enums

Venkat Subramaniam
venkats@agiledeveloper.com
http://www.agiledeveloper.com/download.aspx

Abstract

enums are not new to programming–we've used these in different languages. However, each language handles enums differently. In this article, we discuss some of the problems with enums in general, and in specific to earlier versions of Java, and show how Java 5 has by far provided a better solution.

From Adjectives

In object modeling, we generally consider nouns to represent classes and verbs the relationship between them. Adjectives, on the other hand, represent a set of listed values or enumeration of values– or *enums* for short. For instance, when ordering coffee if we say Small Cup or Large Cup, we have different sizes–Small, Medium, Large–as enumerated list of values. enums are used to represent these select list of values. However, as programmers, we often expect not only to list, but to restrict the values to the select list.

enum in 'C'

enum in C/C++ was a mere collection of numbers. From the modeling point of view, they gave us the concept of an enumerated list of values. Under the hood, these were just `int` data types. Furthermore, you could run into name collision between enums. For instance, if you define `enum ShirtSize { Large };` and `enum CoffeeSize { Large };`, you will get an error that `Large` is being redefined. There are other problems with enums as well. enums were not directly supported in earlier versions of Java, however, for most part, problems with enums in C/C++ were largely carried over to the workarounds used in Java.

enum in pre-Java 5

Java did not provide any support for enums directly. The often used workaround is to define an interface with final fields for enumerated values as in:

```
public interface ShirtSize
{
    public static final int Small = 10;
    public static final int Medium = 20;
    public static final int Large = 30;
    public static final int XLarge = 40;
    public static final int XXLarge = 50;
}

public interface CoffeeSize
{
    public static final int Small = 1;
    public static final int Medium = 2;
    public static final int Large = 3;
}
```

```
public class Test
{
    public static void orderCoffee(int size)
    {
        System.out.println("Order received for: " + size);
    }

    public static void main(String[] args)
    {
        orderCoffee(CoffeeSize.Small);
        orderCoffee(CoffeeSize.Large);

        orderCoffee(5); // Hum
        orderCoffee(ShirtSize.XLarge); // Hum
    }
}
```

Output from the above program is:

```
Order received for: 1
Order received for: 3
Order received for: 5
Order received for: 40
```

There are a number of problems with this approach. Jashua Bloch[1] has discussed this in detail in Item #21 "Replace enum construct with classes."

- enums defined as above are not typesafe. You can't restrict the value being passed for CoffeeSize (in the above example I was able to send an invalid 5 and ShirtSize.XLarge to a method that expects CoffeeSize).
- No namespace concept, you can't define two enumerations with the same set of names but with different intent within the same scope. The workaround for this is to define the final fields within a separate interface as above.
- It's brittle–adding new values to an enum may conflict with existing ones. Assume that the `CoffeeSize` interface above is modified so that `Small` is given a value of 0 instead of 1 and we introduce another final field `Standard` with a value of 1. Now if we compile `CoffeeSize`, but don't recompile `Test`, what happens? You would want the first call to `orderCoffee()` to receive a 0, but instead it still receives a 1. Reason–when `Test` is compiled, `CoffeeSize.Small` is replaced by a constant value of 1 as can be seen from the display of bytecode for the `main()` method of the `Test` class (obtained using `javap –c Test`):

```
public static void main(java.lang.String[]);
  Code:
   0:   iconst_1
   1:   invokestatic    #10; //Method orderCoffee:(I)V
   4:   iconst_3
   5:   invokestatic    #10; //Method orderCoffee:(I)V
   8:   iconst_5
   9:   invokestatic    #10; //Method orderCoffee:(I)V
   12:  bipush  40
   14:  invokestatic    #10; //Method orderCoffee:(I)V
```

```
17:  return
```

- Not printable–within the `orderCoffee()` method instead of printing 1, 3, etc. how do we print nicely Small, Large, etc. without having to use ugly condition statements?

