>
</table>
Wrapper classes

• Constructors
• Class Constants MAX_VALUE and MIN_VALUE
• Conversion Methods
Numeric wrapper class constructors

• You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value
  – For example, the constructors for Integer and Double are
    
    public Integer(int value)
    public Integer(String s)
    public Double(double value)
    public Double(String s)
Numeric wrapper class constants

• Each numerical wrapper class has the constants `MAX_VALUE` and `MIN_VALUE`

• `MAX_VALUE` represents the maximum value of the corresponding primitive data type

• For `Byte`, `Short`, `Integer`, and `Long`, `MIN_VALUE` represents the minimum byte, short, int, and long values

• For `Float` and `Double`, `MIN_VALUE` represents the minimum *positive* float and double values
Numeric wrapper class conversion methods

• Each numeric wrapper class implements the abstract methods `doubleValue`, `floatValue`, `intValue`, `longValue`, and `shortValue`
  – Defined in the abstract `Number` class (covered in three weeks)

• These methods “convert” objects into primitive type values
Numeric wrapper class
static valueOf methods

• The numeric wrapper classes have a useful class method valueOf(String s)

• This method creates a new object initialized to the value represented by the specified string
  – For example
    ```java
    Double doubleObject = Double.valueOf("12.4");
    Integer integerObject = Integer.valueOf("12");
    ```
Numeric wrapper class static parsing methods

• Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value based on 10 or any specified radix (e.g., 2 for binary, 8 for octal, 10 for decimal, 16 for hexadecimal)

  – For example

    \[
    \begin{align*}
    \text{Integer.parseInt("13")} & \text{ returns 13} \\
    \text{Integer.parseInt("13", 10)} & \text{ returns 13} \\
    \text{Integer.parseInt("1A", 16)} & \text{ returns 26}
    \end{align*}
    \]
Automatic conversion between primitive types and wrapper class types

- Converting a primitive value to a wrapper object is called *boxing*.
- Converting a wrapper object to a primitive value is called *unboxing*.
- The Java compiler will automatically convert a primitive data type value to an object using a wrapper class (*autoboxing*) and vice versa (*autounboxing*), depending on the context.
Automatic conversion between primitive types and wrapper class types

```java
Integer[] intArray = {new Integer(2), new Integer(4), new Integer(3)};  // (a)

// Equivalent
Integer[] intArray = {2, 4, 3};  // (b)
```

Autoboxing

```java
Integer[] intArray = {1, 2, 3};
System.out.println(intArray[0] + intArray[1] + intArray[2]);
```

Autounboxing
BigInteger and BigDecimal classes

• If you need to compute with very large integers or high precision floating-point values, you can use the BigInteger and BigDecimal classes in the java.math package
• Both are immutable
• Both extend the Number class and implement the Comparable interface (covered in three weeks)
BigInteger and BigDecimal classes

BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);

BigDecimal a = new BigDecimal(1.0);
BigDecimal b = new BigDecimal(3);
BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);
System.out.println(c);
String class

• The String class has 13 constructors and more than 40 methods

• A good example for learning classes and objects
Constructing strings

• Create from a string literal
  – Syntax
    ```java
    String newString = new String(stringLiteral);
    ```
  – Example
    ```java
    String message = new String("Welcome to Java");
    ```
  – Since strings are used frequently, Java provides a shorthand initializer for creating a string
    ```java
    String message = "Welcome to Java";
    ```

• Create from an array of characters
  – Syntax
    ```java
    String newString = new String(charArray);
    ```
    • where, for example
      ```java
      char[] charArray = { 'C', 'S', 'E', ' ', '8', 'B'};
      ```
Strings are immutable

• A String object is immutable (i.e., its contents cannot be changed once the string is created)

• The following code does not change the contents of the string

  String s = "Java";
  s = "HTML";
Strings are immutable

```java
String s = "Java";
s = "HTML";
```

After executing `String s = "Java";`

After executing `s = "HTML";`

This string object is now unreferenced
Strings are immutable

String s = "Java";

\[s = "HTML";\]

After executing String s = "Java";

\[s\]

: String

String object for "Java"

Contents cannot be changed

After executing s = "HTML";

\[s\]

: String

String object for "Java"

This string object is now unreferenced

String object for "HTML"
Interned strings

• Since strings are immutable and are frequently used, to improve efficiency and save memory, the JVM uses a unique instance for string literals with the same character sequence

