Language Contents

Enums

In prior releases, the standard way to represent an enumerated type was the `int` Enum pattern:

```
// int Enum Pattern - has severe problems!
public static final int SEASON_WINTER = 0;
public static final int SEASON_SPRING = 1;
public static final int SEASON_SUMMER = 2;
public static final int SEASON_FALL   = 3;
```

This pattern has many problems, such as:

- **Not typesafe** - Since a season is just an `int` you can pass in any other `int` value where a season is required, or add two seasons together (which makes no sense).
- **No namespace** - You must prefix constants of an int enum with a string (in this case `SEASON_`) to avoid collisions with other int enum types.
- **Brittleness** - Because int enums are compile-time constants, they are compiled into clients that use them. If a new constant is added between two existing constants or the order is changed, clients must be recompiled. If they are not, they will still run, but their behavior will be undefined.
- **Printed values are uninformative** - Because they are just ints, if you print one out all you get is a number, which tells you nothing about what it represents, or even what type it is.

It is possible to get around these problems by using the *Typesafe Enum* pattern (see *Effective Java* Item 21), but this pattern has its own problems: It is quite verbose, hence error prone, and its enum constants cannot be used in `switch` statements.

In 5.0, the Java™ programming language gets linguistic support for enumerated types. In their simplest form, these enums look just like their C, C++, and C# counterparts:

```
enum Season { WINTER, SPRING, SUMMER, FALL }
```

But appearances can be deceiving. Java programming language enums are far more powerful than their counterparts in other languages, which are little more than glorified integers. The new `enum` declaration defines a full-fledged class (dubbed an *enum type*). In addition to solving all the problems mentioned above, it allows you to add arbitrary methods and fields to an enum type, to implement arbitrary interfaces, and more. Enum types provide high-quality implementations of all the `Object` methods. They are `Comparable` and `Serializable`, and the serial form is designed to withstand arbitrary changes in the enum type.

Here is an example of a playing card class built atop a couple of simple enum types. The `Card` class is immutable, and only one instance of each `Card` is created, so it need not override `equals` or `hashCode`:

```
import java.util.*;

public class Card {
    public enum Rank { DEUCE, THREE, FOUR, FIVE, SIX,
        SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE }

    public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }

    private final Rank rank;
    private final Suit suit;
    private Card(Rank rank, Suit suit) {
        this.rank = rank;
        this.suit = suit;
    }

    public Rank rank() { return rank; }
    public Suit suit() { return suit; }
    public String toString() { return rank + " of " + suit; }

    private static final List<Card> protoDeck = new ArrayList<Card>();

    // Initialize prototype deck
    static {
        for (Suit suit : Suit.values())
            for (Rank rank : Rank.values())
                protoDeck.add(new Card(rank, suit));
    }

    public static ArrayList<Card> newDeck() {
        return new ArrayList<Card>(protoDeck); // Return copy of prototype deck
    }
}
```

The `toString` method for `Card` takes advantage of the `toString` methods for `Rank` and `Suit`. Note that the `Card` class is short (about 25 lines of code). If the typesafe enums (`Rank` and `Suit`) had been built by hand, each of them would have been significantly longer than the entire `Card` class.

The (private) constructor of `Card` takes two parameters, a `Rank` and a `Suit`. If you accidentally invoke the constructor with the parameters reversed, the compiler will politely inform you of your error. Contrast this to the `int` enum pattern, in which the program would fail at run time.

Note that each enum type has a static `values` method that returns an array containing all of the values of the enum type in the order they are declared. This method is commonly used in combination with the for-each loop to iterate over the values of an enumerated type.

The following example is a simple program called `Deal` that exercises `Card`. It reads two numbers from the command line, representing the number of hands to deal and the number of cards per hand. Then it creates a new deck of cards, shuffles it, and deals and prints the requested hands.

```
import java.util.*;

public class Deal {
    public static void main(String args[]) {
        int numHands = Integer.parseInt(args[0]);
        int cardsPerHand = Integer.parseInt(args[1]);
        List<Card> deck  = Card.newDeck();
        Collections.shuffle(deck);
        for (int i=0; i < numHands; i++)
            System.out.println(deal(deck, cardsPerHand));
    }

    public static ArrayList<Card> deal(List<Card> deck, int n) {
         int deckSize = deck.size();
         List<Card> handView = deck.subList(deckSize-n, deckSize);
         ArrayList<Card> hand = new ArrayList<Card>(handView);
         handView.clear();
         return hand;
     }
}

$ java Deal 4 5
[FOUR of HEARTS, NINE of DIAMONDS, QUEEN of SPADES, ACE of SPADES, NINE of SPADES]
[DEUCE of HEARTS, EIGHT of SPADES, JACK of DIAMONDS, TEN of CLUBS, SEVEN of SPADES]
[FIVE of HEARTS, FOUR of DIAMONDS, SIX of DIAMONDS, NINE of CLUBS, JACK of CLUBS]
[SEVEN of HEARTS, SIX of CLUBS, DEUCE of DIAMONDS, THREE of SPADES, EIGHT of CLUBS]
```

Suppose you want to add data and behavior to an enum. For example consider the planets of the solar system. Each planet knows its mass and radius, and can calculate its surface gravity and the weight of an object on the planet. Here is how it looks:

```
public enum Planet {
    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
    Planet(double mass, double radius) {
```

```
        this.mass = mass;
        this.radius = radius;
    }
    public double mass()   { return mass; }
    public double radius() { return radius; }

    // universal gravitational constant  (m^3 kg^-1 s^-2)
    public static final double G = 6.67300E-11;

    public double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    public double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
}
```

The enum type `Planet` contains a constructor, and each enum constant is declared with parameters to be passed to the constructor when it is created.

