COMP6700/2140 Enums

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

March 2017

Topics

1. Global constants in *Java*
2. Typeless interfaces
3. Private Constructor
4. A Gem: *Enum* class before `enums`
5. `enum` Types and their properties
6. Elegance in Practice

Enumerations and `enum`

Lecturettes

COMP6700

Joshua Bloch, inventor of *Java*'s `enums`

Joshua Bloch — an elegant programmer

- *Sun Microsystems*, Senior software designer, till 2004
- *Google Inc*, Chief Java Architect, till 2012
- ??? couldn't find (perhaps he's sold his options and enjoys early retirement)

Key developer of *Java Collection Framework*, (co-)author of award-winning *The Effective Java* (two editions), *The Java Puzzlers* (with Neal Gafter), the man behind Java's `enum` feature (holds a patent).

Problems which call for Enumeration Types

The need to represent a finite set of values in a program is real. Almost always, these values should have common characteristics which are attributed to a type — this has benefits of simplicity of programmatic use and safety. It is not always achievable. In language which do provide support for such artefacts they are called *enums*.

In some programming languages, enums are nothing more than a set of named integer values, but in *Java* an *enum* is a special kind of class, with an instance that represents each value of the *enum*.

The `enum` types help to solve the following problems:

- How to introduce global constants?
- How to make constants type-safe ("dimensionality")?
- How to add object-semantics to constants?

Enumerated "types" in the C Programming Language

The language C has the construct enum to define an *enumerated type*: a type whose legal values consist of a fixed set of constants. This construct is widely used, but as many things in C, it can be a source of bugs:

```c
typedef enum {TOMATO, ORANGE, APPLE} juice_t;
typedef enum {VODKA, WHISKEY, RUM} spirit_t;
typedef enum {NUMBER_ONE, NUMBER_TWO, NUMBER_THREE} coctail_t;
coctail_t x = (VODKA + TOMATO)/2; /* Bloody-Mary! */
juice_t myFavourite = x; /* not driving tonight? */
```

(*Clang* compiler may issue a warning for between different enum type conversions.)

The C enums also do not create a name space for the constants; after the above, one can to put in the following:

```c
typedef enum {ORANGE, APPLE, PEARS} fruit_t;
```

juice_t and fruit_t constants conflict, potential havoc ensues.

Object-Oriented languages can do better.

`interface` as Global Constants Holder

In an "pure" OO language like *Java* enums can be defined as a part of the type declaration: `static final` fields in a (public) class or interface can be used as global constants.

Types in *Java* can be defined in three ways: concrete classes, abstract classes (partial implementation) and interfaces (pure contract). Use of classes involves their other attributes (varying fields and methods), which makes the whole declaration to lack focus.

The main purpose of an interface is to define behaviour. The type defined in an interface can be used to refer to instances of the class, which implement that interface. **But**, *Java* also allows to define an interface *without* any behaviour, *typeless*, which allow to use them as placeholders of global constant values:

```
public interface PhysicsConstants {
    static final double PLANCK_CONSTANT = ...;
    static final double ELECTRON_MASS = ...;
    static final double ELECTRON_CHARGE = ...;
        ... ... ...
}
```

This is not type ("dimensionality") safe: only run-time will reveal the nonsense of

```
ELECTRON_MASS == ELECTRON_CHARGE; // comparing charge and mass
ELECTRON_MASS + ELECTRON_CHARGE; // performing an (arithmetic) operation
```

Typesafe "constants"

Even without falling for such "obvious" errors (add in the lack of namespace protection, see an example in bad_enums), the approach is ridden with design and maintenance problems:

- constant's value does not translate into its meaningful name, size of the group is not known to the client, iteration over the constants is also impeded;
- use of constants is an implementation detail which leaks into the implementing classes;
- its use by clients has nothing to do with the type behaviour, and is confusing;
- worse still, constant interface represents a commitment: future class modifications which do not use the constants still require implementation for binary compatibility (design becomes ugly and illogical).

One solution for *constants problem* is to put them into a utility class with no public constructors:

```
public final class PhysicsConstants {
    private PhysicalConstants() { ... } //instantiation impossible
    public static final double PLANCK_CONSTANT = ...;
        .............
}
```

Most of the interface-based approach shortcomings are still present. When the constant values need to be, say, strings, the string comparisons (needed in all sorts of conditional statements) will incur performance cost, and there is still no compile time check on misspelled constant names.

Private constructor

Better solution is achieved through the use of *type safe enumeration* pattern for class definition. For example, the type which holds four suits of cards can be declared as such class as follows:

```
public class Suit {
    private final String name;
    private Suit(String name) { this.name = name; } // private constructor again!
    public String toString()  { return name; }
    public static final Suit CLUBS    = new Suit("clubs");
    public static final Suit DIAMONDS = new Suit("diamonds");
    public static final Suit HEARTS   = new Suit("hearts");
    public static final Suit SPADES   = new Suit("spades");
}
private static final Suit[] PRIVATE_VALUES = { CLUBS, DIAMONDS, HEARTS, SPADES };
```

When defined like this, there is no way for clients to create objects of the class or to extend it, there will never be any objects of the type besides those exported via the public static final fields (type safety not available via typeless interfaces). Even though the class is not declared final, there is no way to extend it: Subclass constructors must invoke a superclass constructor, and no such constructor is accessible.

