Lesson 27: Wrapper Classes

Primitive data types are *int*, *double*, *boolean*, *char*, and some others of less importance that we haven't studied yet. Some of those others are (See Appendix C for a summary of all the numeric data types.):

1. *long*…an integer…gives more digits than *int.*
2. *short* …an integer…gives fewer digits than *int.*
3. *float* …a floating point number (a *double* is also a floating point number)… gives fewer significant digits than *double*.

Objects required instead of primitives:

Shortly, we will begin studying classes that require primitive data types to be stored in them in a special way. The requirement will be for essentially everything to be **stored as objects**. There are special classes that permit us to **convert primitives into objects** and thus satisfy the demands of those classes that insist on being fed only objects. The classes that convert primitives to objects are called the **Wrapper Classes**…because they "wrap" the number, *boolean, or char* inside an object. Another term for this is "**boxing**" with the number being stored in a "box" (an object).

Four important wrapper classes:

The wrapper classes of greatest importance are *Integer*, *Double*, *Boolean*, and *Character* (notice the capital letters). In the examples below, notice that the awkward **pre Java5.0** way of doing this is demonstrated in the comments.

1. *Integer* class examples:

```
Integer ic = 7;          //Integer ic = new Integer(7);
int i = 10;
Integer ii = i;          //Integer ii = new Integer(i);
```

2. *Double* class examples:

```
Double dc = 1003.45; //Double dc = new Double(1003.45);
double d = -82.19;
Double dd = d;           //Double dd = new Double(d);
```

3. *Boolean* class examples:

```
Boolean bc = false;      //Boolean bc = new Boolean(false);
boolean b = true;
Boolean bb = b;          //Boolean bb = new Boolean(b);
```

4. *Character* class examples:

```
Character wc = 'X';      //Character wc = new Character('X');
char ch = 's';
Character cc = ch;       //Character cc = new Character(ch);
```

The Wrappers classes for the other primitives (*float*, *long*, etc.) are done in exactly the same way.

We can take these wrapper objects and store them in those special classes that demand them. While we are not directly storing primitives there, we are at least storing a "version" of them.

Arithmetic operations on wrapper class objects:

What if we want to multiply (or perhaps add) two wrapper class *Integers*? How do we do it? From example 1 above we have *Integer* objects *ic* and *ii*. Do we just say *ic * ii*? "Yes," if Java 5.0 is being used because it uses "auto-unboxing" to convert the object versions back into primitive types before doing the actual multiplication. **For the sake of understanding backwards compatible code**, here's how it must be done with the older versions of Java:

```
//First, convert back to int form
int j = ic.intValue( );   //Get the int value of object ic and store in j.
int k = ii.intValue( );   //Similarly, get the int value of object ii and store in k.

//Now perform the multiplication with the int versions j and k
int product = j * k;
```

Converting back to primitives:

We just looked at some "backwards" conversions above in which we converted from wrapper class *Integer* objects **back** to primitive *int* versions (also called "unwrapping" or "unboxing"). Let's look at **all** such conversions from Wrapper Class object back to primitives, but before presenting these examples it should be stated again that if Java 5.0 or higher is being used, "auto-unboxing" takes place as illustrated by:

- int i = iObj;   //iObj is an Integer object
- double d = dObj;   //dObj is a Double object
- …etc…

1. Assume *iObj* is an *Integer* object.
```
a. int i = iObj.intValue( );            //most often used…convert to int
b. short s = iObj.shortValue( );        //convert to short
c. long el = iObj.longValue( );         //convert to long
d. float f = iObj.floatValue( );        //convert to float
e. double d = iObj.doubleValue( );      //convert to double
```

2. Assume *dObj* is a *Double* object
```
a. int i = dObj.intValue( );            //convert to int…loses fractional part
b. short s = dObj.shortValue( );        //convert to short…loses fractional part
c. long el = dObj.longValue( );         //convert to long…loses fractional part
d. float f = dObj.floatValue( );        //convert to float…might lose some
                                        //precision
e. double d = dObj.doubleValue( );      //most often used…convert to double
```

3. Assume *bObj* is a *Boolean* object

```
boolean b = bObj.booleanValue( );    //convert to boolean
```

4. Assume *cObj* is a *Character* object

```
char ch = cObj.charValue( );             //convert to char
```

Likewise, the Wrapper classes for the other numeric types (*float*, *short*, etc.) have conversion methods.

