Static Imports, Default Values, Number Types and More!

Introduction

In this article from my free Java 8 course, I will explain various topics that haven't yet been covered in this series.

Static Imports

In this section I'm going to discuss static imports. In the code below, we have two imports. The first "*import org.junit.Test;*" is used to import the class and it allows you to use the short name of the class instead of its full name including the package. The second import statement, "*import static org.junit.Assert.**", imports the static class variables and methods. Because the statement ends in a '*', it will import all the static functions and variables in the class. However, you could also import only specific functions or attributes in the class.

```java
package com.marcusbiel.java8course;

import org.junit.Test;
import static org.junit.Assert.*;

public class DemoTest {
     @Test
     public void shouldDemonstrateStaticImport() {
          assertTrue(true);
     }
}
```
Example 1

In Example 1 you can see an example of using the static variables and methods from the class without using the class name. While you would normally have to call ***Assert.assertTrue***, instead you just had to call ***assertTrue***.

You should note that if a static method or variable with the same name exists in two different classes, you can only import one method by using a static import. For the second method you would have to use the full fledged name, similar to the regular import statement.

Data Types and Their Default Values

In this next section of the article I'm going to discuss the default initial values for instance variables. Let's have a look at all possible instance variable default values in Java by simply printing them out:

```
public class DemoTest {
     private byte myByte;
     private short myShort;
     private int myInt;
     private long myLong;
     private float myFloat;
     private double myDouble;
     private Object myObject;
     private boolean myBoolean;
     private char myChar;

     @Test
     public void shouldDemonstrateDataTypeDefaultValues() {
          System.out.println("byte default value" + myByte);
          System.out.println("short default value" + myShort);
          System.out.println("int default value" + myInt);
          System.out.println("long default value" + myLong);
          System.out.println("float default value" + myFloat);
          System.out.println("double default value" + myDouble);
          System.out.println("Object default value" + myObject);
          System.out.println("boolean default value" + myBoolean);
          System.out.println("char default value" + myChar);
          System.out.println("char default value as int" + (int) myChar);
     }
}
```
Example 2

Output:

```
byte default value: 0
short default value: 0
int default value: 0
long default value: 0
float default value: 0
double default value: 0
Object default value: null
boolean default value: false
char default value:
char default value as int: 0
```
Example 3

For all of the number types, their default values are zero. Objects are set by default to null, and booleans are by default false. You may wonder why the char default value appears to be blank. The default value of char is actually '\u0000', which is known as the null character. As you can see, there is no visual representation of this character. However, if you convert the default char value to an int, it is printed as "0".

Number Types

Next, we will look at number types in more detail.

There are a bunch of different number types that can be utilized in Java. The first four number types that I'll highlight are the *byte, short, int*, and *long* datatypes. All four of these can only store integer values. However, they have different ranges, as you can see in the table at the end of this section. Under normal circumstances programmers use int, because the difference in memory space between number types is relatively irrelevant these days. In some cases you might need to use *long* if your numbers are too large to be stored as an *int*.

Float and *double* are two floating point number types. Again, you can see their ranges below. If you have a decimal value in your code, it is automatically considered a double by definition. If you want to store it as a float, you have to add a capital or lowercase F at the end of the decimal value, for example you'd type '13.63F' instead of just '13.63'. If however, you wrote *float myFloat = 13.63*, that would cause an error indicating that the compiler has found a double when a float is required. You would have to instead type '*float myFloat = 13.63F.'*

Number Ranges:

Number Type	**Number of Bytes**	**Minimum Value**	**Maximum Value**
byte	1	-128	127
short	2	-32768	32767
char	2	0	65535
int	4	-2147483658	2147483647
long	8	-9223372036854775808	9223372036854775807
float	4	1.4E-45	3.4028235E28
double	8	4.9E-324	1.7976931348623157E308

Signed vs. Unsigned Number Types

Another thing I'd like to highlight is the difference between signed and unsigned number types. *Signed* means that if you were to print the value of the data type, you might see a negative sign. For example, the *byte* has a range from -128 to 127. You might wonder why the range ends with 127 instead of 128. This is because Java uses that space to store the positive or negative sign of the number.

An example of an unsigned data type is a *char*. A *char* stores characters. However, you can cast a char to a number type as shown in Example 4. In this case it is always a positive number, making *char's* range from 0 to 65535.

```
char myChar = ‘s’;
System.out.println("char default value as int" + (int) myChar); a
```

Output:

```
116
```

Example 4

Wrapper Types

Another nuance about primitive types is that they exist in parallel as objects, known as "Wrapper Types".

```
Byte b = Byte.valueOf(myByte);
```

Example 5

In Example 5, we have a variable *b* initialized using the *valueOf()* method. The *valueOf()* method is a static method that converts the primitive data type byte into a Byte object. By using *valueOf()*, a cached Byte Object is returned which will save us some memory. To create a fresh object, you would say, 'new Byte(myByte)', but generally that should not be necessary.

One of the reasons wrapper objects are useful is because they can be used in Collections. Collections are a structure similar to arrays, but they're much more flexible. Collections are very useful because you can throw objects in and take them out at will. However, you cannot throw primitive types into a collection. To work around this, we have number objects that are "wrapped" around primitive values.

All number wrapper types extend the abstract type Number. By doing this, they all implement Number functions and can be handled uniformly. This allows them to be converted back into primitive types.

Auto-Boxing and Auto-Unboxing

In Java 5, they also introduced an automatic conversion method from a primitive type to its corresponding wrapper object, called “auto-boxing” and a conversion from wrapper Objects to primitive types called “auto-unboxing”. It is heavily used by developers, however I recommend that you avoid using it because it could cause nasty *NullPointerException* errors or performance issues. All primitive values can’t be null, so using them will never throw a *NullPointerException*. However, when you don’t initialize an object, the value will be null and then when you call, say *b.byteValue()*, this throws a *NullPointerException*. You won’t actually see it when you write the code, because the compiler will automatically convert the wrapper object into the primitive data type, but you will see it when you run the code. Instead, you should typically use the static *valueOf()* method to convert your primitive values to their wrapper types.

Base 2, Base 8, and Base 16

Java enables us to not only save numbers using Base 10, but also Base 2, Base 8, and Base 16. This can be useful if, for example, you have a hexadecimal value in your documentation and you want to have the same value in your code. No matter what number system you use, it has nothing to do with how the values are stored. The computer will always store them in your memory as zeros and ones, no matter the format. Moreover, they will by default be printed out in Base 10 just like a regular number. If you want to display your value in the binary format, you will need specific formatting options which I will explain in a separate article.

Base 2 was introduced in Java 7. Before that, you could only store in Base 10, Base 8, and Base 16. In the code below, you can see some values in various bases. As I apply in the example below you can use underscores to make the numbers more readable. You can add them almost anywhere and in any amount. The only places you cannot add an underscore are at the beginning or end of the number.

```java
@Test
public void shouldDemonstrateBases(){
    int binary = 0B10;
    int baseEight = 017;
    int hex = 0xA;
    System.out.println(binary);
    System.out.println(baseEight);
    System.out.println(hex);
}
```

Output:

```
2
15
10
```

Example 6

As demonstrated in Example 6, you initialize variables with values from alternate number systems you need to indicate that in your code. For binary, you must start your numbers with ‘0B’ or ‘0b’. For base 8, the value starts with a ‘0’. For base 16, the hex code starts with ‘0x’ and the numbers 10-15 are represented by ‘ABCDEF’ or ‘abcdef’ respectively.

Thanks for reading!