

Object-Oriented Thinking (Part 2)

Introduction to Programming and
Computational Problem Solving - 2

CSE 8B

Lecture 9

Announcements

- Assignment 4 is due Oct 27, 11:59 PM
- Quiz 4 is Oct 29
- Assignment 5 will be released Oct 27
 - Due Nov 3, 11:59 PM
- Educational research study
 - Oct 29, weekly survey
- Reading
 - Liang
 - 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

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

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

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 that, together, are **minimal and complete**

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)

Object-oriented thinking

- Classes provide more flexibility and modularity for building reusable software
- Class abstraction and encapsulation
 - Separate class implementation from the use of the class
 - The creator of the class provides a description of the class and let the user know how the class can be used
 - The user of the class does not need to know how the class is implemented
 - The detail of implementation is encapsulated and hidden from the user

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

java.lang.Integer

```
-value: int
+MAX VALUE: int
+MIN VALUE: int

+Integer(value: int)
+Integer(s: String)
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longVlaue(): long
+floatValue(): float
+doubleValue():double
+compareTo(o: Integer): int
+toString(): String
+valueOf(s: String): Integer
+valueOf(s: String, radix: int): Integer
+parseInt(s: String): int
+parseInt(s: String, radix: int): int
```

java.lang.Double

```
-value: double
+MAX VALUE: double
+MIN VALUE: double

+Double(value: double)
+Double(s: String)
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longVlaue(): long
+floatValue(): float
+doubleValue():double
+compareTo(o: Double): int
+toString(): String
+valueOf(s: String): Double
+valueOf(s: String, radix: int): Double
+parseDouble(s: String): double
+parseDouble(s: String, radix: int): double
```

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

```
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

`Integer.parseInt("13")` returns 13

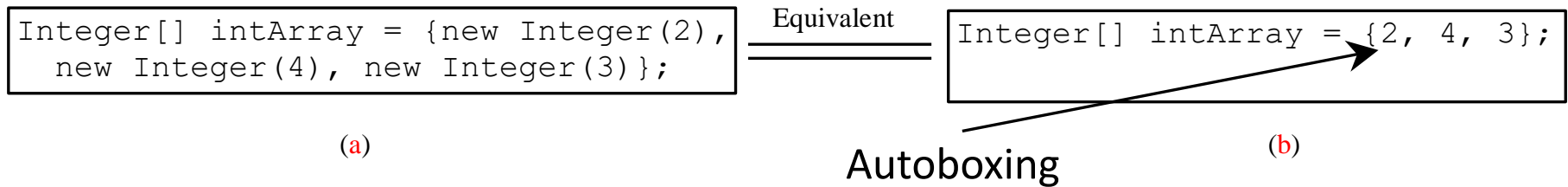
`Integer.parseInt("13", 10)` returns 13

`Integer.parseInt("1A", 16)` returns 26

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



```
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

```
String newString = new String(stringLiteral);
```
 - Example

```
String message = new String("Welcome to Java");
```
 - Since strings are used frequently, Java provides a shorthand initializer for creating a string

```
String message = "Welcome to Java";
```
- Create from an array of characters
 - Syntax

```
String newString = new String(charArray);
```

 - where, for example

```
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

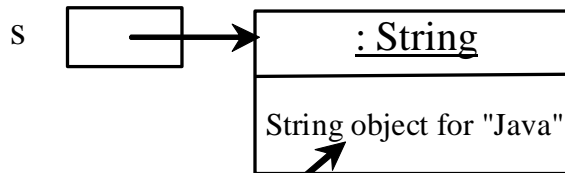
```
String s = "Java";  
s = "HTML";
```

Strings are immutable

```
String s = "Java";
```

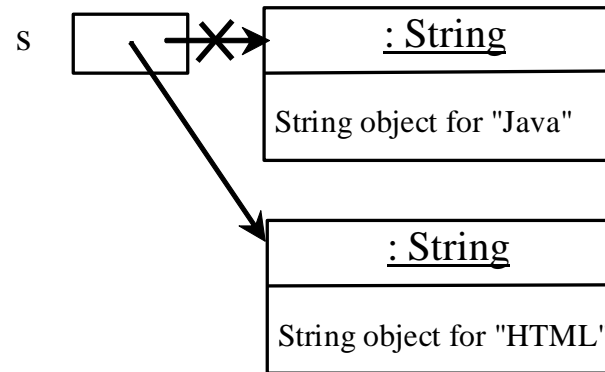
```
s = "HTML";
```

After executing `String s = "Java";`



Contents cannot be changed

After executing `s = "HTML";`



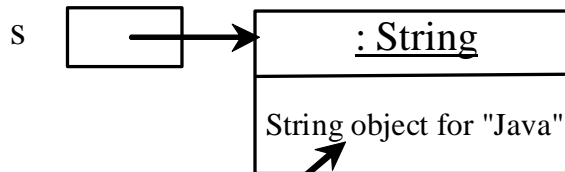
This string object is now unreferenced

Strings are immutable