Another problem with the above approach is the intent is not explicit. We have used enumeration using static final fields as a workaround.

typesafe enum pattern

Bloch recommends the use of *typesafe enum pattern*. Declare the enums as a separate class and enum values as instances of this class as shown in the example below:

```
public class CoffeeSize
{
    private final String theName;

    private CoffeeSize(String name)
    {
        theName = name;
    }

    public String toString() { return theName; }

    public static final CoffeeSize SMALL = new CoffeeSize("Small");
    public static final CoffeeSize MEDIUM = new CoffeeSize("Medium");
    public static final CoffeeSize LARGE = new CoffeeSize("Large");
}

public class Test
{
    public static void orderCoffee(CoffeeSize size)
    {
        System.out.println("Order received for: " + size);
    }

    public static void main(String[] args)
    {
        orderCoffee(CoffeeSize.SMALL);
        orderCoffee(CoffeeSize.LARGE);

        //orderCoffee(5); // Error
        //orderCoffee(ShirtSize.XLarge); // Error
    }
}
```

Output from the above program:

```
Order received for: Small
Order received for: Large
```

The constructor of the `CoffeeSize` class is declared `private`. The `toString()` method returns a printable name for the enum. `SMALL`, `MEDIUM`, and `LARGE` are declared as

instances of `CoffeeSize`. This approach has eliminated most of the problems mentioned in the previous section. It has several advantages:

- You can't create objects of the enum type explicitly since the constructor is private
- You have compile time type-safety (as in the above example, you get an error if you try to pass an invalid 5 or ShiftSize.XLarge).
- Not brittle, you can add constants without breaking existing code. If you introduce a CoffeeSize Standard, but don't recompile Test, the output is still correct.
- The enum is printable, you can decide what you like to print when an enum is use in `println()` method.
- Furthermore, you can add methods to your enum class and also you can have a collection of enums to work with a list of values, if you desire.

This approach still does not fully address the concern we raised in the previous section about the lack of intent and expressiveness. There are other disadvantages as well:

- How do you specify you would like either `Small` or `Large` as in `Small | Large`?
- You can't use a `switch` statement on enum values, instead you are forced to use `if`/`else` statements.
- You still have to worry about Serialization and Deserialization!
- How do you handle comparison of two enums? More code needs to be written for that.
- Result of all these considerations–a lot more effort, for you the programmer, to write enums.

Wouldn't it be nice if enums were first class citizens and directly handled by the language? Then we don't have to put much effort to create them and that would promote good practices as well. Enter Java 5.

enum in Java5

`enum`[2] is directly supported in Java 5 (even though introducing the new keyword `enum` has caused some issues with exiting code). At first sight, it looks like enum in C/C++, but only better. Behind the scene, enums are converted into classes, classes that handle the issues mentioned above. They are comparable, serializable, you can use `switch` statement on them, you can easily get a list of values, and more.

You can add methods, you can define behavior for enum, you can write different constructor to initialize it, and you can customize the methods for specific enum values.

You can write enums that are pretty simple or more complex (or more complicated!) as you desire.

A Simple enum

Let's take our `CoffeeSize` and `orderCoffee()` example to Java 5.

```
public enum CoffeeSize {SMALL, MEDIUM, LARGE};
public class Test
{
    public static void orderCoffee(CoffeeSize size)
    {
        System.out.println("Order received for: " + size);
    }

    public static void main(String[] args)
    {
        orderCoffee(CoffeeSize.SMALL);
        orderCoffee(CoffeeSize.LARGE);

        System.out.println("Available sizes are:");
        for(CoffeeSize size : CoffeeSize.values())
        {
            System.out.println(size);
        }
    }
}
```

Output of the above program is:

```
Order received for: SMALL
Order received for: LARGE
Available sizes are:
SMALL
MEDIUM
LARGE
```

`values()` is a convenience method that returns an iterator on the values of `enum`.

java.util.EnumSet

In the above example, we obtained all possible values for `CoffeeSize`. What if we want only a select few values within a range? The `EnumSet` class comes to help.

