Enumerated Types in Java

Paul A. Cairns

School of Computing Science, Middlesex University, The Burroughs, Hendon,
London, NW4 4BT

e-mail: p.cairns@mdx.ac.uk
tel: +44 (0) 181 362 6852
fax: +44 (0) 181 362 6943

SUMMARY

Java does not contain enumerated types. This paper demonstrates how to produce classes that behave like enumerated types with secure, strong typing. Additional Pascal-like operations are easily incorporated into the enumerated types without repeating code. Full listings of the classes are given.

Keywords: Java, enumerated type, enum

INTRODUCTION

Enumerated types (enums) are a useful tool when programming in high-level languages. They improve the readability of code by associating mnemonic names with constants; they are more efficient because the programmer need not be concerned with the assignment of values and the compiler can optimise their implementation; and they are more secure as a compiler can apply strong type-checking rules to them[1,2,3].

This is not to say that enums are a universal requirement of programming languages. Enums need special treatment by a compiler, which can make a compiler less conceptually simple and as a consequence larger. For this reason they are explicitly omitted from Oberon[4]. However, compiler elegance is an issue orthogonal to those listed above.

Java is intended by its creators to be both easy and safe to use[5]. Thus it is somewhat surprising that Java has no facility for defining enums. Indeed, the absence of enums provokes barely any comment[5,6], which is all the more surprising when investigation of the Java class libraries reveals many lists of `public final static int`'s that could have been defined more efficiently and more securely with enums.

The main aim of this paper is to describe how to define classes that behave like enums and have secure, strong typing. Kalev[7] produced an enum type in a natural way but this does not guarantee that the only constants of the enum type are the enum constants and so the typing in this technique is not secure. A secondary goal is that any errors when using the new enum classes should be made known directly and immediately rather than as side-effects.

A SINGLE ENUMERATED TYPE

The basic example of an enumerated type that we try to reproduce would be given in C as:

```
enum Colour {Red, Green, Blue};
```

This means the `Red`, `Green` and `Blue` are constants (in fact, integers) with distinct values and of the type `Colour`.

The two classes `Palette` and `Colour`, defined in Figure 1 and Figure 2 respectively, achieve the same result. `Palette` acts as a container class for the constants of type `Colour`.

```
import java.util.*;

public final class Palette{
  public static final Colour Red   = Colour.newColour();
  public static final Colour Green = Colour.newColour();
  public static final Colour Blue  = Colour.newColour();
}
```

Figure 1 - The class `Palette`

```
import java.io.*;
import java.util.*;

public final class Colour{
  private final static int ENUM_SIZE = 3;
  private static int       fNumMade  = 0;

  private Colour(){}

  public static Colour newColour(){
    Colour lNextElement;

    if( fNumMade < ENUM_SIZE ){
      lNextElement = new Colour();
      fNumMade++;
    }
    else{
      System.out.println( "All Colours are instantiated" );
      lNextElement = null;
    }

    return lNextElement;
  }
}
```

Figure 2 - The class `Colour`*

The constructor of class `Colour` is private so new instances of `Colour` can only be created through the public method `newColour()`. This method restricts the number of instances of `Colour` that can be made to the maximum of `ENUM_SIZE`, in this case three. As long as Palette instantiates three `Colours`, there can be no other objects of class `Colour` and the strong-typing is secure.

There is much room for expanding on these basic classes. By requiring appropriate parameters in `newColour`, it is possible to associate meaningful data with a constant such as

a string for `toString()` conversions and debugging and also an integer value, for example, `Red` could hold the value `0xFF0000`, its RGB value.

Of course, the proviso that `Palette` creates all possible instances of `Colour` is non-trivial to enforce. The constants are created in a separate class from the `ENUM_SIZE` and could get out of step. If too many are created by `Palette` then `newColour()` alerts the programmer via the system console. If too few, then the program is open to corruption by an unexpected agent creating an invalid `Colour`[8]. This problem is addressed in the next section.

The system console is used to report errors because of the strict way in which Java traps exceptions. All calls to functions that may throw exceptions must be surrounded by exception catching routines. Such routines cannot be part of a `final static` declaration. Thus, if `newColour()` were to throw exceptions there would have to be some call to the `Palette` class to create the constants before they could be used. This is not necessary with the current classes and there is the added bonus that `Palette` has a particularly simple form.

