

# Software Development (CS2500)

Lecture 45: More Productivity with enums and Iterators

M.R.C. van Dongen

February 11, 2011

## Contents

1	Outline	1
2	Improvement	2
3	Strategy Enums	3
3.1	A First Stab . . . . .	3
3.2	Strategy Enum . . . . .	4
4	Using Attributes	6
5	Iterators	7
5.1	The Iterable Interface . . . . .	7
5.2	The Iterator Interface . . . . .	8
5.3	Implementation . . . . .	8
5.4	Anonymous Classes . . . . .	9
5.5	Using Anonymous Classes . . . . .	10
6	For Monday	10
7	Acknowledgements	10
8	Bibliography	10

## 1 Outline

This lecture studies *strategy enums* and *Iterators*. At the end it discusses *anonymous classes*. The first part of this lecture is based on [Bloch, 2008, Items 30–31]. This includes examples. Some of this lecture is based on the Java API documentation.

## 2 Improvement

Last Wednesday we implemented our `Operation` class as follows.

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

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

Looking back at it, we can see that it isn't that pretty at all. There still is a lot of repetition in the code: all overrides of `toString( )` are identical (up to a symbol which 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.

```
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; }
}
```

### 3 Strategy Enums

In this section we shall study some more examples of specific behaviour for enum constants. We shall start with a payroll example. Throughout the example we shall use `doubles` to represent money. This isn't really a good idea because the rounding which you get with `doubles` is usually too much and makes the results unreliable. Usually, it's better to represent money (and other quantities which require an *exact* representation) using objects which 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 money with `doubles`. If there's time left, we may study the `BigDecimal` class in some other lecture.

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 which depends on their grade. Our application gets the pay rate as its input. Their pay for a given day of the week is given by

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

The base pay is given by  $\text{pay rate} \times \text{hours worked}$ . 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.

#### 3.1 First Stab at Implementation

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

```

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;
    }
}

```

Don't Try this at Home

There's not much wrong with this implementation at first sight. Again the problem is maintenance. What if we add an extra type of day? For example, a Bank Holiday (special kind of Monday). We'd have to change the implementation of `overtimePay()`. The application breaks if we forget to make the change.

### 3.2 Strategy Enum to the Rescue

So, how do we fix the implementation? The idea is that we want the compiler to help 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 which 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 kinds of strategies. They are *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*. By implementing the property as an attribute, the kind of day is *determined* by the attribute: we can *compute* the kind of day from the attribute. The kind of day *determines* the strategy. Therefore, the attribute *determines* the strategy. We could implement our attribute as a boolean: `isWeekday`. This would work now, but, as always, the requirements may change. For example, what if we get double overtime for hours worked on Christmas days? It is probably better to have a strategy which is 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.

```

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.

```

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:

```
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 the fact 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.

## 4 Using Instance Attributes instead of Ordinals

This section demonstrates that using the enum `ordinal()` method to implement a property may lead to serious flaws in your programs. To understand the problem, consider the following code fragment.

```
public enum Ensemble {
    SOLO, DUET, TRIO, QUARTET, QUINTET,
    SEXTET, SEPTET, OCTET, NONET, DECTET;

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

There are several problems with this approach.

- The first and most obvious problem 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 which 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 is really a property which depends on the constants: it should be implemented as an attribute. If we use this approach then we overcome all the previous disadvantages.

```

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;

    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 remains to work if constants are added.

## 5 Iterators

The ability to *iterate* over a collection of things is convenient. For example, using the generalised for-loop we may write.

```

Type[] things = {magic};
for (Type thing : things) {
    {use thing}
}

```

Generalised for loops work for *any* kind of array.

Many classes from the Java collections also allow the notation.

