

A Tutorial on Enumerations in Java

Jeremy Greenwald
CS Tutors – Portland State University

July 26, 2005

1 Enumerations in Java

Prior to Java 1.5/5, Java lacked language support for enumerations. Although JDK 1.5 has introduced enumerations into the language, development will be done with JDK 1.4 for a while. This tutorial will explore three increasingly complex ways to define enumerations. Each increase in complexity will buy us increased type-safety. It is up to the individual developer to decide what level of complexity is warranted in each situation.

1.1 The Interface Antipattern

The first way to implement enumerations consists of using the java interface. This is sometimes considered an *antipattern* because interfaces are supposed to be used to describe behavior, where as this pattern uses them more to manage namespaces. But since this method is extremely easy to implement it is often used when type safety is not critical.

A java interface can contain final static fields. Consider the following interface definition.

```
public interface EnumVer1 {
    public static final int LEFT = 1;
    public static final int RIGHT = 2;
    public static final int UP = 3;
    public static final int DOWN = 4;
}
```

This enum can be used two ways. First the variables can be referenced with their fully-qualified names, such as

```
if( answer == EnumVer1.LEFT ) // move left
```

Or any class that wishes to use these variables can implement this interface,

```
class UseEnumVer1 implements EnumVer1 {  
    . . . .  
  
    void someMethod() {  
        . . . .  
        if( answer == LEFT ) // move left  
        . . . .  
    }  
}
```

Obviously this pattern is very simple to implement. But there are several problems with it. First it provides very little type safety. If you want to implement a method that takes a direction as an argument, to use `EnumVer1` the method implementation must look like this,

```
void move( int direction ) {  
    switch direction:  
        case EnumVer1.LEFT:  
            // move left  
        case EnumVer1.RIGHT:  
            // move right  
        case EnumVer1.UP:  
            // move up  
        case EnumVer1.DOWN:  
            // move down  
}
```

If you correctly document the `move` method, stating that the client is expected to use the `EnumVer1` fields *and* the client calling the `move` method actually reads and understands your documentation; then they can call the `move` method like this,

```
move( EnumVer1.LEFT );
```

this code snippet is of course much more readable than

```
move( 1 );
```

But the less readable code is still legal and the compiler will process it without comment. Even

```
move( 7 );
```

will compile without warning. To anticipate this problem the `move` method should have some error detecting code, such as a default case in the switch block. We will see how to solve this problem in the next section by using concrete classes to represent enums.

Another problem with this pattern is that client code has to explicitly mention all the values of the enum e.g. all `EnumVer1` values are hard coded in the switch statement above. If `EnumVer1` is modified to include new values every piece of code that references `EnumVer1` may have to be changed. We will see later how to solve this problem by including some helper methods in our enum classes.

1.2 Enums as Concrete Classes

Having examined the weakness of implementing enums as interfaces, let's look at using concrete classes instead.

```
public class EnumVer2 {
    public static final EnumVer2 LEFT  = new EnumVer2( );
    public static final EnumVer2 RIGHT = new EnumVer2( );
    public static final EnumVer2 UP    = new EnumVer2( );
    public static final EnumVer2 DOWN  = new EnumVer2( );
}
```

This only slightly more complicated class allows the compiler to do more type checking, hence improving the safety of any client code. Now the `move` method is defined like this,

```
void move( EnumVer2 direction ) {
    if( direction == EnumVer2.LEFT ) {
        // move left
    }
}
```

```

    } else if( direction == EnumVer2.RIGHT ) {
        // move right
    } else if( direction == EnumVer2.UP ) {
        // move up
    } else if( direction == EnumVer2.DOWN ) {
        // move down
    } else {
        // report unknown direction
    }
}

```

Now a user of the `move` method can only call `move` with an argument of type `EnumVer2`, even though `EnumVer2` has no fields. The identity of the different values of `EnumVer2` correspond only to their memory addresses. Since they are also final, their addresses won't change during the lifetime of any one VM. If a user attempts to call `move` like this,

```
move( 1 );
```

the compiler will report a symbol not found error, since the method `move(int)` does not exist. The else clause in the `move` method is now much less likely to be executed because of this type-checking.

There are still a few problems with this pattern. First, the constructor is public, new `EnumVer2` objects can be instantiated. That can be easily solved by making the constructor private. Second, any client code that calls the `move` method will still have to write a lot of code to decide which `EnumVer2` object to use. For example imagine a text-base game that asks a player which way they want to move. If the variable `answer` is a `java.lang.String` calling `move` might look something like this,

```

if( answer.equals("left") ) {
    move( EnumVer2.LEFT );
} else if( answer.equals("right") ) {
    move( EnumVer2.RIGHT );
} else if( answer.equals("up") ) {
    move( EnumVer2.UP );
} else if( answer.equals("down") ) {
    move( EnumVer2.DOWN );
} else {

```

```
    // report unknown direction
}
```

Any change to `EnumVer2` might require a change to every section in the game code that uses `EnumVer2`. For example, if it is decided that a player can also jump, this if-then-else block would have to be modified to prevent it from reporting an unknown direction when the user wants to jump. This would not be a difficult change, but the more places that have to be modified, the more likely a bug will go unnoticed by the developer or the compiler. This will be solved by adding a few helper methods to the enum in the next section.