A further drawback with these classes is that the constants cannot be used as case identifiers in `switch` statements but, as Java does not allow anything other than constants of a fundamental type as case identifiers, there is no way around this.

PROLIFERATING ENUMERATED TYPES

In any application of reasonable size, there are generally more than one enumerated types. Using the above technique, each such type requires two public classes and so two corresponding files in the project. Kalev believes that even one extra file per enum is worse than having no enums at all, but experience[3] and experiment[9] indicate that the advantages of strong-typing are not to be dismissed lightly. We contend therefore that the advantages of having enums outweighs the disadvantages of having two files to define them.

However, it is good practice to reduce repeating code. The obvious solution would be to use inheritance and make a general super-class for all enumerator constants that handles how many of each constant type can be made. This does not work because the super-classes constructor must be `protected` or `public` for the sub-class to be created. But as soon as this is done, the sub-class's constructor also becomes `public` to the project containing its definition. Thus, even when the sub-class's constructor is declared `private`, it is possible to directly create instances of the sub-class and so have invalid instances.

```
import java.io.*;
import java.util.*;

public class Registrar{
  private int       fNumMade = 0;
  private String    fClassName;
  private Hashtable fConstTable  = new Hashtable();

  public Registrar( String pName ){
    fClassName  = pName;
  }

  public void Register( Object pNewEl ){
    fConstTable.put( new Integer( fNumMade++ ), pNewEl );
  }

  private class EnumerateEnumElement implements Enumeration{
    private int fCurrentItem = 0;

    public boolean hasMoreElements(){ return(fCurrentItem < fNumMade ); }

    public Object nextElement(){
       Integer lIndex = new Integer( fCurrentItem++ );

       return fConstTable.get( lIndex );
    }
  }

  public Enumeration elements(){
    return new EnumerateEnumElement();
  }

  //other enum features...
}
```

Figure 3 - The class `Registrar`

Instead, the new class `Registrar`, defined in Figure 3, controls the creation of instances.

```
import java.util.*;

public class Palette{
  private static final Registrar fReg = new Registrar( "Palette" );
  public static final Colour Red   = Colour.newColour( fReg );
  public static final Colour Green = Colour.newColour( fReg );
  public static final Colour Blue  = Colour.newColour( fReg );

  public static Enumeration elements() { return fReg.elements(); }
}
```

Figure 4 - The new definition of `Palette`

The Palette has a private Registrar that it uses to create all the Colours, see Figure 4. The Colour class still has a private constructor but now newColour() simply tries to Register() any new elements with the given Registrar. In order to be consistent, the first Registrar that is used to create a Colour must be used in creating all subsequent Colours. Failure to do so results in an error, see Figure 6. The collaboration required to set up the Palette class is described using UML in .

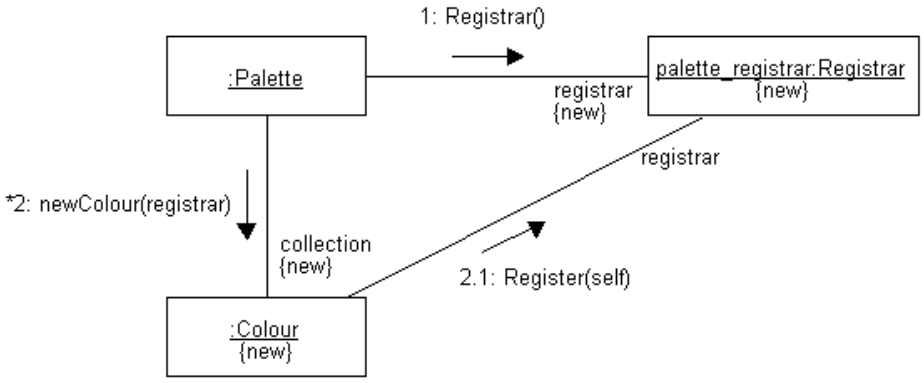

Figure 5 - Collaboration diagram to define Palette

Provided Palette gets in first at creating Colours no-one else can do so as its Registrar is not available to anyone else. This registration is still not foolproof but if someone goes to all the effort of creating a Registrar to pass to Colour before Palette does, then they obviously do not want their program to work! Even in this case, when Palette comes to be used, error messages are generated.

```
public final class Colour{
  private static Registrar fRegistrar = null;

  private Colour(){}

  public static Colour newColour( Registrar pReg ) {
    Colour lNewElement = null;

    if( fRegistrar == null) fRegistrar = pReg;

    if( fRegistrar == pReg ) {
       lNewElement = new Colour();
       pReg.Register( lNewElement );
    }
    else
       System.out.println( "Invalid registrar for Colour" );

    return lNewElement;
  }
}
```

Figure 6 - The new definition of Colour

Note, with this technique the number of constants in the enum does not need to be explicitly stated thus avoiding the possibility of the number of constants created getting out of step with the expected size of the enum.

