Enhancements in Java 1.5

Generics

- Enhancement to the type system:
  - a type or method can operate on objects of various types with compile-time type safety,
  - compile-time type safety to the Collections Framework and eliminates the drudgery of casting (no `ClassCastException` at run time).
- Improved readability and robustness.

```
// Removes 4-letter words from c. Elements must be
strings
static void expurgate(Collection c) {
  for (Iterator i = c.iterator(); i.hasNext(); )
    if (((String) i.next()).length() == 4)
      i.remove();
}
```

```
// Removes the 4-letter words from c
static void expurgate(Collection<String> c) {
  for (Iterator<String> i = c.iterator(); i.hasNext
(); )
    if (i.next().length() == 4)
      i.remove();
}
```

Enhanced for Loop

- New language construct:
  - eliminates the drudgery and error-proneness of iterators and index variables when iterating over collections and arrays.
- The for-each loop not usable for programs that need access to the iterator:
  - filter elements (generics example),
  - replace elements in a list or array as you traverse it.

```
void cancelAll(Collection<TimerTask> c) {
  for (Iterator<TimerTask> i = c.iterator();
i.hasNext(); )
    i.next().cancel();
}
```

```
void cancelAll(Collection<TimerTask> c) {
  for (TimerTask t : c)
    t.cancel();
}
```

```
// Returns the sum of the elements of a
int sum(int[] a) {
  int result = 0;
  for (int i : a)
    result += i;
  return result;
}
```

Autoboxing/Unboxing

```
Integer value = new Integer(2); // boxing
int i = value.intValue(); // unboxing
```

- This facility eliminates manual conversion between primitive types (such as `int`) and wrapper types (such as `Integer`).
  - add an `int` in a `Vector` ?
  - computational cost.

```
import java.util.*;

// Prints a frequency table of the words on
the command line
public class Frequency {
  public static void main(String[] args) {
    Map<String, Integer> m = new
TreeMap<String, Integer>();
    for (String word : args) {
      Integer freq = m.get(word);
      m.put(word, (freq == null ? 1 : freq +
1));
    }
    System.out.println(m);
  }
}
```

Typesafe Enums

- Create enumerated types with arbitrary methods and fields.
- `int` Enum patterns:
  - not typesafe,
  - no namespace,
  - brittleness: compiled into clients that use them, that must be recompiled,
  - printed values are uninformative.

```
// 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;
```

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

```
public enum Planet {
  MERCURY (3.303e+23, 2.4397e6),
  VENUS   (4.869e+24, 6.0518e6),
  EARTH   (5.976e+24, 6.37814e6),
  ...

  private final double mass;   // in kilograms
  private final double radius; // in meters
  Planet(double mass, double radius) {
    this.mass = mass;
    this.radius = radius;
  }
  // universal gravitational constant  (m3 kg-1 s-2)
  public static final double G = 6.67300E-11;

  double surfaceGravity() {
    return G * mass / (radius * radius);
  }
  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 (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
...
```

Varargs

- This facility eliminates the need for manually boxing up argument lists into an array when invoking methods that accept variable-length argument lists.
- Beware of overloading methods with varargs.

```
Object[] arguments = {
  new Integer(7),
  new Date(),
  "a disturbance in the Force"
};

String result = MessageFormat.format("At {1,time} on
{1,date}, there was {2} on planet {0,number,integer}.
", arguments);
```

```
public static String format(String pattern, Object...
arguments);
```

```
String result = MessageFormat.format("At {1,time} on
{1,date}, there was {2} on planet {0,number,integer}.
", 7, new Date(), "a disturbance in the Force");
```

```
// Simple test framework
public class Test {
  public static void main(String[] args) {
    int passed = 0;
    int failed = 0;
    for (String className : args) {
      try {
        Class c = Class.forName(className);
        c.getMethod("test").invoke(c.newInstance());
        passed++;
      } catch (Exception ex) {
        System.out.printf("%s failed: %s%n",
className, ex);
        failed++;
      }
    }
    System.out.printf("passed=%d; failed=%d%n",
passed, failed);
  }
}
```

Static Import

- This facility lets you avoid qualifying static members with class names without the shortcomings of the "Constant Interface antipattern".

```
double r = Math.cos(Math.PI * theta);
```

```
import static java.lang.Math.PI;
// or
import static java.lang.Math.*;
```

Annotations (Metadata)

- This language feature lets you avoid writing boilerplate code under many circumstances by enabling tools to generate it from annotations in the source code. This leads to a "declarative" programming style where the programmer says what should be done and tools emit the code to do it. Also it eliminates the need for maintaining "side files" that must be kept up to date with changes in source files. Instead the information can be maintained in the source file.