This *very elegant trick* by Joshua Bloch is a classic example when a successful solution to a recurring problem (software designers call it a *design pattern*), can be promoted to a language feature (not every design pattern can be implemented like this). The feature which implements the type safe enumerations is the *enum*.

Enumeration types: `enum`

`enum`'s are the types, all instances of which are known when the type is defined:

```
enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES, } // named instances
```

It is a *class disguised as a new language feature*. In a simple form like above, the usage of *Suit* is similar to the constant's interface (without the negatives). But *enum*s can be used with much greater power (this *is* a class!). You can declare constructor and methods inside *enum*:

```
public enum Chess {
  PAWN(1), BISHOP(10), KNIGHT(15), ROOK(20), QUEEN(100); // named constructors
  Chess(int value) { this.value = value; } // implicit private constructor
  private final int value; // constant value tied to enum instance
  public int value() { return value; }
  public String printValue() {return value + ((value > 1) ? " points" : " point");}
}
```

- enum types guarantee type safety (they are uniquely defined types)
- enums can be put into *Collection*s (they are objects)
- enums can be used as `switch` keys (since *JDK 7*, also *String*) — demo ChessTest.java.
- enum cannot extend another class (because it implicitly extend *Enum* abstract class)
- enum type has two static methods: `values()` returns an array of all the values (in the declaration order) and `valueOf(String)` method which translates a constant's *name* into the *constant itself*

enum: a Simple Construct of Great Beauty

The Chess example (the source code: Chess.java and ChessTest.java):

```
public class ChessTest {
    public static void main(String[] args) {
        System.out.println("You don't play a chess game to count points, but:");
        for (Chess c : Chess.values())
            System.out.println(c + " is worth "
                + c.printValue() + ", and it's " + appearance(c));
    }
    private enum ChessStature { SHORT, MEDIUM, TALL } // this is inner enum type
    private static ChessStature appearance(Chess c) {
        switch(c) { // c can be only Number or enum
          case PAWN:    return ChessStature.SHORT;
          case BISHOP:
          case KNIGHT:
          case ROOK:    return ChessStature.MEDIUM;
          case QUEEN:   return ChessStature.TALL;
          default: throw new AssertionError("Unknown figure: " + c);
        }
    }
}
```

Demo run to output.

An example: feel the elegance

Another example is Planets.java from *Java Tutorial*:

```
public enum Planets {
   MERCURY (3.303e+23, 2.4397e6),  VENUS   (4.869e+24, 6.0518e6),
   EARTH   (5.976e+24, 6.37814e6), MARS    (6.421e+23, 3.3972e6),
   JUPITER (1.9e+27,   7.1492e7),  SATURN  (5.688e+26, 6.0268e7),
   URANUS  (8.686e+25, 2.5559e7),  NEPTUNE (1.024e+26, 2.4746e7),
   PLUTO   (1.27e+22,  1.137e6);

   private final double mass;   // in kilograms
   private final double radius; // in meters
   Planets(double mass, double radius) {
       this.mass = mass;
       this.radius = radius;
   }
   private double mass()   { return mass; }
   private double radius() { return radius; }
   public static final double G = 6.67300E-11; // Newton's constant (m3 kg-1 s-2)
   double surfaceGravity() {
       return G * mass / (radius * radius);
   }
```

Planets (concluding)

```
// concluding from the previous slide (enum Planets)
   double surfaceWeight(double otherMass) {
       return otherMass * surfaceGravity();
   }
   public static void main(String[] args) {
       double earthWeight = Double.parseDouble(args[0]);
       double mass = earthWeight/EARTH.surfaceGravity();
       for (Planets p : Planets.values())
          System.out.printf("Your weight on %s is %f%n", p, p.surfaceWeight(mass));
   }

   % java Planets 75
   Your weight on MERCURY is 28.331821
   Your weight on VENUS is 67.874932
   Your weight on EARTH is 75.000000
   Your weight on MARS is 28.405289
   Your weight on JUPITER is 189.791814
   Your weight on SATURN is 79.951165
   Your weight on URANUS is 67.884540
   Your weight on NEPTUNE is 85.374605
   Your weight on PLUTO is 5.015585
```

One more example (from an Assignment I used in yesteryear) Coins.java

More enum Magic

How about making every enum constant behave differently, as if they were different subclasses of an abstract parent class. Thanks to J. Bloch's "The Effective Java" — no "as if", exactly "what the doctor ordered":

```
class VarOpsEnum  {
   public enum Operation { //adding implementation for every enum constant
      PLUS("+")   { double apply(double x, double y) { return x + y; } },
      MINUS("-")  { double apply(double x, double y) { return x - y; } },
      TIMES("*")  { double apply(double x, double y) { return x * y; } },
      DIVIDE("/") { double apply(double x, double y) { return x / y; } };
      private final String symbol;
      Operation(String symbol) { this.symbol = symbol; }
      @Override public String toString() { return symbol; }
      abstract double apply(double x, double y); // declaration of behaviour
   }
   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));
   }
}
```

Where to look for this topic in the textbook?

- Hortsmann's Core Java for the Impatient, Ch. 4.3
- Oracle's Java Tutorial Enum Types