Enums

Introduction

In this article from my free Java 8 course, I will talk about the enum. Enums are constant values that can never be changed.

The Final Tag

To display why this is useful, I'm going to start by introducing the concept of constant values in Java and talk about their flaws. Let's say we wanted to save an array, as is, so that any time it is used in the future,we know for sure which references are in which places. We could do this using the variable modifier `final`. A variable declared as `final` is locked in and cannot have a new value assigned to it. For example, if we made `persons2 final`, and then tried to set it to `null`, Java will give an exception.

```
final Person[] persons2 = {new Person(), null, null};
persons2 = null; // Compilation Error
```

Example 1

This could be very useful if you have a set of values you want to stay constant. One of the things you could do is create a set of state values. State values, as the name implies, tell us what state our program is in. Let's say our three state values for a program are PENDING, PROCESSING, and PROCESSED. We could place these String values into an array and add the `final` tag to preserve them. We're also going to make the array `static` so that it's shared among all the objects. Since it can't be modified, there's no reason not to share it among objects; also it helps us communicate our intent that these state values are the same for all objects. We'll call this array `MY_STATE_VALUES`. Here's what this would look like:

```
private static final String MY_STATE_VALUES[] = {"PENDING", "PROCESSING",
"PROCESSED"};
```

Example 2

You might notice that my variable name does not follow the default camelCase notation, but instead is composed of words in UPPERCASE LETTERS connected by underscores. This is because our 'variable', isn't really variable. It's `static` and `final`. You just want it read, the reference variable can never be modified; so to represent this difference, the naming convention is different. For `static final` variables, always use capital letters connected by underscores.

While we can't modify our array reference variable, we can iterate through it using a for-each loop. Inside this for-each loop, we could compare our states to a certain value and call some method when the value and the state match.

Comparing Strings

Since `String` is an object and not a primitive value, the equals operator ('==') doesn't work. Instead, `String` has a method `equals()` to compare equality. We call it by using `String.equals(otherString)` and it returns a boolean value (`true` / `false`) as to whether or not the two strings are equal. If you're interested in learning more about this, you should check out my article about Identity and Equality in Java.

In a for-each loop, I'm going to create three if-statements, one for each state in our MY_STATE_VALUES array. Typically when you are comparing a constant `String` value and a variable using `equals()`, you write CONSTANT_VALUE.equals(variable), because a variable could be set to `null` and this would cause an error. If we had instead written `"PENDING".equals(state)` and state was `null`, the condition would just return false and this would not cause any error. This is a general principle I advise you to aim for - whenever you can, aim to write stable code, even when the input given is invalid. In this case, I chose not to do this, because I assume that my variable is going to always be a constant and my code is more readable this way.

```
for (String state : MY_STATE_VALUES) {

     if (state.equals("PENDING")) {
          //call a method
     }
     if (state.equals("PROCESSING")) {
          //call a method
     }
     if (state.equals("PROCESSED")) {
          //call a method
     }
}
```

Example 3

So this looks like it would work great. We iterate through everything in the MY_STATE_VALUES array and the three methods are all called in the proper order. Sadly though, that's not the case. While the "`final`" modifier protects our array reference variable, it doesn't protect the array object or any of its inner references at all. If state was changed to "blah" right before our if statements, that would be a huge problem. But don't worry, there's a solution: the `enum`.

Enum

While an array that is tagged `final` provides us with an unchangeable array, as I've shown above, those values can be changed, which is a problem. An `enum` allows us to define an enumeration which means a complete ordered list of a collection. This enumeration cannot be modified once created. An example of an enumeration in real life would be a traffic light. It has three distinct values: GREEN, YELLOW and RED; no matter what happens you can never change these values. This could be extremely useful if applied to something like the set of states we talked about in Examples 2 and 3. Let's create an enum called LoggingLevel which stores our three states:

```
package com.marcusbiel.java8course;

public enum LoggingLevel {
     PENDING, PROCESSING, PROCESSED;
}
```

Example 4

An `enum` is declared similarly to a `class`, using "`public enum EnumName`". I'm going to call our enum `LoggingLevel` and for this example, I'll use the same three states we used in our array: `PENDING`, `PROCESSING`, and `PROCESSED`. Inside the declaration, we put in these values. You don't need double quotes, because these aren't strings. They're our own set of enumeration values. And that's it! You've defined your enum.

An enum is useful when we want to have a list of items, such as states or logging levels, that we want to use in our code, but we don't want them to be changeable. While the references contained in the array object can be modified, once created the enum can't be modified.

Uses for Enums

Enums are considered constant values. You can create reference variables that point to enums and print out enums' values. You can also check the equality of enums using '==' instead of the `equals()` method. This is because all enums are constants and we know that no enum value is ever copied. Therefore, checking for identity is the same as checking for equality. If you're interested in the difference between these two, check out my article Identity and Equality in Java.

```
LoggingLevel currentLoggingLevel = LoggingLevel.PROCESSING;
System.out.println("current LoggingLevel is:" + currentLoggingLevel);
System.out.println(currentLoggingLevel == LoggingLevel.PROCESSING); // true
```

Example 5

You can also search through your enumeration to find a specific enum value using a String. For example:

```
LoggingLevel currentLoggingLevel = LoggingLevel.valueOf(“PROCESSING”);
LoggingLevel illegalLoggingLevel =
LoggingLevel.valueOf(“processing”);//Compiler Error
```

Example 6

Please note that the functionality displayed in Example 6 is **case sensitive**, meaning that the second line of code would cause a compiler error.

Adding Values to Enums

Another feature of enums is that we can add constructors to them. We can assign many different types of values to each value in your enum. In this example, let’s assign number values, different from the default values, to each `LoggingLevel` value. If we create a `private` variable `i`, and set it to the number we send our constructor, we can assign each logging level a value that is usable. Since both the constructor and the value should only be visible inside the class, I’m going to make them `private`.

Since the enum will call the constructor on its own, you’re not allowed to construct enum values. This is why we make our enum constructor private.

If you want to show off and seem smart, you should know that you can technically also make an enum constructor package-private, however, it's about as useful as the human appendix. Using the package-private modifier doesn’t help because you still can’t construct an enum from outside.

```
public enum LoggingLevel {
PENDING(1), PROCESSING(2), PROCESSED(3);
private int i;

private LoggingLevel(int i) {
     this.i = i;
}
```

Example 7

Applying Enums to our Example

Now if we go back to the example I used in the beginning, we don't even need our array anymore, because we already have those values defined in the `enum`. We can instead apply all our code using the `LoggingLevel enum`. To get all the values of our `enum` we use the `values()` method, which returns all of our `enum` values in an array.

```
for (LoggingLevel state : LoggingLevel.values()) {
     if (state == LoggingLevel.PENDING) {
          // call a method
     }
     if (state == LoggingLevel.PROCESSING) {
          // call a method
     }
     if (state == LoggingLevel.PROCESSED) {
          // call a method
     }
}
```

Example 8

If you compared our code to the code in Example 3, you'd notice that we don't need to use the `equals()` method because we're not comparing two different `String` objects anymore, we're comparing constant `enum` values. This works because that's how Java defines enums. Behind the scenes Java converts our `enum` values to final primitive values that are unchangeable. So we know that whenever we call `LoggingLevel.PENDING`, it will always be the same value.

We can actually shorten our code even further if we add something called a switch-statement to replace the if statement we used in Example 5, but we'll do that in the next article which discusses the Java switch statement in detail.

Thanks for reading!