Software Development (CS2500)

Lecture 41: The Joys of `enums`

M. R. C. van Dongen

January 22, 2014

Contents

1 Outline

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

These groups of named constants are better 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. Next 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 state and common and specific behaviour. The last part of this lecture—the part about enums—is based on [Bloch 2008, Item 30]. Some of this lecture is based on the Java API documentation.

2 Multiway Branching

The switch statement makes a decision based on a single value. It lets you make similar decisions as the following code fragment.

Don't Try this at Home

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

With the switch statement you write it as follows:

Java

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

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.

Java

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

The statements ⟨statements #1⟩–⟨statements #n⟩ are normal statements that may contain the *break* statement. If there is a break statement in ⟨statements #*i*⟩, it should be last statement. For the moment we shall assume that the statements are non-empty. The switch statement works as follows.

1. First ⟨expr⟩ is evaluated. It should evaluate to an integer or character constant or to an enum.
2. Let ⟨res⟩ be the result of evaluating ⟨expr⟩.

3. The constant ⟨`constant #i`⟩ is the *guard* of ⟨`statement #i`⟩.
4. ⟨`res`⟩ is compared against the guards from top to bottom until the first match.
5. If there is no match then nothing happens.
6. Otherwise, let ⟨`constant #m`⟩ be the first match.
7. ⟨`statements #m`⟩ are carried out.
8. If the last statement in ⟨`statements #m`⟩ is a `break` statement, this terminates the `switch` statement.
9. Otherwise, the `switch` statement continues with matching ⟨`expr`⟩ against ⟨`constant #m+1`⟩, ⟨`constant #m+2`⟩, and so on.

In the previous explanation it was assumed that the statements were all non-empty. In general the `switch` statement also may also have empty statements. For example, in the following ⟨`statements`⟩ is carried out if ⟨`expr`⟩ is equal to ⟨`constant #1`⟩, if it is equal to ⟨`constant #2`⟩, ..., or if it is equal to ⟨`constant #m`⟩.

```Java
switch (⟨expr⟩) {
case ⟨constant #1⟩:
case ⟨constant #2⟩:
...
case ⟨constant #m⟩: ⟨statements⟩
...
}
```

It you cannot guarantee that ⟨`expr`⟩ will be matched by some `case`, you must add a `default` case after the last `case`. The `default` acts as a universal guard (a guard that matches anything) and covers the default case.

```Java
switch (⟨expr⟩) {
    case ⟨constant #1⟩: ⟨statements #1⟩
    case ⟨constant #2⟩: ⟨statements #2⟩
    ...
    case ⟨constant #n⟩: ⟨statements #n⟩
    default: ⟨default statements⟩
}
```

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

```Java
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 `enum`s we shall 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 `int`s. Unfortunately, this is not a good idea. The following demonstrates the anti-pattern. The example implements two classes of `enum`s: one for apples, and one for oranges.

Don't Try this at Home

```
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;
```

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 `int`s, the compiler can't detect when you're comparing apples and oranges. Also it can't detect when values are out of range.

Don't Try this at Home

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

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 `String`s. 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 programmers usually add a prefix 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 DIY

In this section we shall try to implement proper enumerated constants. This is only intended as an exercise because the next section will show that `Java` already provides enumerated classes.

The following is a possible implementation of a `Beef` class. We're using class constants `SHANK` and `SIRLOIN` to represent the enumerated `Beef` constants. Each constants is represented by an instance of a unique anonymous class.

```Java
public abstract class Beef {
    public static final Beef SHANK = new Beef( ) {
        @Override public double price( ) { return 1.0; }
    };
    public static final Beef SIRLOIN = new Beef( ) {
        @Override public double price( ) { return 2.0; }
    };
    public abstract double price( );

    private Beef( ) { }

    public static void main( String[] args ) {
        final Beef shank = Beef.SHANK;
        final Beef sirloin = Beef.SIRLOIN;
        ...
    }
}
```

Notice that the enumerated constants define proper `Beef` behaviour, which is implemented by overriding the abstract method `Price`.

Unfortunately, the `Beef` class has a serious disadvantage because the class cannot be made `final`. (Why?) This really is a problem because it provides the programmer the freedom to implement bogus `Beef` objects.

```Java
public class MrEd extends Beef, implements Horse {
    @Override public double price( ) { return 0.2; }

    @Override public void talk( ) { ... }
}
```

Allowing non-final classes such as the `Beef` class may have serious implications for the food supply chain…

Fortunately, `Java` already provides a mechanism for defining enumerated constants.

5 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 `enum`s you would implement the fruit example from the previous section as follows.

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

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`. All `Java` `enum` constructors are (implicitly) `private`. All instance methods are `final`, except for `toString( )`.

The following demonstrates that `enum`s are better than int enums.

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

```Don't Try this at Home
if (Apple.FUJI == Orange.BLOOD) { /* ?? */ }
Apple apple = Orange.BLOOD;     // ??
```

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. This is the only method that may be overridden. The returned `String` is the same name as declared in the `enum` declaration. (Unless the method is overridden, of course.)

`name( )`: Returns the original 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.

6 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 shows how you may implement state and behaviour in an `enum` class.

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.