1.3 Providing Useful Helper Methods

Having shown a simple implementation of an enum, let's look at adding a few helper methods that will make the enum easier to use,

```
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
public final class EnumVer3 {
    // instance fields
    private final String moveName;

    // static field
    private static final List _VALUES = new ArrayList();

    /** Unmodifiable view of all move types */
    public static final List VALUES = Collections.unmodifiableList( _VALUES );
    // the different values of the enum
    public static final EnumVer3 LEFT =
        new EnumVer3( "left" );
    public static final EnumVer3 RIGHT =
        new EnumVer3( "right" );
    public static final EnumVer3 UP =
        new EnumVer3( "up" );
    public static final EnumVer3 DOWN =
        new EnumVer3( "down" );
```

```

private EnumVer3( String name ) {
    this.moveName = name;
    EnumVer3._VALUES.add( this );
}

public String toString() {
    return moveName;
}

public static EnumVer3 valueOf( String i ) {
    Iterator iter = VALUES.iterator();
    while( iter.hasNext() ) {
        Object obj = iter.next();
        if( i.equalsIgnoreCase( obj.toString() ) ) {
            return (EnumVer3)obj;
        }
    }

    throw new IllegalArgumentException( "Unknown move direction" );
}
}

```

In this class we have made the constructor private. This ensures that users of this class can not instantiate new `EnumVer3` objects.

Next notice that `_VALUES` is a private field. This field is used to hold all the values of the enum as they are being constructed. We don't want to make this field public because this would allow a client to add values to it. While they couldn't add new `EnumVer3` objects (remember we made the constructor private), they could add additional copies of any of `EnumVer3` objects already constructed. They could also add any other object they want since `_VALUES` is of type `List` whose `add` method takes an `Object` as an argument. We don't want to have to check every object we pull out of `_VALUES`. In general anything we don't want a user of our enum to do, we should try to make impossible for them to do. But we would still like to have a visible `List` that holds every value of our enum. So we declare another field, `VALUES`, to be

public and initialize it using the `unmodifiableList(List)` factory method in the `java.util.Collections` class. Now the user can safely inspect all the values of `EnumVer3` through the public field `VALUES`.

It is important that `_VALUES` be declared and initialized before any `EnumVer3` objects are created. This is because the (private) constructor attempts to add objects to `_VALUES`. If it were not initialized before these objects are instantiated the VM would throw a `NullPointerException`. Also note that `VALUES` can be initialized before the enum values are instantiated. Any object added to `_VALUES` after `VALUES` is initialized will be viewable through `VALUES`. This is a standard feature of the anonymous class implemented inside the `java.util.Collections` class.

Now let's look at the `valueOf(String)` method. It allows client code to get one of the enums without having to explicitly loop over all possible values. Recall our text-base game, it can now check the `answer` like this,

```
try {
    move( EnumVer3.valueOf( answer ) );
} catch( IllegalArgumentException iae ) {
    // report unknown direction
}
```

Notice how this code snippet is more compact than our earlier switch or if-then-else statements. Also if new values of `EnumVer3` are created the above code doesn't need any modification (although the `move` method still does, but there really isn't a way to get around that). Keep in mind that an enum might have a lot more than four values and there could be dozens of places where code might want to check user input. This is why we put all instantiated `EnumVer3` objects in `_VALUES`. If we wanted to add a `JUMP` value to the `EnumVer3` class, we would only have to add two lines to `EnumVer3`.

```
public static final EnumVer3 JUMP =
    new EnumVer3( "jump" );
```

The constructor and the `valueOf` method wouldn't need any modifications.

This is the same reason we have the public `VALUES` variable. If we wanted to list out all the possible moves a player could make, we could iterate over `VALUES`. When we added `JUMP` to `EnumVer3`, it would automatically show up in the list, without requiring any changes out of `EnumVer3`.

2 Conclusions

This tutorial has shown some of the dangers of using the Java interface to implement enums. It has also shown how to overcome these dangers by using concrete classes. But the price we paid was an almost ten times increase in code size (5 lines vs 46). And we have only added some basic functionality to our enums. This author has written a class similar to `EnumVer3` to help parse command line switches. It uses reflection to check the values of the switches, construct other objects, and to build help and usage documentation. But these features have grown the class to 200 lines. Clearly not every situation will require this much complexity. The individual developer should be aware of when 5 lines is good enough and when 46 are required.