Here is a sample program that takes your weight on earth (in any unit) and calculates and prints your weight on all of the planets (in the same unit):

```
    public static void main(String[] args) {
        double earthWeight = Double.parseDouble(args[0]);
        double mass = earthWeight/EARTH.surfaceGravity();
        for (Planet p : Planet.values())
           System.out.printf("Your weight on %s is %f%n",
                             p, p.surfaceWeight(mass));
    }
```

```
$ java Planet 175
Your weight on MERCURY is 66.107583
Your weight on VENUS is 158.374842
Your weight on EARTH is 175.000000
Your weight on MARS is 66.279007
Your weight on JUPITER is 442.847567
Your weight on SATURN is 186.552719
Your weight on URANUS is 158.397260
Your weight on NEPTUNE is 199.207413
Your weight on PLUTO is 11.703031
```

The idea of adding behavior to enum constants can be taken one step further. You can give each enum constant a *different* behavior for some method. One way to do this by switching on the enumeration constant. Here is an example with an enum whose constants represent the four basic arithmetic operations, and whose `eval` method performs the operation:

```
public enum Operation {
    PLUS, MINUS, TIMES, DIVIDE;
```

```
        // Do arithmetic op represented by this constant
        double eval(double x, double y){
            switch(this) {
                case PLUS:   return x + y;
                case MINUS:  return x - y;
                case TIMES:  return x * y;
                case DIVIDE: return x / y;
            }
            throw new AssertionError("Unknown op: " + this);
        }
    }
```

This works fine, but it will not compile without the `throw` statement, which is not terribly pretty. Worse, you must remember to add a new case to the `switch` statement each time you add a new constant to `Operation`. If you forget, the `eval` method with fail, executing the aforementioned `throw` statement

There is another way give each enum constant a different behavior for some method that avoids these problems. You can declare the method abstract in the enum type and override it with a concrete method in each constant. Such methods are known as *constant-specific* methods. Here is the previous example redone using this technique:

```
public enum Operation {
  PLUS   { double eval(double x, double y) { return x + y; } },
  MINUS  { double eval(double x, double y) { return x - y; } },
  TIMES  { double eval(double x, double y) { return x * y; } },
  DIVIDE { double eval(double x, double y) { return x / y; } };

  // Do arithmetic op represented by this constant
  abstract double eval(double x, double y);
}
```

Here is a sample program that exercises the `Operation` class. It takes two operands from the command line, iterates over all the operations, and for each operation, performs the operation and prints the resulting equation:

```
    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.eval(x, y));
    }
```

```
$ java Operation 4 2
4.000000 PLUS 2.000000 = 6.000000
4.000000 MINUS 2.000000 = 2.000000
4.000000 TIMES 2.000000 = 8.000000
4.000000 DIVIDE 2.000000 = 2.000000
```

Constant-specific methods are reasonably sophisticated, and many programmers will never need to use them, but it is nice to know that they are there if you need them.

Two classes have been added to `java.util` in support of enums: special-purpose `Set` and `Map` implementations called `EnumSet` and `EnumMap`. `EnumSet` is a high-performance `Set` implementation for enums. All of the members of an enum set must be of the same enum type. Internally, it is represented by a bit-vector, typically a single `long`. Enum sets support iteration over ranges of enum types. For example given the following enum declaration:

```
enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY }
```

you can iterate over the weekdays. The `EnumSet` class provides a static factory that makes it easy:

```
for (Day d : EnumSet.range(Day.MONDAY, Day.FRIDAY))
    System.out.println(d);
```

Enum sets also provide a rich, typesafe replacement for traditional bit flags:

```
EnumSet.of(Style.BOLD, Style.ITALIC)
```

Similarly, `EnumMap` is a high-performance `Map` implementation for use with enum keys, internally implemented as an array. Enum maps combine the richness and safety of the `Map` interface with speed approaching that of an array. If you want to map an enum to a value, you should always use an `EnumMap` in preference to an array.

The `Card` class, above, contains a static factory that returns a deck, but there is no way to get an individual card from its rank and suit. Merely exposing the constructor would destroy the singleton property (that only a single instance of each card is allowed to exist). Here is how to write a static factory that preserves the singleton property, using a nested `EnumMap`:

```
private static Map<Suit, Map<Rank, Card>> table =
    new EnumMap<Suit, Map<Rank, Card>>(Suit.class);
static {
    for (Suit suit : Suit.values()) {
        Map<Rank, Card> suitTable = new EnumMap<Rank, Card>(Rank.class);
        for (Rank rank : Rank.values())
            suitTable.put(rank, new Card(rank, suit));
        table.put(suit, suitTable);
    }
}

public static Card valueOf(Rank rank, Suit suit) {
    return table.get(suit).get(rank);
}
```

The `EnumMap` (`table`) maps each suit to an `EnumMap` that maps each rank to a card. The lookup performed by the `valueOf` method is internally implemented as two array accesses, but the code is much clearer and safer. In order to preserve the singleton property, it is imperative that the constructor invocation in the prototype deck initialization in `Card` be replaced by a call to the new static factory:

```
// Initialize prototype deck
static {
    for (Suit suit : Suit.values())
        for (Rank rank : Rank.values())
            protoDeck.add(Card.valueOf(rank, suit));
}
```

It is also imperative that the initialization of `table` be placed above the initialization of `protoDeck`, as the latter depends on the former.

So when should you use enums? Any time you need a fixed set of constants. That includes natural enumerated types (like the planets, days of the week, and suits in a card deck) as well as other sets where you know all possible values at compile time, such as choices on a menu, rounding modes, command line flags, and the like. It is *not* necessary that the set of constants in an enum type stay fixed for all time. The feature was specifically designed to allow for binary compatible evolution of enum types.