The Enum Type

Pieter van den Hombergh
Richard van den Ham

Fontys Hogeschool voor Techniek en Logistiek

March 1, 2018

Topics

Enum Types

Use case

Use for constants

- Constant values, reasons to define them.
- Traditional way: C:

```
#define BLACK 0x000000
```

or

```
enum {MON,TUE,WED,THU,FRI,SAT,SUN} WEEKDAYS;
```

which is a just a list of named int values.

- Equivalent code in Java:

```
public static final int BLACK = 0x000000;
public static final String DEFAULT_GREETING = "Hello"
```

- In many cases you want to create a *value domain*, like in the c-enum example.

Problems with traditional constants and value domains

- There is no **Type** defined.
- The constants are just named `int`egers or `String`s.
- No Type: no type-safety; You can assign any value to the `int`/`String`, not just the named constants.
- In short, this use is not type safe as in assignment safe.

Java enums

- Addresses the type safeness issues.
- Defines a (fixed) domain with (fixed, final) values.
- Is a class, so can implement interfaces.
- An `enum` typed class is a subclass of `Enum`, and therefor cannot extend other classes, `enum` is a special kind of `class`.
- Can have multiple domain values, and all values are constants (`static final` in Java speak).
  - You can, however, specialize the instances by overwriting methods declared in the `enum` type.
- You cannot create new instances.
  That has already been done between compiler and class loader.
- A enum type declared inside another class behaves as a static inner class, i.e. has NO association with the outer class.

Math op example

The interface:

```
public interface MathOp {
    double apply( double x, double y );
}
```

which is implemented in the enum on the next sheets.
Note that the method to be implemented is NOT implemented in the enum class object, but rather in the specializing instances (PLUS, MINUS, TIMES and DIVIDE), which all MUST implement the `apply(double, double)` method.

source: effective Java, second edition.

Implementing as enum

```
public enum Operation implements MathOp {

    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;
        }
    };
```

Implementation other details

```
    private final String symbol;

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

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

Implementation, helpers

```
// Implementing a fromString method on an enum type - Page 154
private static final Map<String, Operation> stringToEnum
        = new HashMap<String, Operation>();

static { // Initialize map from constant name to enum constant
    for ( Operation op : values() ) {
        stringToEnum.put( op.toString(), op );
    }
}

// Returns Operation for string, or null if string is invalid
public static Operation fromString( String symbol ) {
    return stringToEnum.get( symbol );
}
```

Implementation, use

```
    // Test program to perform all operations on given operands
    public static void main( String[] args ) {
        double x = Double.parseDouble( args[ 0 ] );
        double y = Double.parseDouble( args[ 1 ] );
        for ( Operation op : Operation.values() ) {
            System.out.printf( "%f %s %f = %f%n", x, op, y, op.apply( x, y ) );
        }
    Operation o = Operation.DIVIDE;
    double z = o.apply( 2.0D, 3.0D );
    }
```

Java enum consequences and summary

- Type-safeness as in: defines a true type, with all the advantages that has in a type safe language.
- All instances which are values of the enum domain are created as singletons. These instances are available once the **class loader** is ready with the class.
- The enums can have members, just like normal classes.
- The enum instances can have operations with a possible per value different implementation.
- All fields should be made immutable, but may themsleves be mutable. Example you have an enum used as registry and want singleton poperties. That registry could use a `Map` reffered by a final field.
- An enum class is a final class, so you cannot extend the domain (adding enum values) without recompiling the enum class file and the **client** application.

Questions and links

Questions?

Questions or remarks?