Java Object toString() Method

Introduction

In this article from my free Java 8 course, I will explain the *Object toString()* method. I will also briefly touch on getClass() as it is relevant to the default *toString()* method.

The toString() Method

The *Object toString()* method returns a String that represents the object to the user that can be printed to the console or a user interface. Let's take a look at the default *toString()* method of the class Object.

```
public String toString() {
	return getClass().getName() + '@' + Integer.toHexString(hashCode());
}
```

Example 1

Example 1 shows how the *toString()* method concatenates the *getClass().getName()* method, an '@' symbol and a hexadecimal value for the *Object*'s hashCode to create a *String* that represents the object. The *getClass()* method returns the runtime class of the class the object belongs to. The *getName()* method then returns the shortname of the class without the full-fledged package name. For example, if you have a BMW class and you instantiate a BMW object, a call to *getClass().getName()* will return "BMW". The *Integer.toHexString(hashCode())* method creates a hexadecimal representation of the object's hashCode. Here is a brief example of a method that would utilize a *toString()* call:

```
@Test
public void shouldConvertBMWToString() {
	BMW bmw = new BMW(new Name("Marcus", "Biel"), new Color("silver"));
	System.out.println(bmw.toString());
	System.out.println(bmw);
}
```

Example 2

Both of the *System.out.println()* lines in Example 2 call *Object toString()*. This is because the *println()* method is overloaded, meaning that it exists in several different variations that expect different arguments. The first variation is expecting to print a String. Meanwhile, the second call is expecting an Object, which it then proceeds to call the *String.valueOf()* method, which will then call the *toString()* method. Please note that in production, generally speaking, you should use logging instead of *System.out.println()*. While *System.out.println()* works well for debugging or diagnostic information in the console, it lacks the flexibility of logging in terms of output. A logger also normally yields better performance.

Returning to the method above, either *System.out.println()* call will return BMW@e2144e4. That String isn't very useful to us, especially if we are debugging the code and trying to understand the current state of the object. Presumably, if we are calling a BMW object *toString()* we know it's a BMW object. For that reason, you should override the *toString()* method for most Entity classes.

Overriding the Object toString() Method

```
package com.marcusbiel.java8course.car;

public class BMW implements Car, Cloneable {

     private Name ownersName;
     private Color color;

     public BMW(Name ownersName, Color color) {
          this.ownersName = ownersName;
          this.color = color;
     }
}
```

Example 3

Here you can see the BMW class that I referenced in my previous example. As you saw in the last section, when we call *println(bmw.toString())*, we get something like BMW@e2144e4. That is because we have not overridden the toString method as of yet. Before we override the method, we should define what we want it to return. In the case of this class, it has two attributes: the owner's name (ownersName) and the color (color). We also may want to return what type of class the object is, and we can easily do that by calling the *getClass()* method I highlighted before.

```
@Override
public String toString() {
	return getClass().getName() + " [" + ownersName + ", " + color + "]";
}
```
Example 4

Above, I have overridden the *toString()* method for the BMW class. I used the *@override* as a tool that I can use even though it is not necessary for the code to run. It causes my compiler to make sure that I'm actually overriding a method (and not just writing a new method), and allows someone reading my code to realize that I'm overriding a method. Another point that I'd like to highlight is that I'm not writing *color.toString()*. This is unnecessary because the "+" sign between Strings allows the compiler to realize that I am concatenating strings, and automatically calls the *toString()* method for these objects.

```
@Test
public void shouldConvertBMWToString() {
	BMW bmw = new BMW(new Name("Marcus", "Biel"), new Color("silver"));
	System.out.println(bmw.toString());
}
```
Example 5

If I run this method again, assuming that we have created the *Name toString()* and the *Color toString()* methods, our output will now be "BMW [Marcus Biel, silver]". Now when we call the *toString()* method we have something more meaningful than the hashCode that we can print to the console, log, or print to a User Interface that will allow the user to see the content of the object.

StringBuilder: An Alternative to String Concatenation

The final thing I'd like to highlight in this article is the StringBuilder class. String concatenation with the "+" can cost a small amount of performance per call, and if you have a loop concatenating millions of Strings this small difference could become relevant.

However, since the compiler will replace String concatenation and use a StringBuilder in most cases, you should go for the code that is the most readable first. Further optimize for performance only when needed, covered by tests.

Below, here is an alternative *toString()* method that uses *StringBuilder* rather than concatenating the string. It will create the String dynamically, without all the plusses.

```
@Override
public String toString() {
      return new StringBuilder ("BMW [").append(ownersName).append(",")
           .append(color).append("]").toString();
}
```

Example 6

Thanks for reading!