cs401 Lecture #11: Autoboxing the Wrapper classes and Templated Containers (Collections)

The Wrapper Classes

Autoboxing is a great convenience but there are a couple caveats to beware of

- Let's use the Integer class as our example.

```
Integer i = new Integer(23); // This is the traditional way to init an Integer object via its C'Tor
Integer j = 23;  // However Java permits init by direct assignment from its primitive type. (autoboxing)
```

- Autoboxing (boxing and unboxing) allows you (in most contexts) to provide a primitive where an object is expected and vice versa.

```
Integer a=12; // Works. Due the automatic boxing of the primitive value 12 into an Integer object with value 12
Integer b=12;
Integer c = 24;
int twelve=12;
if (a.intValue() == 12 ) System.out.println( "a == 12" ); // WORKS because .intValue() produces a primitive int
if (a.intValue() == twelve ) System.out.println( "a == twelve" ); // WORKS same reason
if ( (a+b) == c) System.out.println( "(a+b) == c" ); // WORKS due to the unboxing of both Integer objects and adding them to gether as primitives
if (a == twelve) System.out.println( "a == twelve" ); // WORKS due to box or unbox of one of the operands the other
if (a == b) System.out.println( "a == b" ); // WORKS due to unboxing of each operand into its primitive value. == works with primitives

// **HERE IS WHERE IT GETS FUNKY

Integer x = new Integer(12); // Note the 12 was NOT autoboxed into the x object
Integer y = new Integer(12); // Note the 12 was NOT autoboxed into the y object
if (x == y) System.out.println( "x == y" ); // NOPE. DOES NOT WORK :(

// Maybe if I make one of my operands autoboxed ?

if (a == x) System.out.println( "a == x" ); // NOPE. If either Integer object was init'd with its C'Tor then that Integer Object is unboxed and == will not work
```

Some program examples of autoboxing: Autobox1.java Autobox2.java

The Number classes (Integer, Double etc.) are most useful for the methods they offer (ParseInt) and the convenience of autoboxing for the containers that don't like primitives. The wrapper classes are immutable. Once initialized there are no setter methods to change the value of the primitive inside. Do you understand why Integer swap does not work? If you reassign a new int into an Integer object you are creating a whole new object and assigning its ref (address) into the ref var.

The principal usefulness of the Wrapper clases is with templated containers that do not allow primitives to stored in them. HashMap, HashSet and ArrayList are amoung these. Suppose you wanted to create a HashMap that represents a histogram of words and their frequency of occurance in a file. Your map would consist of entries whose key is the word and the associated value is the number of time that word was found in the file. The HashMap is the obvious choice for this problem. You woould naturally want to declare a HashMap something like this:

```
HashMap<String,int> histogram = new HashMap<String,int>();
```

You have the right idea but HashMap (and all the other templated containers) do not allow primitives to be stored in them. This is where the Integer, Double, character, etc., wrapper classes com,e into play. Your histogram must be declared as follows:

```
HashMap<String,Integer>histogram = new HashMap<String,Integer>();
```

Now you can take advantage of autoboxing to put entries into your histogram as follows:

```
String s = "foobar";
int i = 3;
histogram.put( s, i );
```

Although it looks like you have violated the Integer type by passing in a primitive int as the second arg, autoboxing takes care of the assignment of a primitive int into a Integer object.

The HashMap class implements a common data structure know as a hash table. A hash table is sometimes described as a table where each row has two columns. Within each row (pair) the first column is called the **key** and the second the **value**. Each row represents an ordered pair <key><value>. Another common name for a hash is an associative store. Each key has an associated value that is stored with it. Think of a dictionary. Assume that there only one instance of any given word. In such an analogy each row would consist of the unique word followed by its associated defintition. The first column of the entire table is the the set of all the keys of the table. There are no duplicates allowed in the keys column. No such restriction applies to the values column. When you put something into the hash, you put both the key and its associated value in at the same time. When you remove something from the hash you remove both key and its associated value art the same time. If you put a pair into the hash and your new key already exists, then you are overwriting that row in the table. In such as case the new key entry is the same as the old but the new associated value may be different.

The HashMap has several properties that must be understood clearly.

- There is no notion of ordering defined on the rows in the table. This means that although the HashMap does give a means to extract the set of keys or values into an array, those array elements will be in no specific guaranteed order.
- The runtime of get and put are O(1) or (pseudo/amortized) constant time. Exactly how this is achieved is a topic for a data structures and algortihms course.
- The keys are unique. No two rows in the hash will have the same key. If you try to put in a pair (row) that has an exising key - you end up overwriting the entire original (row) pair.

  Sample Program using HashMap

- HMDemo1.java
- StateCapitols.java

  It is in understanding these properties that the usefulness of the HashMap object is realized. When you re-write your Jumbles program using a HashMap you will find that the code is much shorter and your program runs much faster. In fact, everyone's program should run in under 1 second (on our hydrogen machine) even with no effort to optimize - as long as you use the HashMap correctly.

-