Additional Methods of Wrapper Classes

Main purpose:

As was stated in the last lesson, the **main purpose** of the Wrapper classes is to **convert the primitive data types into their object equivalents**. Here, in this lesson we explore some of the other methods of the Wrapper classes.

Looking for a home:

These particular methods have **nothing** to do with the objects the Wrapper Classes produce. They could have been included in any class; however, as a matter of convenience they were placed in the Wrapper Classes…and especially the *Integer* class.

Notice that all methods given in this lesson are ***static***, i.e. **they do not require an object.**

Most frequently used:

The description and signatures of the two very most useful methods are given here:

Conversion from a *String* to an *int* type:

```
public static int parseInt(String s) //signature…from Integer class
```

Example:

```
String s = "139";
int i = Integer.parseInt(s);
```

The method *parseInt* is overloaded. Its other form is *parseInt(s, base)* where the second parameter, *base*, is the base of the number represented by *String s*.

Example:

```
String s = "3w4br";
int base = 35;
int i = Integer.parseInt(s, base); //i = 5879187
```

Conversion from a *String* to a *double* type:

```
public static double parseDouble(String s) //signature…from Double class
```

Example:
String s = “282.8026”;
double d = Double.parseDouble(s);

The equivalents of these for the *Boolean* and *Character* classes do not exist.

When using either the *parseInt* or *parseDouble* methods there is a danger of throwing an exception (causing an error). Suppose we have *String s = “123”* and we wish to convert to an *int* type. This makes perfect sense, and the following line of code using this *s* will yield an integer value of *123*.

```
int i = Integer.parseInt(s);      //yields i = 123
```

But what if *s* equals something like *“abc”*? How will the *parseInt* method react? It will throw an exception. Specifically, it will throw a *NumberFormatException*.

Base conversion methods:

In Lesson 14 we became familiar with some base conversion methods of the *Integer* class. They all converted *int* types to the *String* equivalent of various number systems. Below are examples of usage where *s* is assumed to be a *String* and *i* is assumed to be an *int* type:

```
1. s = Integer.toHexString(i);
        //…or use Integer.toString(i, 16);

2. s = Integer.toOctalString(i);
        //…or use Integer.toString(i, 8);

3. s = Integer.toBinaryString(i);
        //…or use Integer.toString(i, 2);
```

Additional methods:

A description of each is given followed by the method signature and then an example of usage:

Conversion of an *int* type to a *String*:

public String toString(int i);   //Signature…from Integer class
//See 1, 2, & 3 above for a two-parameter version.

Example:
int i = 104;
String ss = Integer.toString(i);

You should be aware that there is an easier way to convert an integer into a *String*. Just append an *int* type to an empty *String* and the compiler will think you want to make a *String* out of the combination.

Example:

```
int j = 3;
String s = “” + j;        // s will be equal to “3”
s = “” + 56;              // s will be equal to “56”
```

Conversion of a *String* to an *Integer* object.

public static Integer valueOf(String s); //Signature…from Integer class

Example:

```
String s = “452”;
Integer iObj = Integer.valueOf(s);
```

Data member constants of the *Integer* class

• *Integer.MIN_VALUE* has a value of -2,147,483,648

• *Integer.MAX_VALUE* has a value of 2,147,483,647

These two constants (see Appendix C) give the two extreme possible values of *int* variables.

The SIZE constants:

The wrapper classes *Double*, *Float*, *Long*, *Integer*, *Short*, *Character*, and *Byte* all employ the constant *SIZE*. This reports how many bits comprise the primitive types that these classes represent. The values are obtained by multiplying the number of bytes for each type in Appendix C by 8 bits in each byte. Their values are:

```
Double.SIZE = 64
Float.SIZE = 32
Long.SIZE = 64
Integer.SIZE = 32
Short.SIZE = 16
Character.SIZE = 16
Byte.SIZE = 8
Boolean does not have a SIZE constant
```