```

ArrayList<Type> things = new ArrayList<Type>( );
{magic};
for (Type thing : things) {
    {use thing}
}

```

But ArrayLists aren't arrays. So how does this work? The following section shows how.

### 5.1 The Iterable Interface

The ArrayList class implements the Iterable interface. To implement the interface you only have to do one thing: override `Iterator iterator( )`. Iterator is a generic class, just like ArrayList, so it is parameterised over a different type which is written in angled brackets ('<' and '>' after Iterator). Suppose you write the following:

```
ArrayList<String> strings = <magic>;
for (String str : strings) {
    // Use string.
}
```

Java

Java translates this to:

```
ArrayList<String> strings = <magic>;
Iterator<String> iterator = strings.iterator( );
while (iterator.hasNext( )) {
    String string = iterator.next( );
    // Use string.
}
```

Java

## 5.2 The Iterator Interface

The following are the public methods of the `Iterator` interface. The `E` in the types below is the type of the things “in” the `Iterator`, so if you write `Iterator<String>`, then `E` is equal to `String`.

**boolean hasNext( ):** Returns true if there are more elements.

**E next( ):** Returns the next element in the iteration.

**void remove( ):** Removes the last element returned by `next( )`. This method is optional, i.e. there’s no need to override it. The contract of the `Iterator` class is that you can only have one call to `remove( )` after the most recent call to `next( )`. The idea is that you can only remove an object from its current (`next( )`) position.

## 5.3 Implementing Iterable and Iterator

Let’s implement a naive class called `DVDCollection` which stores the names of dvd names which are represented as a `String`. For simplicity we can only construct `DVDCollection` objects and iterate over their underlying collection.

The following are the main details of the class. The details of the inner `DVDIterator` class are presented in the following listing.



```

import java.util.Iterator;

public class DVDCollection implements Iterable<String> {
    private String[] dvds;

    public DVDCollection( String[] dvds ) {
        this.dvds = dvds;
    }

    public Iterator<String> iterator( ) {
        return new DVDIterator( );
    }

    private class DVDIterator implements Iterator<String> {
        // Omitted
    }
}

```

The following is the inner class.

```

private class DVDIterator implements Iterator<String> {
    private int index = 0;

    @Override
    public boolean hasNext( ) {
        return index < dvds.length;
    }

    @Override
    public String next( ) {
        return dvds[ index ++ ];
    }

    @Override
    public void remove( ) {
        String[] newDvds = new String[ dvds.length - 1 ];
        for (int dest = 0; dest < index - 1; dest ++ ) {
            newDvds[ dest ] = dvds[ dest ];
        }
        for (int source = index; source < dvds.length; source ++ ) {
            newDvds[ source - 1 ] = dvds[ source ];
        }
        dvds = newDvds;
    }
}

```

## 5.4 Anonymous Classes

In our DVDCollection class the DVDIterator class was responsible for creating the Iterator. Java also allows for a different technique. This involves an *anonymous* class.

The following shows the technique. The anonymous starts with the brace at the end of the line with new and ends at the brace on the second-last line.

```

public Iterator<String> iterator( ) {
    return new Iterator<String>( ) {
        private int index = 0;
        @Override
        public boolean hasNext( ) {
            return index < dvds.length;
        }
        @Override
        public String next( ) {
            return dvds[ index ++ ];
        }
        @Override
        public void remove( ) {
            String[] newDvds = new String[ dvds.length - 1 ];
            for (int dest = 0; dest < index - 1; dest ++ ) {
                newDvds[ dest ] = dvds[ dest ];
            }
            for (int source = index; source < dvds.length; source ++ ) {
                newDvds[ source - 1 ] = dvds[ source ];
            }
            dvds = newDvds;
        }
    };
}

```

## 5.5 Using Anonymous Classes

You may create an anonymous class anywhere where an expression is expected. However, anonymous classes are more limited than ordinary classes. They don't have a name. Anonymous classes can't implement multiple interfaces. They cannot simultaneously extend a class and implement an interface. They cannot have new public methods. They can only override methods from their superclass or interface.

## 6 For Monday

Study the lecture notes.

## 7 Acknowledgements

The first part of this lecture is based on [Bloch, 2008, Items 30–31]. Some of this lecture is based on the Java API documentation.

## 8 Bibliography

### References

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