The `Registrar` class comes into its own in providing more sophisticated enum operations such as those supported by Pascal. Here `Registrar` can create an `Enumeration` object for the enum so that the enum can be treated as an ordered list of constants. The `Enumeration` is made public via the `Palette` class. In a similar way, it would be possible to have methods corresponding to the `Pred` and `Succ` methods of Pascal.

To create more enumerated types is now a simple exercise. The modifications to `Palette` should be obvious. As for `Colour`, it is simply a matter of replacing all references to `Colour` with references to the new type. This is a self-contained task, which throws up compiler errors if not done correctly.

Sub-ranges are another way of employing enums. With these, the original list of constants is restricted to some special subset, for example, `Colour` might have the sub-range `NotBlue` made up of only `Red` and `Green`. Cardelli proposes a method of defining sub-ranges using inheritance[10] however in this scheme the elements of the original enum are not constants but types. In Java, this would translate to one file for every enum constant!

Building on the classes given here, sub-ranges could be crudely defined as shown in . Supporting operations, such as checking for membership of the sub-range and an `Enumeration` only over the sub-range, could also be provided by using a variation on `Registrar`. These are issues away from the aims of this paper and are left as an exercise for the interested reader.

```
import java.util.*;

public final class NotBlue{
  public static final Colour Red   = Palette.Red;
  public static final Colour Green = Palette.Green;
}
```

Figure 7 - The sub-range `NotBlue`

CONCLUSIONS

It is possible to create objects in Java that behave like enumerated types with secure strong-typing. They can even be embellished to support more sophisticated features such as those supported in Pascal and even `toString()` conversions. Moreover, it can be done so that only a small amount of code must be repeated for each new type. Some code must always be repeated as inheritance is not an available technique.

The main problem of course is with the Java language itself[11]. It seems that in eliminating the variety of types available to C and Pascal, Java has thrown the baby out with the bath-water.

ACKNOWLEDGMENTS

Many thanks to Prof. Harold Thimbleby and Dr. Matt Jones for their helpful advice and suggestions.

REFERENCES

1. S. McConnell, *Code Complete*, Microsoft Press, 1993
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd edn., Prentice-Hall, 1988
3. N. Wirth, 'An Assessment of the Programming Language Pascal' in A. Feuer and N. Gehani (eds.), *Comparing and Assessing Programming Languages,* Prentice-Hall, 1984
4. N. Wirth, 'From Modula to Oberon', *Software - Practice and Experience*, 18(7), 661-670 (1988)
5. K. Arnold and J. Gosling, *The Java Programming Language,* 2nd edn., Addison-Wesley, 1998
6. J. Gosling, B. Joy and G. Steele, *The Java Language Specification,* Addison-Wesley, 1996
7. D. Kalev, 'Porting C++ Applications to Java' *C/C++ Users Journal*, 16(2), 73-82 (1998)
8. T. Pratchett, *The Colour of Magic*, Corgi (Paperback), 1983
9. J. D. Gannon, 'An Experimental Evaluation of Data Type Conventions' *CACM,* 20(8), 584-593 (1977)
10. L. Cardelli, 'A Semantics of Multiple Inheritance', Information and Computation 76, 138-164 (1988)
11. H. Thimbleby, 'Java - a critique', *Software - Practice and Experience*, to appear

[*] In the code listings, all variables have prefix of either 'f', 'l' or 'p'. 'f' denotes a field in an object, 'l' denotes a variable local to a method and 'p' denotes a parameter passed to a method. Thus, rather than a more usual

Hungarian notation denoting the type of the variable, the prefix denotes the *scope* of the variable. In object-oriented languages where types abound, the author has found this to be more useful information.