```
String s = "Java";
```

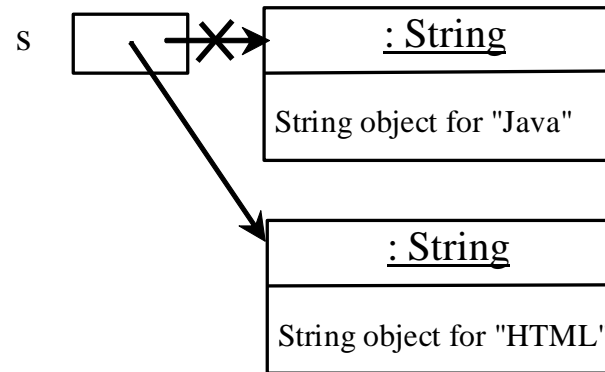
```
s = "HTML";
```

After executing `String s = "Java";`



Contents cannot be changed

After executing `s = "HTML";`



This string object is now unreferenced

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

```
String s1 = "Welcome to Java";
```

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

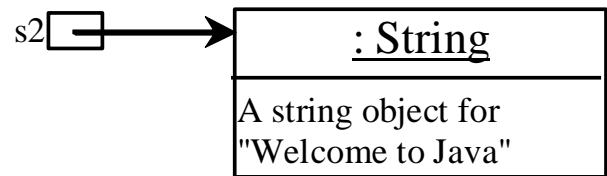
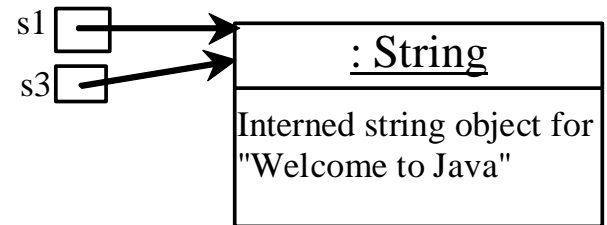
```
String s3 = "Welcome to Java";
```

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

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

s1 == s2 is false

s1 == s3 is true



Replacing and splitting strings

java.lang.String
+replace(oldChar: char, newChar: char): String
+replaceFirst(oldString: String, newString: String): String
+replaceAll(oldString: String, newString: String): String
+split(delimiter: String): String[]

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

```
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*
 - Liang, appendix H
 - <https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html#sum>

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 string consists 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

java.lang.StringBuilder

+StringBuilder()

Constructs an empty string builder with capacity 16.

+StringBuilder(capacity: int)

Constructs a string builder with the specified capacity.

+StringBuilder(s: String)

Constructs a string builder with the specified string.

Modifying strings in the builder

java.lang.StringBuilder	
+append(data: char[]): StringBuilder	Appends a char array into this string builder.
+append(data: char[], offset: int, len: int): StringBuilder	Appends a subarray in data into this string builder.
+append(v: <i>aPrimitiveType</i>): StringBuilder	Appends a primitive type value as a string to this builder.
+append(s: String): StringBuilder	Appends a string to this string builder.
+delete(startIndex: int, endIndex: int): StringBuilder	Deletes characters from startIndex to endIndex.
+deleteCharAt(index: int): StringBuilder	Deletes a character at the specified index.
+insert(index: int, data: char[], offset: int, len: int): StringBuilder	Inserts a subarray of the data in the array to the builder at the specified index.
+insert(offset: int, data: char[]): StringBuilder	Inserts data into this builder at the position offset.
+insert(offset: int, b: <i>aPrimitiveType</i>): StringBuilder	Inserts a value converted to a string into this builder.
+insert(offset: int, s: String): StringBuilder	Inserts a string into this builder at the position offset.
+replace(startIndex: int, endIndex: int, s: String): StringBuilder	Replaces the characters in this builder from startIndex to endIndex with the specified string.
+reverse(): StringBuilder	Reverses the characters in the builder.
+setCharAt(index: int, ch: char): void	Sets a new character at the specified index in this builder.

The toString, capacity, length, setLength, and charAt methods

java.lang.StringBuilder
+toString(): String
+capacity(): int
+charAt(index: int): char
+length(): int
+setLength(newLength: int): void
+substring(startIndex: int): String
+substring(startIndex: int, endIndex: int): String
+trimToSize(): void

Returns a string object from the string builder.

Returns the capacity of this string builder.

Returns the character at the specified index.

Returns the number of characters in this builder.

Sets a new length in this builder.

Returns a substring starting at startIndex.

Returns a substring from startIndex to endIndex-1.

Reduces the storage size used for the string builder.

Next Lecture

- Inheritance
- Reading
 - Liang
 - Chapter 11