WRAPPER CLASSES, AUTO-BOXING AND AUTO-UNBOXING

Wrapper classes

The wrapper classes provide a mechanism to "wrap" primitive values in an object so that the primitives can be included in activities only for objects, like being added to Collections. There is a wrapper class for every primitive in Java. The wrapper class for int is Integer and the class for float is Float and so on. Wrapper classes also provide many utility functions for primitives like Integer.parseInt().

Use case: Write a method which can accept anything. You can write the method as public method(Object obj){}. Now since all classes are children of Object class, their objects can be passed. However primitive types are not children of Object and hence cannot be passed. We can wrap the primitives inside corresponding wrapper classes and then pass to the function.

Three most common approaches for creating wrapper objects are:

1. Using constructor as

Integer i1 = new Integer(42);

All of the wrapper classes except Character provide two constructors: one that takes a primitive type and one that takes a String representation of the primitive type: new Integer(42) or new Integer("42"). The Character class provides only one constructor, which takes a char as an argument: new Character('c').

2. Using valueOf() as

Float f2 = Float.valueOf("3.14f");

3. Directly assigning a primitive to a wrapper reference, in which case java will automatically create an object for you, which is called as **Autoboxing** as

Long weight = 1200L;

There are many **utility functions** for the wrapper class, mainly for conversions like:

double d4 = Double.parseDouble("3.14");

Double d5 = Double.valueOf("3.14");

Auto-boxing and Auto-unboxing

From Java 5, Java does the conversion of primitive to wrapper and wrapper to primitive automatically. When you use a primitive in the context of a wrapper class, java will automatically convert (or wrap or box) the primitive into the wrapper class, which is called **auto-boxing**. When you use a

wrapper class in the context of a primitive, java will automatically convert (or unwrap or unbox) the wrapper into the primitive, which is called**auto-unboxing.**

Consider an **example without auto-boxing**:

```
Integer y = new Integer(567);

int x = y.intValue();

x++;

y = new Integer(x);

System.out.println("y = " + y);
```

Same example above with auto-boxing:

```
Integer y = new Integer(567);

y++; // unbox (unwrap), increment it, box (rewrap)

System.out.println("y = " + y);
```

Consider another **example** of a method method1 that accepts an Integer wrapper and returns an integer value:

```
public int method1(Integer i){return i;}
```

When you invoke the method as int j = method1(5); Java wrap the primitive 5 as an Integer wrapper object and pass to the method and then inside the method it unwrap Integer i and return as a integer. Note that within method1, you can call any utility method on I, which would not have been possible without boxing.

Note that **wrapper reference variables can be null**. Doing auto-unboxing on them might result in a NullPointerException.

Boxing, ==, and equals()

The == checks whether two objects are actually equal, i.e. whether they actually refer to the same object in memory. The equals() method checks whether two objects are meaningfully equal, for example, it checks whether two wrappers or strings are having the same value.

The == actually checks whether two variables are having exactly same value. Note that a primitive variable contain what you see, but a reference variable contain the address to the object that it holds. When == is used to compare a primitive to a wrapper, the wrapper will be unwrapped and the comparison will be primitive to primitive, and hence it will be true always as it is a primitive comparison and not object comparison.

Reuse of wrapper class objects

In order to save memory, two instances of the following wrapper objects (created through boxing), will always be equal (==) when their primitive values are the same:

- Boolean
- Byte
- Character from \u0000 to \u007f (7f is 127 in decimal)
- Short and Integer from -128 to 127

  For example, System.out.println(i1 == i2); will print true when i1 and i2 are having same value between -128 to 127 and will print false if i1 and i2 are outside -128 to 127 range even though both are same. This is similar to string pool concept, but with a range limit in certain cases.

  Source : http://javajee.com/wrapper-classes-auto-boxing-and-auto-unboxing