

Software Development (CS2500)

Lecture 44: The Joys of enums

M.R.C. van Dongen

February 9, 2011

Contents

1	Outline	1
2	Multiway Branching	2
3	Int Enums	4
4	Enums to the Rescue	5
5	State and Behaviour	6
6	Specific Behaviour	8
7	For Friday	10
8	Acknowledgements	10

1 Outline

Many applications require the use of named constants. For example: A suit of cards: HEARTS, SPADES, CLUBS, and DIAMONDS; predefined colours: BLACK, WHITE, RED, BLUE, ...; and so on.

These named constants are also known as *enumerated types*: in Java enums. They are the topic of this lecture.

We start with the `switch` statement. This is a multi-way branching construct. It is not really about enums, but we need it in some examples. We study a common, flawed pattern called *int enums*. Java enums overcome most of the flaws of `int enums`. Java enums are just objects. They may have data and common and specific behaviour. The last part of this lecture — the part about enums — is based on [Bloch, 2008, Item 30]. This includes examples. Some of this lecture is based on the Java API documentation.

2 Multiway Branching

The switch statement allows a Java program to make a decision based on a single value. It allows you to make similar decisions as the following code fragment.

```
if (var == 0) {  
    // First stuff  
} else if (var == 1 || var == 3) {  
    // Second stuff  
} else if (var == 2 || var == 4) {  
    // Third stuff  
} ...  
} else {  
    // Final stuff  
}
```

Don't Try this at Home

With the switch statement you write it as follows:

```
switch (var) {  
case 0: // First stuff  
case 1:  
case 3: // Second stuff  
case 2:  
case 4: // Third stuff  
...  
default: // Final stuff  
}
```

Java

The decisions in the switch statement depend on the values of the *guard* values before the colons.

In the simplest case, the construct is written as follows.

```
switch (<expr>) {  
case <constant #1>: <statements #1>  
case <constant #2>: <statements #2>  
...  
case <constant #n>: <statements #n>  
}
```

Java

The statements $\langle \text{statements \#1} \rangle$ – $\langle \text{statements \#n} \rangle$ are statements which may contain the *break* statement. For the moment we shall assume that the statements are non-empty. The switch statement works as follows.

1. First $\langle \text{expr} \rangle$ is evaluated. It should evaluate to an integer or character constant or to an enum.
2. Let $\langle \text{res} \rangle$ be the result of evaluating $\langle \text{expr} \rangle$.
3. The constant $\langle \text{constant \#i} \rangle$ is the *guard* of $\langle \text{statement \#i} \rangle$.

4. $\langle \text{res} \rangle$ is compared against the guards until the first match.
5. If there is no match then nothing happens.
6. Otherwise, let $\langle \text{constant } \#m \rangle$ be the first match.
7. $\langle \text{statements } \#m \rangle$ are carried out.
8. If evaluating the statements in $\langle \text{statements } \#m \rangle$ requires the evaluation of the break statement then this transfers the flow of control to the end of the switch statement.
9. Otherwise, the switch statement continues with matching $\langle \text{expr} \rangle$ against $\langle \text{constant } \#m+1 \rangle$, $\langle \text{constant } \#m+2 \rangle$, and so on.

In the previous explanation it was assumed that the statements were all non-empty. In general the switch statement also allows empty statements. For example, in the following $\langle \text{statements} \rangle$ is carried out if $\langle \text{expr} \rangle$ is equal to $\langle \text{constant } \#1 \rangle$, if it is equal to $\langle \text{constant } \#2 \rangle$, ..., or if it is equal to $\langle \text{constant } \#m \rangle$.

```
switch ( $\langle \text{expr} \rangle$ ) {  
  case  $\langle \text{constant } \#1 \rangle$ :  
  case  $\langle \text{constant } \#2 \rangle$ :  
  ...  
  case  $\langle \text{constant } \#m \rangle$ :  $\langle \text{statements} \rangle$   
  ...  
}
```

