Assignment 12

Refactoring “int enum pattern” to Java enum

By:
Aster Bodden, Abdullah Alaqeel

Introduction

The Goal of this assignment is to show through a story the advantages that the Java enum has over “int pattern enum”. In order to show this we were asked to refactor an example that uses the “C style enum” into a full java enum. In order to complete the assignment we first made a sample exercise where would like to use enums. The example we chose was to have the different enum values represent a String Operation:

Example:

Each enum represent an operation to the String: “Hello world ”

```
UPPERCASE (ie. HELLO WORLD),
LOWERCASE (ie. hello world),
FIRSTUPPER (ie. Hello World),
CAMELCASE (ie. helloWorld)
```

Int Pattern Enum Design

Understanding the int enum pattern is east, you just store the enums as constant integer values. In our example that would look like this:

```
private static final int UPPERCASE   = 0;
private static final int LOWERCASE   = 1;
private static final int FIRSTUPPER  = 2;
private static final int CAMELCASE   = 3;
```

As a convention we use variable names as all capitalized. The variables are final because we want them to be constant. The static keyword makes sure that there is only one instance of the variable for the entire class. The private modifier although unnecessary it is probably a better idea to keep these variables hidden from the client. Know that we know about this type of enum, here are the reason you should use an alternative:

1. **Complete lack of type safety:**
   For example:
   ```
   int crazy = ( UPPERCASE * LOWERCASE ) + (FIRSTUPPER);
   ```
   Explanation: All the java compiler knows is that these constant are ints and therefore they have all the operations of an int, even though they represent different things and mixing them with arithmetic doesn’t make a lot of sense.

2. **Hard to print to console:**
   Explanation: In order to print the representation of each constant we would have to map a String to the inter value of the constant. To do this we would have to create an instance of Map<> and individually the int value as a key and the string as the value. This is still not error prone since we can create a silent bug in our code, where we do something like this:

```
Map<Integer, String> strings = new Map<>();
// asume all the mappings are added, and
// we have a random int variable x and by mistake we do
Strings.get(x)
// if x is in the correct range we have caused a silent bug
```

3. **Hard to iterate through all constant values**
   Explanation: This becomes a problem for the same reason that it is hard to print, We have a mapping between the values and the way it will be iterated. This could mean that we would need to iterate based on the number they represent.
   Example:

```
For (int i = 0; i < 3; i++) {
        // block to do depending on the value of i
}
```

   This will work for the way the example was made because the int values of the constant are continuous from 0 -> 2. But if we had discontinuous integers such as:

```
private Static final int OCTANE = 8;
private static final int  QUART= 4;
Private static final int QUANTITY = 3
```

   We would have to store it in a data structure, making more work for the programmer

4. **Hard and volatile maintenance:**
   Explanation: All logical statement that use these type of enums depend that the associated int value will not change. Therefore Over time if someone wants to modify the int value, it will break the code in lots of places. Even adding extra constants will need a lot of effort to modify the project structure.

Java Enum Design

The Java Language Developers solves all of these issues by creating the powerful enum types. This section will explain how changing the int pattern into the java enum is a very easy and useful thing to do.

1. **Introduce Enums to the class**
   If the class's main purpose is to enumerate using a preset enums, then make the class an Enum class. This is done by changing the declaration from `Class` to `Enum`. Otherwise, create a nested enum class. In our code, we used the first method.

2. **Change the constants into enums**
   This can be done using the following steps:

1. Since enum constants does not have a visibility modifier or a type, delete them from your code.
2. Separate the constants by commas and add a semicolon after the last constant.
3. If the variable assignments are not important, delete them. Otherwise, introduce them as constructor calls. (I.e `CONSTANT(VALUE)`)

3. **Change the relevant methods**
   If you have a method with switch statement that takes the constants as input, then make it abstract, and write the method declaration with the relevant switch case code after the enum constant declaration, surrounded by curly brackets.
   In our example, we had a method called apply, that takes in the constant and a string, and applies the relevant actions on the string. One of our constant declarations became:

```
UPPERCASE {
      public String apply(String str){
            return str.toUpperCase();
      }
}
```

   Other methods should also be adjusted if necessary to benefit from the new class structure.

4. **Write a constructor if necessary, or make it private**
   If you have constructor calls in your declarations, then you need to write the code of the constructor. If not, then write an empty constructor and make it private.

5. **Test your code**
   Make sure you test your code to make sure your new structure works and make the required adjustments.

Contributions

Aster
- Wrote most of the code
- Wrote the description of the int pattern design

Abdullah
- Added a case to the code and made some enhancements
- Wrote the description of the java enum design