```Java
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 = 6.6730OE-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; }
}
```

Before studying the start of the class let's have a look at the instance attributes: `mass`, `radius`, and `gravity`. They look pretty much like any instance attribute of any other class; 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 that is currently being contructed. However, there is one peculiar aspect about the constructor: it is implicitly private. This is because constructors of `enum` classes are implicitly `private`.

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 when 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 we need arguments in the `Planet`. After all, the `Planet` objects have different state (mass, radius, gravity) and we can only initialise the state when we're 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 prints some useful information about the planets.

```Java
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( final Planet planet, final double mass ) {
        return mass * planet.getGravity( );
    }
}
```

When we run the program we get the following.

```Unix Session
$ 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 because the 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 because `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.

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

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

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

7 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 `double`s and get the result. Therefore, the signature of the method should be something like `double apply( double first, double second )`. Forthermore, `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.

Don't Try this at Home

```
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;
    }
}
```

This is not very pretty. First we have to implement a `default` case because the `default` case the compiler doesn't know that all `case`s cover all possible `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.

Java

```
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 );
}
```

The group following the name of the `enum` constants acts as a 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.

```Java
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.

```Java
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:

```Unix Session
$ 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
$
```

8 Improvement

Looking back at the implementation of the previous section, we can see that this implementation isn't pretty at all. There still is a lot of repetition in the code: all overrides of `toString( )` are identical (up to a symbol that is completely determined by the `Operator`). This suggests the symbol really should be an attribute of the `Operator`. By introducing this attribute, we can replace all overrides by a single override, thereby improving the maintainability of the code. The following demonstrates the idea.

```Java
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 );
    private final String symbol;

    Operation( String symbol ) {
        this.symbol = symbol;
    }

    @Override public String toString( ) { return symbol; }
}
```

9 Strategy Enums

In this section we shall study some more examples of specific behaviour for `enum` constants. Throughout this section we shall use a payroll aplication. For simplicity this application uses `doubles` to represent moneys. This isn't really a good idea because `double` arithmetic may cause rounding errors, which may make the results unreliable. Usually, it's better to represent moneys (and other quantities that require *exact* representation) with objects that support arbitrary number precision. For example, we could use the `BigDecimal` class. However, for the rest of this section we shall ignore this and represent our moneys as `doubles`.

Our application computes the total pay of an employee for a given day based on their pay rate, and the day of the week. The following are the rules.

- Employees have a pay rate that depends on their grade.
- Our application gets the pay rate as its input.
- An employee's pay for a given day of the week is given by

$$\text{pay} = \text{base pay} + \text{overtime pay for that day}\,.$$

- The base pay is given by pay rate $\times$ hours worked.
- The overtime pay is given by

$$\text{overtime pay} = \text{pay rate} \times \text{overtime hours}/2\,.$$

The overtime hours depend on the kind of day.

Weekdays: For a week day the overtime hours are the hours worked on that day in excess of 8 hours, where 8 is the hours per shift.

Weekend: For weekend days, the overtime hours are the hours worked on that day.

9.1 First Stab at Implementation

That looks pretty simple. Many programmers may implement this as follows.

Don't Try this at Home

```
public enum SimplePayrollDay {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;

    private static final int HOURS_PER_SHIFT = 8;

    public double pay( double hoursWorked, double payRate ) {
        double basePay = hoursWorked * payRate;
        double overtimePay = overtimePay( hoursWorked, payRate );

        return basePay + overtimePay;
    }

    public double overtimePay( double hoursWorked, double payRate ) {
        double overtime;

        switch (this) {
        case SATURDAY:
        case SUNDAY: // Weekend
            overtime = hoursWorked;
            break;
        default:     // Weekday
            double difference = hoursWorked - HOURS_PER_SHIFT;
            overtime  = (difference < 0 ? 0 : difference);
        }
        return overtime * payRate / 2;
    }
}
```

At first sight, there doesn't seem much wrong with this implementation. However, as is usual, the problem is maintenance. What if we add an extra type of day? For example, a Bank Holiday (special kind of Monday). Should that happen, we'd have to change the implementation of `overtimePay( )`. The application will break if we forget to make the change.

9.2 Strategy Enum to the Rescue

So, how do we fix the implementation? The idea is that we make sure the compiler helps us if we forget to make a change to the `pay( )` computation. The `switch` statement gives some clue. (The kind of pattern we're looking for is similar to the pattern that we factored out at the start of this lecture.)