Consider the following example:

```
public enum DaysOfWeek {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
FRIDAY, SATURDAY}

import java.util.EnumSet;

public class Test
{
    public static void main(String[] args)
    {
        System.out.print("Work days are: ");
        for(DaysOfWeek day :
            EnumSet.range(DaysOfWeek.MONDAY, DaysOfWeek.FRIDAY))
        {
            System.out.print(day + " ");
        }
    }
```

```
}
```

We have defined a `DaysOfWeek enum` which has the days of the week. Assume we are interested in listing only the work days of the week (which may be a null set for some very fortunate people!). The `range()` method of the `EnumSet` allows us to do just that. The output of the above program is:

```
Work days are: MONDAY TUESDAY WEDNESDAY THURSDAY FRIDAY
```

Customizing enums

We can customize enums further as shown in the example below:

```
public enum Methodologies
{
    Evo(5),
    XP(21),
    Scrum(30);

    private int daysInIteration;
    Methodologies(int days)
    {
        daysInIteration = days;
    }

    public void iterationDetails()
    {
        System.out.println(this + " recommends " +
                  daysInIteration + " days for iteration");
    }
}

public class Test
{
    public static void main(String[] args)
    {
        for(Methodologies m : Methodologies.values())
        {
            m.iterationDetails();
        }
    }
}
```

In this example, we've defined `Methodologies` as `enum` and specified `Evo`, `XP`, and `Scrum` as possible values. We're interested in the differences in the duration of iteration of each of these methodologies. The `private` field `daysInIteration` is assigned a value within the constructor. We've also added a method `iterationDetails()` which displays the duration information for the specific methodology. The output from the above program is:

```
Evo recommends 5 days for iteration
XP recommends 21 days for iteration
Scrum recommends 30 days for iteration
```

Overriding methods

You can also override methods in enums to provide different implementation for each enum value. Let's consider an Activity of the day enum which defined different days of the week and specific activities for each day:

```
public enum ActivityOfDay
{
    SUNDAY
    {
        public void action()
        {
            System.out.println("Relax");
        }
    },

    MONDAY
    {
        public void action()
        {
            System.out.println("Watch Football");
        }
    },

    TUESDAY
    {
        public void action()
        {
            System.out.println("Swimming");
        }
    },

    WEDNESDAY
    {
        public void action()
        {
            System.out.println("Music");
        }
    },

    THURSDAY
    {
        public void action()
        {
            System.out.println("Tennis");
        }
    },

    FRIDAY
    {
        public void action()
        {
            System.out.println("Dinner");
        }
    },

    SATURDAY
```

```
    {
        public void action()
        {
            System.out.println("Movie");
        }
    };

    public abstract void action();
}


public class Test
{
    public static void main(String[] args)
    {
        System.out.println("Activity during the week");
        for(ActivityOfDay day : ActivityOfDay.values())
        {
            System.out.print(day + " - ");
            day.action();
        }
    }
}
```

In the above example, we have an `abstract` method in the `AcitvityOfDay enum`. This abstract method is being overridden for each enum value `SUNDAY`, `MONDAY`, etc. The `action()` method doesn't have to be abstract. You can provide an implementation and selectively override it for specific `enum` values, if you desire. In such case, where you don't override it would use the implementation provided in the `enum`. The output from the above program is:

```
Activity during the week
SUNDAY - Relax
MONDAY - Watch Football
TUESDAY - Swimming
WEDNESDAY - Music
THURSDAY - Tennis
FRIDAY - Dinner
SATURDAY - Movie
```

Conclusion

Java 5 has raised enum to a first class citizen. It removes some of the problems that enums have in traditional C/C++ and the workarounds in earlier versions of Java. It is very powerful. However, it also allows you to make it pretty complicated and hard to understand. Within reasons, enum in Java 5 is by far is a better solution to how enums should be handled.

References

1. Jashua Bloch, "Effective Java," Addison Wesley.
2. http://java.sun.com/j2se/1.5.0/docs/guide/language/enums.html