Java

Usually, the switch statement is used with a default case, which acts as a universal guard (a guard that matches anything) and covers the default case.

```
switch ( $\langle \text{expr} \rangle$ ) {  
  case  $\langle \text{constant } \#1 \rangle$ :  $\langle \text{statements } \#1 \rangle$   
  case  $\langle \text{constant } \#2 \rangle$ :  $\langle \text{statements } \#2 \rangle$   
  ...  
  case  $\langle \text{constant } \#n \rangle$ :  $\langle \text{statements } \#n \rangle$   
  default:  $\langle \text{default statements} \rangle$   
}
```

Java

The following is an example. The variable character is a char.

```
switch (character) {
    case 'A':
    case 'B':
    case 'C':
        System.out.println( "Range: A-C." );
        break;
    case 'e':
        System.out.println( "It's an 'e'" );
        break;
    default:
        System.out.println( "It's not in {A,B,C,e}" );
}
```

3 The int-enum Anti-Pattern

In the build up to enums we shall first study a commonly used programming anti-pattern called the int-enum pattern [Bloch, 2008, Item 30].

An *enumerated type* is a type whose legal values consist of a fixed set of constants. For example: the seasons of the year, the suits in a deck of cards, Frequently, enumerated types are implemented using constant ints. Unfortunately, this is not a good idea. The following demonstrates the anti-pattern.

```
public static final int APPLE_FUJI    = 0;
public static final int APPLE_PIPPIN  = 1;

public static final int ORANGE_NAVEL  = 0;
public static final int ORANGE_TEMPLE = 1;
public static final int ORANGE_BLOOD  = 2;
```

Don't Try this at Home

This technique is called the *int enum pattern*. *Never, ever, ever, use it*. It is seriously flawed. The following are some problems with int enums.

Type safety: Int enums don't provide type safety. Since int enum values are ints, the compiler can't detect when you're comparing apples and oranges. Also it can't detect when values are out of range.

```
if (APPLE_FUJI == ORANGE_BLOOD) { /* ?? */ }
int apple = ORANGE_BLOOD;        // ??
```

Don't Try this at Home

Maintainability: Programs with int enums are brittle. Int enums are compile-time constants. They are compiled into clients that use them. If an enum constant changes, the client will break. Unless, of course, it is recompiled.

Ease of use: Int enums are difficult to use. It is difficult to translate them to Strings. There is no reliable way to iterate over all allowed int enum values.

Namespace: Int enum types have no private name space. This is why usually a prefix is added to the constant names to implement a “namespace”. The prefix corresponds to the type. For example, you add `APPLE_` before apple constant names, `PEAR_` before pear constant names, and so on.

4 Enums to the Rescue

As of Release 1.5 Java provides the enum *type*. They overcome most, if not all, shortcomings of int enums. Using enums you would implement the fruit example from the previous section as follows.

```
public enum Apple { FUJI, PIPPIN }
public enum Orange { NAVEL, TEMPLE, BLOOD }
```

Java

Each ‘public enum {class} { {constants} }’ is a *class*. Each constant in {constants} is an instance of the class: an *object*. For each constant in any enum class, Java automatically defines one public final class attribute. The name of the constant {constant} in class {class} is {class}. {constant}. For example the Java compiler automatically translates the first enum class from the previous example to a class file that has constants `Apple.FUJI` and `Apple.PIPPIN`. Java enum classes are *final* and have *no* public constructors.

The following demonstrates that enums are better than int enums.

Type safety: Java enums are type safe. If you try to write the following, the compiler will complain.

```
if (Apple.FUJI == Orange.BLOOD) { /* ?? */ }
Apple apple = Orange.BLOOD;      // ??
```

Don't Try this at Home

The reason why the compiler will complain is that it will only let you compare Apples with Apples, let you assign Apples to Apple variables, and let you use Apple values where Apple values are expected. (However, since enums *are* objects, it *is* possible to use `null` where an enum value is expected.)

Maintainability: Java does not compile enums as constants into clients that use them. You can rearrange enum values without breaking clients.

Ease of use: As we shall see shortly, it is easy to translate enums to Strings and easy to iterate over all enum constants in the class.

Namespace: Enum classes have a private name space. So you can have two enum constants in two different enum classes, where the constants have the same name.