We need different *strategies* for paying overtime. This is different from the strategy for `toString( )` in the `Operation` class, which is the same for all instances of the class. Here, *some* strategies are shared, but not all. Currently we have two strategies. Each strategy is *determined by* the kind of day: week days, and weekend days. The kind of day is a *property* of the day. A property can be implemented as an *attribute*. Implementing the property as an attribute lets the attribute *determine* the kind of day. We can *compute* the kind of day from the attribute. Likewise, the kind of day *determines* the strategy. Therefore, the attribute *determines* the strategy.

We could implement our attribute as a `boolean`: `isWeekday`. However, this approach is the same as using the `int`-enum anti-pattern. It would work for now, but changing the requirements could break the implementation. For example, what if we get double overtime for hours worked on Christmas days? It is probably better to have a new strategy based on an `enum` type. This *strategy `enum`* determines the strategy for computing overtime pay. (Of course we implement it as an inner

(enum) class.)

The following is a possible implementation. Some of the details of the inner class PayType (an enum class) are omitted. They are listed further on.

```Java
public enum PayrollDay {
    SUNDAY(    PayType.WEEKEND ),
    MONDAY(    PayType.WEEKDAY ),
    TUESDAY(   PayType.WEEKDAY ),
    WEDNESDAY( PayType.WEEKDAY ),
    THURSDAY(  PayType.WEEKDAY ),
    FRIDAY(    PayType.WEEKDAY ),
    SATURDAY(  PayType.WEEKEND );

    private static final int HOURS_PER_SHIFT = 8;
    private final PayType type;

    PayrollDay( PayType type ) { this.type = type; }

    public double pay( double hoursWorked, double payRate ) {
        double basePay = hoursWorked * payRate;
        double overtimePay = type.overtimePay( hoursWorked, payRate );

        return basePay + overtimePay;
    }

    private enum PayType {
        WEEKEND { /* omitted. */ }, WEEKDAY { /* omitted. */ };
        public abstract
        double overtimePay( double hoursWorked, double payRate );
    }
}
```

The full details of the inner class are as follows. For simplicty, we shall leave the magic constant 2.

```Java
private enum PayType {
    WEEKEND {
        @Override
        public double overtimePay( double hoursWorked, double payRate ) {
            return hoursWorked * payRate / 2;
        }
    }, WEEKDAY {
        @Override
        public double overtimePay( double hoursWorked, double payRate ) {
            double difference = hoursWorked - HOURS_PER_SHIFT;
            double overtime   = (difference < 0 ? 0 : difference);
            return overtime * payRate / 2;
        }
    };
    public abstract
    double overtimePay( double hoursWorked, double payRate );
}
```

This implementation is clearly better. The overtime pay computation is *what varies*. The strategy enum *isolates* what varies. This *localises* the code for computing the overtime pay. Localising the computation of overtime has the advantage that a global change in the rules for computation may now be implemented by a local change in the Java program. The following demonstrates the advantages.

- It is easy to remove days and strategies by removing existing enum constants.
- It is now possible to change the computation for an existing strategy by making a local change to

the implementation of `overtimePay( )` of that strategy.
- It is now easy to add new days for existing strategies. This may be done by making a local change to the `PayrollDay`. All we need is adding a new `PayrollDay` constant.
- It is also easy to add new days for new strategies. Again this may be done by making local changes. This time we need to add a new `PayrollDay` constant and a new `PayType` strategy constant:

```Java
public enum PayrollDay {
    ...
    BANK_HOLIDAY( PayType.BANK_HOLIDAY ),
    ...
    private enum PayType {
        ...
        BANK_HOLIDAY {
            @Override
            public double overtimePay( double hoursWorked, double payRate ) {
                return hoursWorked * payRate;
            }
        ...
    }
}
```

Notice that this example demonstrates that `enum` classes have separate name spaces. Both `enum` classes, `PayrollDay` and `PayType`, have a constant called `BANK_HOLIDAY` but there's no possibility you can ever mix them up.

10 Use Instance Attributes instead of Ordinals

This section demonstrates that implementing a property with the `enum` `ordinal( )` method may be dangerous.

To understand the problem, consider the following code fragment.

```Don't Try this at Home
public enum Ensemble {
    SOLO,   DUET,   TRIO,  QUARTET, QUINTET,
    SEXTET, SEPTET, OCTET, NONET,   DECTET;

    public int size( ) { return 1 + ordinal( ); }
}
```

This is a typical implementation, which exploits that the size of the ensembles corresponds to the value that's returned by the `ordinal( )` method. This implementation works and it may be difficult to see why it is flawed. However, there are several problems with this approach.

