CMSC131

Wrappers, Boxing and Unboxing,
Object (Im)mutability

Wrapper Classes

For reasons that we will explore more soon, Java provides an object-based wrapper class for each primitive datatype. Java will “box” or “unbox” primitives to/from wrapper-based objects as needed without an explicit request.

– The way these classes work, once an object of one of these types is created, the values stored in them cannot be altered. We call this type of object an **immutable object**.

– Recall that the ***String*** class also creates immutable objects, and what that meant.

Primitive wrapper class details…

In Java, the wrapper classes for primitive types are fairly easy to remember:

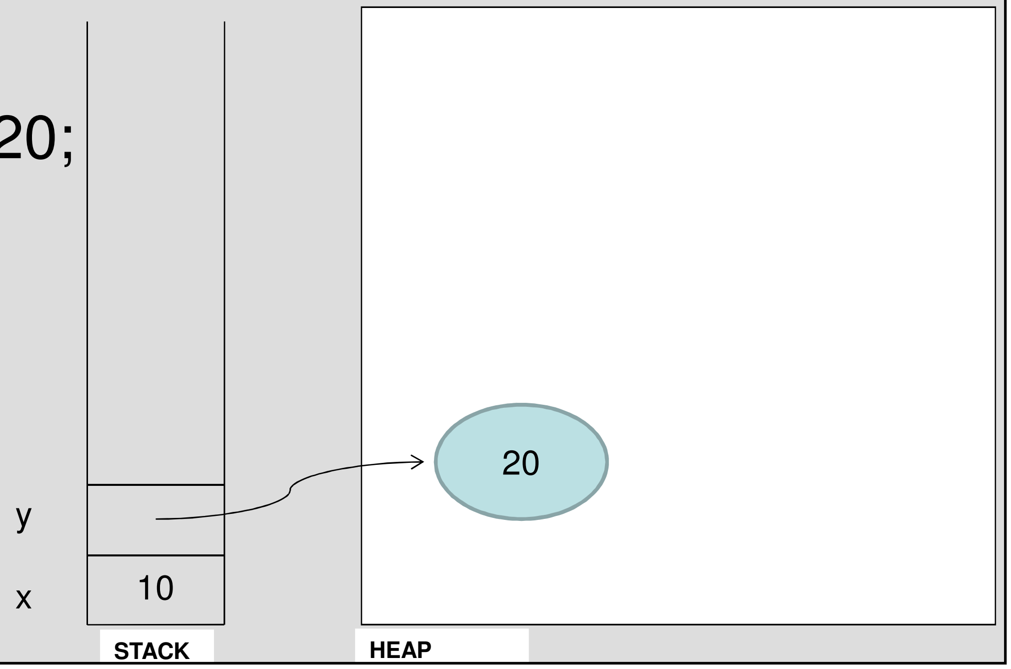
Primitive	Wrapper
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Java provides other classes called "wrappers" that are not based on primitives, but when we talk about wrappers in this class, these are what we mean.

An int and an Integer in memory.

int x = 10;

Integer y = 20;



`Integer` and `String` objects are *immutable*

In object oriented languages, some object types are **immutable** or (as we'll discuss later) *effectively* immutable. Integer (and all wrappers) and String are examples.

These are data types where once an object is created, it is designed to never have its data contents altered. A copy based upon such an object with some differences in the data can be made (as we have seen with methods such as **toUpperCase**, **concat**, **substring**) but the original object is untouched.

What about `Integer` objects?

The following two lines of code accomplish the same thing, similar to how Java automatically created String objects for us:

```
Integer value1 = 8;
Integer value1 = new Integer(8);
```

The following line of code will cause `value1` to refer to an Integer object that contains 9:

```
value1++;
```

However, it's NOT the same object as before!

```
Integer value1 = 8;
Integer value2 = value1;
value1++; //This will unbox the int,
                //  increment it, and rebox it.
System.out.println(value1 + " " + value2);
```

What will the output be?

```
Integer value1 = 8;
Integer value2 = value1;
value1++;
System.out.println(
   value1 + " " + value2
);
```

Immutable Objects

With immutable objects, once the object is created, the values it holds cannot be altered.

With an effectively immutable object, the values it holds are not meant to be altered and no explicit means are provides to alter them, but they aren't 100% protected again certain advanced things.

There are two big differences between immutable and *effectively* immutable in Java.

– true immutable object types need all data fields to be ***declared*** as `final`

– there are rules regarding the way the constructors work that need to be followed regarding them being thread-safe

Mutable Objects

Object types where information is publicly available (and thus can change) or which provide public setters are called mutable.

Different languages approach mutability in different ways, so it is important to explore the conventions of each language you learn.

- In Objective-C there are usually both mutable and immutable versions of types.
- Though not an object, it's interesting to note that in Fortran, even literals like 1 weren't always immutable!

What if we *want* mutable strings?

The **`String`** class in Java is immutable.

The **`StringBuffer`** class in Java is mutable.

Let's look at an example: **`StringHolder.java`**

It's important to note that unlike classes such as **String** and **Integer**, with **StringBuffer** we must explicitly create a new object and then send any initial information to the constructor.

The `StringBuffer` Class

Some key methods we can use are:

- append, which is overloaded in many ways
- insert, which is also overloaded in many ways
- delete, which allows you to delete any sub-part of the string
- replace, which allows you to replace any sub-part of the string with another string

These methods also return a *reference* to the **`StringBuffer`** being modified.

http://download.oracle.com/javase/7/docs/api/java/lang/StringBuffer.html

Mutability: Good or Bad?

While a significant issue, some might argue it's not "too bad" since aliasing of mutable objects can be "solved" by making deep copies when needed.

Are there any reasons why mutable objects would actually be good in their own right?

Consider the posted example:

`StringEfficiencyExample`

What are the trade-offs?

What are the trade-offs involved in the previous example?

- Time?
- Space?
- Other?

Let’s create a Data Object!

We will now begin to explore how to create a class that will represent a data type. We will be able to create objects based on this class.

For a start, we will want it to be able to have a getter (method to retrieve the stored value) and a setter (method to store a new value in the object) as well as useful things such as a way to increment the value by 1 and a way to turn the value into a String for printing.

MutableInteger.java

```
public class MutableInteger {
  private int value;  //Note this is not static field.

  public MutableInteger(int value_in) {
     set(value_in); //Since we have a setter, we use it.
  }

  public int get() {return value;}
  public void set(int value_in) {value=value_in;}
  public int plusplus() {value++; return value;}
  //We cannot overload an operator in Java, but this recreates the post++ behavior

  public String toString() {
     return Integer.toString(value);
       //We can use the static toString method of the
       //  Integer class to convert an int to a String
  }
}
```

Java Class: Color

There is an immutable class type called **Color** in the Java AWT library.

- You can use references to pre-made **Color** objects via static constant fields like **RED** and **BLUE**.
- You can create a new **Color** object using a specific combination of shades of red, blue, and green.

Mutable Class: Grid

In an upcoming lab and upcoming project, you'll be using some classes we created here at UMD that stored things in a **Grid**.

- When you create an object of this type, it will contain many instance fields. Among them will be a two-dimensional data structure that can hold **Color** objects.
- In the lab and project you will be implementing static helper methods that are passed a reference to a **Grid** object, and then "painting" images into it by setting row/col positions inside the grid to certain colors.