The following are some of the methods which are defined for enum classes.

compareTo(that): Compares this enum with that for order.

equals(that): Returns true if this enum equals that.

hashCode(): Returns a hash code for this enum.

toString(): Returns the name of this enum constant. The returned String is the same name as declared in the enum declaration. (Unless the method is overridden, of course.)

name(): This is a final method. Returns the name of this enum. The returned String is the same name as declared in the enum declaration.

ordinal(): Returns the *ordinal* of this enum. Here the ordinal of the enum is the position (an int) in its enum declaration. As usual, the first ordinal value is zero.

5 State and Behaviour

We've seen that Java enums are flexible. But will they let you implement specific behaviour? For example, what if you want the colour of a fruit? In the remainder of this section we shall implement an enum class which is interesting enough to demonstrate how enum classes can implement state and behaviour.

Consider the eight planets of the solar system. Each planet has a mass and a radius. Using the mass and radius you can compute the planet's surface gravity. Notice that normally you would implement the mass and radius as instance attributes and compute the surface gravity with an instance method. This is exactly what we're going to do in our enum class.

The following is a possible implementation of our Planet class.

```
public enum Planet {  
    MERCURY( 3.303e+23, 2.439e6 ),  
    VENUS   ( 4.869e+24, 6.052e6 ),  
    EARTH   ( 5.975e+24, 6.378e6 ),  
    MARS    ( 6.419e+23, 3.393e6 ),  
    JUPITER( 1.899e+27, 7.149e7 ),  
    SATURN  ( 5.685e+26, 6.027e7 ),  
    URANUS  ( 8.683e+25, 2.556e7 ),  
    NEPTUNE( 1.024e+26, 2.477e7 );  
  
    // Universal gravitational constant in m^3/kg s^2.  
    private static final double G = 6.67300E-11;  
    private final double mass;  
    private final double radius;  
    private final double gravity;  
  
    Planet( double mass, double radius ) {  
        this.mass    = mass;  
        this.radius  = radius;  
        gravity = G * mass / (radius * radius);  
    }  
  
    public double getMass( )    { return mass; }  
    public double getRadius( ) { return radius; }  
    public double getGravity( ) { return gravity; }  
}
```

Java

Before studying the start of the class let's have a look at the instance attributes: `mass`, `radius`, and `gravity`. They look pretty much as attributes of any other class and there's nothing special about them.

Next let's look at the constructor. This also works pretty much as expected. For our `Planet` application the constructor uses its arguments to initialise the attributes of the object which is currently being constructed. However, there is one peculiar aspect about the constructor: it is implicitly private. This is caused by the fact that the constructor is that of an enum class.

The instance methods `getMass()`, `getRadius`, and `getGravity()` also work as per usual: given a class instance reference they are used to get attributes of that instance. So if you write `PLUTO.getMass()` you get the mass of `PLUTO`.

Finally, let's have a look at the constants at the top of the class. For our `Apple` and `Pear` class there were no parentheses and arguments inside them. Looking back we can now see why arguments are needed here and not in the `Apple` and `Pear` class. After all, *something* must be responsible for constructing the `Planet` objects. So `MERCURY(3.303e+23, 2.439e6)` constructs the object called `MERCURY`, `VENUS(4.869e+24, 6.052e6)` constructs the object called `VENUS`, and so on.

Having implemented the class, we can now build some more functionality on top of it. The following class can print some useful information about the planets.

```
public class WeightTable {
    public static void main( String[] args ) {
        for (Planet planet : Planet.values() ) {
            double weight = surfaceWeight( planet, 1.0 );
            System.out.println( "1kg on " + planet
                               + " has a surface weight of "
                               + weight + "." );
        }
    }

    private static double surfaceWeight( Planet planet, double mass ) {
        return mass * planet.getGravity();
    }
}
```

When we run the program we get the following.

```
$ java WeightTable
1kg on MERCURY has a surface weight of 3.7051525865812165.
1kg on VENUS has a surface weight of 8.870805573987766.
1kg on EARTH has a surface weight of 9.80144268461249.
1kg on MARS has a surface weight of 3.720666819023476.
1kg on JUPITER has a surface weight of 24.794508028173404.
1kg on SATURN has a surface weight of 10.443575504720215.
1kg on URANUS has a surface weight of 8.868889152162147.
1kg on NEPTUNE has a surface weight of 11.137021762915634.
$
```

Wow. That's pretty impressive for a short program like that. Let's get back and see why the program is so short.

