Java SE Documentation

Autoboxing

As any Java programmer knows, you can't put an `int` (or other primitive value) into a collection. Collections can only hold object references, so you have to *box* primitive values into the appropriate wrapper class (which is `Integer` in the case of `int`). When you take the object out of the collection, you get the `Integer` that you put in; if you need an `int`, you must *unbox* the `Integer` using the `intValue` method. All of this boxing and unboxing is a pain, and clutters up your code. The autoboxing and unboxing feature automates the process, eliminating the pain and the clutter.

The following example illustrates autoboxing and unboxing, along with generics and the for-each loop. In a mere ten lines of code, it computes and prints an alphabetized frequency table of the words appearing on the command line.

```
import java.util.*;

// Prints a frequency table of the words on the command line
public class Frequency {
   public static void main(String[] args) {
      Map<String, Integer> m = new TreeMap<String, Integer>();
      for (String word : args) {
          Integer freq = m.get(word);
          m.put(word, (freq == null ? 1 : freq + 1));
      }
      System.out.println(m);
   }
}

java Frequency if it is to be it is up to me to do the watusi
{be=1, do=1, if=1, is=2, it=2, me=1, the=1, to=3, up=1, watusi=1}
```

The program first declares a map from `String` to `Integer`, associating the number of times a word occurs on the command line with the word. Then it iterates over each word on the command line. For each word, it looks up the word in the map. Then it puts a revised entry for the word into the map. The line that does this (highlighted in green) contains both autoboxing and unboxing. To compute the new value to associate with the word, first it looks at the current value (`freq`). If it is null, this is the first occurrence of the word, so it puts 1 into the map. Otherwise, it adds 1 to the number of prior occurrences and puts that value into the map. But of course you cannot put an `int` into a map, nor can you add one to an Integer. What is really happening is this: In order to add 1 to freq, it is automatically unboxed, resulting in an expression of type int. Since both of the alternative expressions in the conditional expression are of type int, so too is the conditional expression itself. In order to put this int value into the map, it is automatically boxed into an Integer.

The result of all this magic is that you can largely ignore the distinction between int and Integer, with a few caveats. An Integer expression can have a null value. If your program tries to autounbox null, it will throw a NullPointerException. The == operator performs reference identity comparisons on Integer expressions and value equality comparisons on int expressions. Finally, there are performance costs associated with boxing and unboxing, even if it is done automatically.

Here is another sample program featuring autoboxing and unboxing. It is a static factory that takes an int array and returns a List of Integer backed by the array. In a mere ten lines of code this method provides the full richness of the List interface atop an int array. All changes to the list write through to the array and vice-versa. The lines that use autoboxing or unboxing are highlighted in green:

```
// List adapter for primitive int array
public static List<Integer> asList(final int[] a) {
    return new AbstractList<Integer>() {
        public Integer get(int i) { return a[i]; }
        // Throws NullPointerException if val == null
        public Integer set(int i, Integer val) {
            Integer oldVal = a[i];
            a[i] = val;
            return oldVal;
        }
        public int size() { return a.length; }
    };
}
```

The performance of the resulting list is likely to be poor, as it boxes or unboxes on every get or set operation. It is plenty fast enough for occasional use, but it would be folly to use it in a performance critical inner loop.

So when should you use autoboxing and unboxing? Use them *only* when there is an "impedance mismatch" between reference types and primitives, for example, when you have to put numerical values into a collection. It is *not* appropriate to use autoboxing and unboxing for scientific computing, or other performance-sensitive numerical code. An Integer is *not* a substitute for an int; autoboxing and unboxing blur the distinction between primitive types and reference types, but they do not eliminate it.