Iterations

- in Java 1.4

```
List strings;
...
for(int i = 0; i < strings.size(); i++) {
    String str = strings.elementAt(i);
    System.out.println(str);
}
```

or better using an iterator

```
for(Iterator iter = strings.iterator(); iter.hasNext();) {
    String str = (String) iter.next();
    System.out.println(str);
}
```

Iterations (cont’d)

- in Java 1.5

```
List<String> strings;
...
for(String str : strings) System.out.println(str);

int[] vector = new int[100];
...
int sum = 0;
for(int elem : vector) sum += elem;
```

Enumeration types in Java 1.5

- in Java 1.4
  - enumeration types are implemented by using the type `int`

```
public static final int RED    = 0;
public static final int YELLOW       = 1;
public static final int BLUE   = 2;
...
switch(myColor) {
  case Color.RED:    System.out.println(”red”); break;
  case Color.YELLOW: System.out.println(”yellow”); break;
  case Color.BLUE:   System.out.println(”blue”); break;
};
```

Enumeration types (cont'd)

Advantages of explicit enumeration types:

- they are type safe (checked at compile time)
  - `int` enums don't provide any type safety at all
- they provide a proper name space for the enumerated type
  - with `int` enums you have to prefix the constants to get any semblance of a name space
- they are robust
  - `int` enums are compiled into clients, and you have to recompile clients if you add, remove, or reorder constants
- printed values are informative
  - if you print an `int` enum you just see a number
- can be stored in collections (objects)
- arbitrary fields and methods can be added

Enumeration types (cont'd)

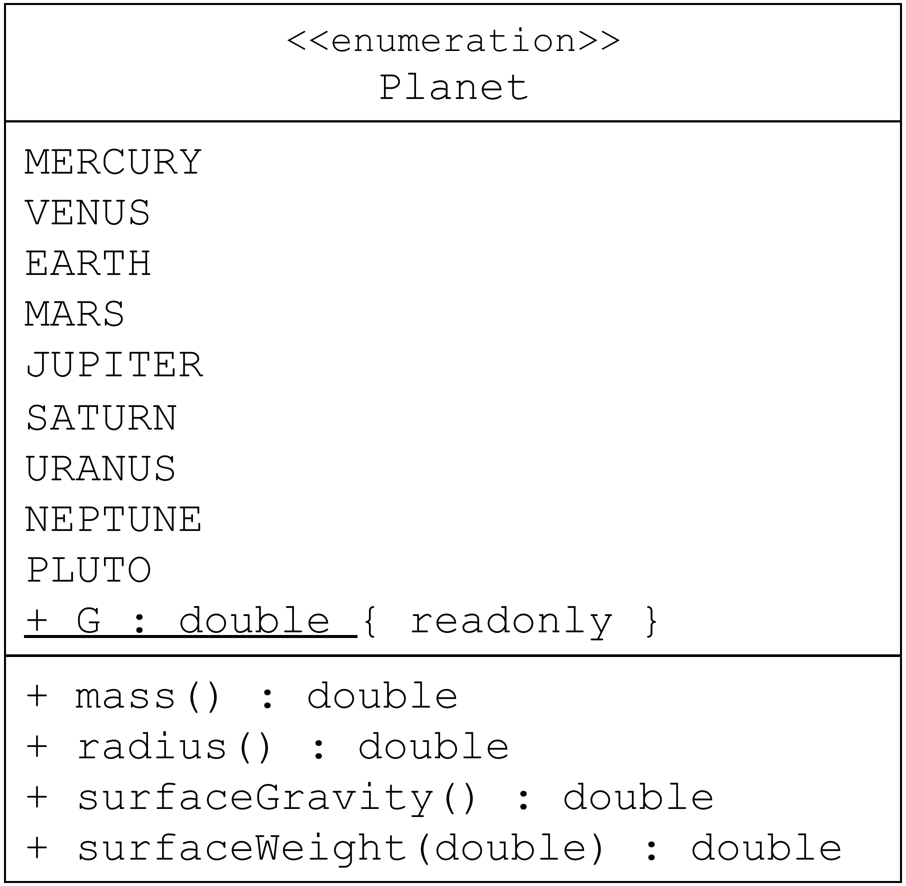
- in Java 1.5

Simple example:

```
public enum Color {RED, YELLOW, BLUE};
...
for (Color myColor : Color.values())
  System.out.println(myColor);
```

`values()` is a static method of an enumeration type returning an array containing all the values of the enum type in the order they are declared.

Enum types - UML class diagram



<<enumeration>>
Planet

MERCURY
VENUS
EARTH
MARS
JUPITER
SATURN
URANUS
NEPTUNE
PLUTO
+ G : double { readonly }

+ mass() : double
+ radius() : double
+ surfaceGravity() : double
+ surfaceWeight(double) : double

```
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 (m3 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(); }
}
```

```
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);
}



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));
}
```