The first reason why the program is so short is that class method `values()` is very convenient: you get it for free with any `enum` class. The method simply returns an array consisting of all `Planet` constants. Using the enhanced `for` notation we iterate over all the planets.

The second reason why the program is so short is that `toString()` properly returns the names of the `Planet` constants. Again, you get this behaviour for free with any `enum` class.

With `int` enums you could never have implemented this application with such little programming effort.

Finally, there is no reason why all methods in `enum` classes should be getter methods. For example we could have implemented an instance method `double surfaceWeight(double mass)` in the `Planet` class.

```
public double surfaceWeight( double mass ) {  
    return mass * gravity;  
}
```

Java

With this method we could have written the `for` loop in the `WeightTable` program as follows:

```
for (Planet planet : Planet.values( )) {  
    System.out.println( "1kg on " + planet  
                        + " has a surface weight of "  
                        + planet.SurfaceWeight( 1.0 ) + "." );  
}
```

Java

6 More Specific Behaviour

Our `Planet` application is very well behaved. The result of all methods depends on the input and instance attributes only. This is not always the case. For example, consider a calculator application. There are four operations `PLUS`, `MINUS`, `TIMES`, and `DIVIDE`. Ideally, we'd like to apply each operation to two doubles and get the result. So the signature of the method should be something like `double apply(double first, double second)`. Furthermore, `PLUS.apply(0.0, 1.0)` should return `1.0`, `MINUS.apply(0.0, 1.0)` should return `-1.0`, and so on. *Here the result also depends on the enum constant.*

So, how do we implement this? The following is a first try.


```

public enum Operation {
    PLUS, MINUS, TIMES, DIVIDE;

    public double apply( double first, double second ) {
        double result;
        switch(this) {
            case PLUS:    result = first + second; break;
            case MINUS:   result = first - second; break;
            case TIMES:   result = first * second; break;
            case DIVIDE:  result = first / second; break;
            default: String error = "Unknown Operation: " + this;
                     throw new AssertionError( error );
        }
        return result;
    }
}

```

Don't Try this at Home

This is not very pretty. First we have to implement a default case. Without the default case the compiler would have complained because it doesn't know that all cases cover all enum constants.

The second reason why this attempt is not very pretty is that the code is fragile. If an Operation is added or removed then we have to change the method.

The following is much prettier. First we make apply abstract. Next we let each enum value implement its *own* behaviour by overriding an abstract method.

```

public enum Operation {
    PLUS { @Override
        public double apply( double x, double y ) { return x + y; } },
    MINUS { @Override
        public double apply( double x, double y ) { return x - y; } },
    TIMES { @Override
        public double apply( double x, double y ) { return x * y; } },
    DIVIDE { @Override
        public double apply( double x, double y ) { return x / y; } };

    public abstract double apply( double first, double second );
}

```

Java

The group following the name of the enum constants acts as the body of a private class which may be used to implement specific behaviour of the constants. Inside you may have attributes, override abstract methods, and implement private methods.

That was pretty nice, but for the Operation constants to "print" themselves in a meaningful way then we need to override toString() on a per constant basis. This is pretty standard.

```

public enum Operation {
    PLUS    { @Override
              public String toString( ) { return "+"; }
              @Override
              public double apply( double x, double y ) { return x + y; }},
    (rest of class omitted)
}

```

With this class we can write programs as the following without much effort.

```

public class Calculator {
    public static void main( String[] args ) {
        for (Operation op : Operation.values( )) {
            double first  = 6;
            double second = 2;
            double result = op.apply( first, second );
            System.out.println( first + " " + op + " " + second
                               + " = " + result );
        }
    }
}

```

We may use the program as follows:

```

$ java Calculator
6.0 + 2.0 = 8.0
6.0 - 2.0 = 4.0
6.0 * 2.0 = 12.0
6.0 / 2.0 = 3.0
$

```

7 For Friday

Study the lecture notes, and [Bloch, 2008, Item 30] if you have the book.

8 Acknowledgements

The second part of this lecture is based on [Bloch, 2008, Item 30]. Some of this lecture is based on the Java API documentation.

References

[Bloch, 2008] Joshua Bloch. *Effective Java*. Addison–Wesley, 2008.