• Such an instance is called *interned*
Interned strings

- A new object is created if you use the new operator
- If you use the string initializer, no new object is created if the interned object is already created

```java
String s1 = "Welcome to Java";
String s2 = new String("Welcome to Java");
String s3 = "Welcome to Java";

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

`s1` is `false`
`s1` is `true`
## Replacing and splitting strings

<table>
<thead>
<tr>
<th>java.lang.String</th>
</tr>
</thead>
<tbody>
<tr>
<td>+replace(oldChar: char, newChar: char): String</td>
</tr>
<tr>
<td>+replaceFirst(oldString: String, newString: String): String</td>
</tr>
<tr>
<td>+replaceAll(oldString: String, newString: String): String</td>
</tr>
<tr>
<td>+split(delimiter: String): String[]</td>
</tr>
</tbody>
</table>

Returns a new string that replaces all matching character in this string with the new character.

Returns a new string that replaces the first matching substring in this string with the new substring.

Returns a new string that replace all matching substrings in this string with the new substring.

Returns an array of strings consisting of the substrings split by the delimiter.
Replacing a string

- "Welcome".replace('e', 'A') returns a new string WAlcomA
- "Welcome".replaceFirst("e", "AB") returns a new string WABlcome
- "Welcome".replace("e", "AB") returns a new string WABlcomAB
- "Welcome".replace("el", "AB") returns a new string WABcome
Splitting a string

• Split a string into an array of strings
  – For example, using # as a delimiter
    ```java
    String[] tokens = "CSE#8B#uses#Java".split("#", 0);
    for (int i = 0; i < tokens.length; i++)
      System.out.print(tokens[i] + " ");
    ```
  – Displays CSE 8B uses Java
Matching, replacing, and splitting by patterns

• You can match, replace, or split a string by specifying a pattern
  – For example
    "Java".equals("Java");
    "Java".matches("Java");

• This is an extremely useful and powerful feature known as *regular expression* (see appendix H)
Convert character and numbers to strings

• The String class provides several static `valueOf` methods for converting a character, an array of characters, and numeric values to strings.

• These methods have the same name `valueOf` with different argument types `char`, `char[]`, `double`, `long`, `int`, and `float`.
  
  – For example, to convert a `double` value to a string, use `String.valueOf(5.44)`.
  
  • The return value is a string consisting of characters '5', '.', '4', and '4'.
StringBuilder and StringBuffer classes

• The StringBuilder and StringBuffer classes are alternatives to the String class

• In general, a StringBuilder or StringBuffer can be used wherever a string is used

• StringBuilder and StringBuffer are more flexible than String

• You can add, insert, or append new contents into a string buffer, whereas the value of a String object is fixed once the string is created
StringBuilder constructors

<table>
<thead>
<tr>
<th>java.lang.StringBuilder</th>
</tr>
</thead>
<tbody>
<tr>
<td>+StringBuilder()</td>
</tr>
<tr>
<td>+StringBuilder(capacity: int)</td>
</tr>
<tr>
<td>+StringBuilder(s: String)</td>
</tr>
</tbody>
</table>

Constructs an empty string builder with capacity 16.
Constructs a string builder with the specified capacity.
Constructs a string builder with the specified string.
Modifying strings in the builder

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>append(data: char[]): StringBuilder</td>
<td>Appends a char array into this string builder.</td>
</tr>
<tr>
<td>append(data: char[], offset: int, len: int): StringBuilder</td>
<td>Appends a subarray in data into this string builder.</td>
</tr>
<tr>
<td>append(v: aPrimitiveType): StringBuilder</td>
<td>Appends a primitive type value as a string to this builder.</td>
</tr>
<tr>
<td>append(s: String): StringBuilder</td>
<td>Appends a string to this string builder.</td>
</tr>
<tr>
<td>delete(startIndex: int, endIndex: int): StringBuilder</td>
<td>Deletes characters from startIndex to endIndex.</td>
</tr>
<tr>
<td>deleteCharAt(index: int): StringBuilder</td>
<td>Deletes a character at the specified index.</td>
</tr>
<tr>
<td>insert(index: int, data: char[], offset: int, len: int): StringBuilder</td>
<td>Inserts a subarray of the data in the array to the builder at the specified index.</td>
</tr>
<tr>
<td>insert(offset: int, data: char[]): StringBuilder</td>
<td>Inserts data into this builder at the position offset.</td>
</tr>
<tr>
<td>insert(offset: int, b: aPrimitiveType): StringBuilder</td>
<td>Inserts a value converted to a string into this builder.</td>
</tr>
<tr>
<td>insert(offset: int, s: String): StringBuilder</td>
<td>Inserts a string into this builder at the position offset.</td>
</tr>
<tr>
<td>replace(startIndex: int, endIndex: int, s: String): StringBuilder</td>
<td>Replaces the characters in this builder from startIndex to endIndex with the specified string.</td>
</tr>
<tr>
<td>reverse(): StringBuilder</td>
<td>Reverses the characters in the builder.</td>
</tr>
<tr>
<td>setCharAt(index: int, ch: char): void</td>
<td>Sets a new character at the specified index in this builder.</td>
</tr>
</tbody>
</table>
# The `toString`, `capacity`, `length`, `setLength`, and `charAt` methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>toString()</code></td>
<td>Returns a string object from the string builder.</td>
</tr>
<tr>
<td><code>capacity()</code></td>
<td>Returns the capacity of this string builder.</td>
</tr>
<tr>
<td><code>charAt(index: int): char</code></td>
<td>Returns the character at the specified index.</td>
</tr>
<tr>
<td><code>length()</code></td>
<td>Returns the number of characters in this builder.</td>
</tr>
<tr>
<td><code>setLength(newLength: int): void</code></td>
<td>Sets a new length in this builder.</td>
</tr>
<tr>
<td><code>substring(startIndex: int): String</code></td>
<td>Returns a substring starting at startIndex.</td>
</tr>
<tr>
<td><code>substring(startIndex: int, endIndex: int): String</code></td>
<td>Returns a substring from startIndex to endIndex-1.</td>
</tr>
<tr>
<td><code>trimToSize()</code></td>
<td>Reduces the storage size used for the string builder.</td>
</tr>
</tbody>
</table>
Next Lecture

• Inheritance and polymorphism
• Reading
  – Chapter 11