- The first and most obvious flaw is that if the order of the constants changes then the class will break.
- The class will also break if constants are removed (except if they're the last constants).
- The class also breaks if constants are added that create "holes," e.g. if a constant is added for an ensemble with 20 members.
- Finally, the class will break if `enum` constants are added for ensembles with same size as existing ensembles, e.g. a double-quartet.

The number of musicians in an ensemble is really a property of the ensemble. Since ensembles are implemented as `enum` instances, the property should be implemented as an instance attribute.

If we use this approach then we overcome all the previous disadvantages.

```java
public enum Ensemble {
    SOLO( 1 ), DUET( 2 ), TRIO( 3 ), QUARTET( 4 ),
    QUINTET( 5 ), SEXTET( 6 ), SEPTET( 7 ), OCTET( 8 ),
    DOUBLE_QUARTET( 8 ), NONET( 9 ), DECTET( 10 );
    private final int size;

    private Ensemble( final int size ) {
        this.size = size;
    }

    public int size( ) {
        return size;
    }
}
```

This solution solves all the problems with the previous approach.

- This time the order of the constants can be changed without breaking the class.
- In addition constants can be removed without breaking the class.
- Likewise, the class will still work if constants are added.

11 The `EnumSet` Class

This section studies another design anti-pattern. In this case the antipattern is the *bit field* anti-pattern, which is commonly used to represents sets. The pattern exploits that if a set is small enough if can be represented by a small integral number (a bit sequence). If the type of the constant has n bits, the type can represent 2^n values, which is exactly equal to the number of subsets of a set with n members.

Before we look at the example, let's study some bit operators. These operators only work on `int`s and `long`s. They are not examinable.

`lhs << rhs` Shift the `int lhs` to the left by `rhs` bits:[1]
- `(1 << 1) == 2;`
- `(2 << 2) == 8;`
- `(3 << 32) == 3;`

`~operand` Complement of `operand`:
- `(~0) == -1;`
- `(~1) == -2;`
- `(~-1) == 0;`

`lhs & rhs` Bitwise and of `lhs` and `rhs`:
- `(7 & 3) == 3;`
- `(16 & 15) == 0;`
- `(32 & 31) == 0;`

`lhs | rhs` Bitwise or of `lhs` and `rhs`:
- `(7 | 3) == 7;`
- `(4 | 3) == 7;`
- `(32 | 31) == 63;`

[1] Only the last 5 bits of `rhs` are used for the shift operation.

The following example, which is based on [Bloch 2008, Item 32], shows the bit-field anti-pattern.

```Java
public class TextStyle {
    public static final int STYLE_BOLD      = 1 << 0;
    public static final int STYLE_ITALIC    = 1 << 1;
    public static final int STYLE_UNDERLINE = 1 << 2;
    ...
    private int style = 0;

    public void computeUnion( int otherStyle ) {
        style |= otherStyle;
    }

    public void computeDifference( int otherStyle ) {
        style &= ~otherStyle;
    }

    public boolean containsStyle( int otherStyle ) {
        return otherStyle == (style & otherStyle);
    }
}
```

The style of the text is represented as the bitwise or of the int constants. Since each constant is represented using a different bit, this approach works and it has been used for years by programmers that know the trick.

The following are some disadvantages of the pattern.

- This design has all the disadvantages of the bit-enum anti-pattern.
- You cannot use this technique to represent sets with more than 32 members.

Fortunately, there is an easy solution to representing sets consisting of enumerated constants: the `EnumSet` class. The following shows the first part of the solution.

```Java
import java.util.*;

public class TextStyle {
    public enum Style { BOLD, ITALIC, UNDERLINE }
    private EnumSet<Style> style
        = EnumSet.copyOf( new HashSet<Style>( ) );

    ...
}
```

The only difficulty with the `EnumSet` class is that creating instances requires an existing set or the members that you want to add to the set. In the previous implementation we create an empty set by first creating an empty `HashSet` instance and then make a copy of that set using the class method `EnumSet.CopyOf( )`.

The following is the rest of the solution.

```java
public void computeUnion( EnumSet<Style> otherStyle ) {
    // addAll inherited from Set
    style.addAll( otherStyle );
}

public void computeDifference( EnumSet<Style> otherStyle ) {
    // removeAll inherited from AbstractSet
    style.removeAll( otherStyle );
}

public boolean containsStyle( EnumSet<Style> otherStyle ) {
    // containsAll inherited from AbstractCollction.
    return style.containsAll( otherStyle );
}
```

12 For Friday

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

13 Acknowledgements

This lecture is partially based on [Bloch 2008, Item 30]. This lecture is also based on the Java API documentation.

References

Bloch, Joshua [2008]. *Effective Java*. Addison–Wesley. ISBN: 978-0-321-35668-0.