Mengenal JAVA Generics dalam JAVA 1.5 (“Tiger”)

Nanik Tolaram
nanikjava@yahoo.com

Java programming language dengan codename "Tiger" merupakan versi yang akan dirilis oleh SUN Microsystem tahun ini. Visi daripada versi dari "Tiger" ini adalah untuk memberikan kemudahan dalam programming, performance, desktop client dan skalabilitas. Meskipun versi ini belum direlease namun sudah tersedia "early access" daripada fitur-fitur yang akan didapatkan dalam versi tersebut. Salah satu fitur yang akan kita bahas dalam versi 1.5 adalah implementasi daripada JSR yaitu JSR014 (Public Draft 2.0, June 2003), spesifikasi daripada fitur tersebut bisa didownload di http://www.jcp.org/en/jsr/detail?id=014. Mengapa fitur ini diperlukan ? karena kita ketahui bahwa pemrograman dengan Java kita banyak berinteraksi dengan object dan kita memerlukan tempat untuk menyimpan object-object tersebut ke sebuah container seperti Collection, List, ArrayList, dll, dan sudah barang tentu dengan menggunakan Collection objects kita pasti melakukan casting (downcasting) untuk object yang kita akan ambil dari Collection, dan ini menjadi error-prone karena tidak ada jaminan bahwa object yang akan kita retrieve tersebut adalah object yang kita inginkan. Disinilah apa yang sering kita lihat dengan ClassCastException error.

Kita akan membahas komponen-komponen daripada Generics tersebut dan disebutkan dibawah ini,

1. ***Generics*** - Memberikan type safety checking untuk collections untuk menghapus supaya tidak perlu lagi menggunakan class casting untuk pengambilan object.
2. ***Enhanced for loop*** - for loop iterator yang memudahkan untuk meiterate sebuah collection tanpa harus melakukan casting dan tanpa menggunakan object Iterator
3. ***Autoboxing/unboxing*** - Menghapus konversi secara manual dari tipe primitive (int) ke tipe object (Integer)
4. ***Typesafe Enums*** - Memberikan fasilitas Typesafe Enum pattern
5. ***Static import*** - Mengimport class yang mempunyai method static tanpa harus menggunakan full class name pada saat memanggil method yang diinginkan
6. ***Metadata*** - Memberikan kemudahan dalam membuat code yang sering digunakan dengan menggunakan tag, ini juga disebut dengan "declarative" programming style, dimana code-

code yang sering dipakai akan digenerate oleh compiler tanpa kita harus menulis code. Cara ini biasa digunakan oleh tools seperti Xdocklet http://xdoclet.sourceforge.net/).

Untuk memasyarakatkan versi "Tiger" ini, Sun Microsystem, mengeluarkan prototype untuk JSR014 untuk digunakan dengan JDK versi yang tersedia yaitu versi 1.4. Pada saat artikel ini ditulis, versi JDK yang dipakai adalah JDK1.4.2 namun versi JDK1.4 pun bisa dipakai. JAR files yang diperlukan dapat didownload di website Sun, yaitu http://developer.java.sun.com/developer/earlyAccess/adding_generics/. Download file tersebut dan extract filenya, lalu tempatkan file collect.jar dan gjc-rt.jar ke dalam directory <JDK>\jre\lib (sebagai contoh kalau anda punya JDK ada di directory D:\JDK1.4, maka file tersebut dicopy ke dalam directory D:\JDK1.4\JRE\LIB.

Ikuti langkah-langkah dibawah ini untuk mempersiapkan environment anda supaya dapat memakai fasilitas Generics tersebut. Dalam artikel ini file yang didownload diunzip ke dalam directory E:\adding_generics-2_2-ea,

1. Buka file java.bat dan javac.bat yang ada di dalam directory scripts, dan tambahkan script dibawah ini di baris paling pertama

   ```
   set J2SE14=e:\jdk1.4
   set JSR14DISTR=%J2SE14%\jre\lib
   ```

   Path yang diset diatas harap disesuaikan dengan yang ada di PC anda.

2. Masukkan path ke dalam environment variable PATH anda dengan cara sebagai berikut,

   ```
   set PATH=%PATH%;E:\adding_generics-2_2-ea\scripts
   ```

Untuk mecompile program, anda cukup menjalankan javac.bat disertakan nama file anda dan untuk menjalankannya anda menggunakan java.bar.

Generics

Kegunaan dari Generics adalah untuk mencegah terjadinya error seperti ClassCastException pada saat kita menggunakan object yang disimpan dalam sebuah Collection object, sebagai contoh kita ambil cara yang biasa kita lakukan pada sehari-hari kita coding menggunakan Collection,

```
public void iterateCollection(Collection coll)
{
  for (Iterator i=c.iterator(); i.hasNext();)
  {
    String str = (String) i.next();
    System.out.println("Content = " + str);
  }
}
```

dan kalau kita pakai Generics,

```
public void iterateCollection(Collection<String> coll)
{
  for (Iterator<String> i=c.iterator(); i.hasNext();)
  {
```

```
    String str = i.next();
    System.out.println("Content = " + str);
  }
}
```

Jika kita lihat pada saat kita menggunakan Collection seperti biasa kalau salah satu element di dalam Collection tersebut bukan object String maka anda akan mendapatkan error pada saat runtime, tetapi pada saat kompilasi anda tidak akan mendapatkan error, namun jika anda menggunakan Generics anda akan mendapatkan error pada saat kompilasi, ini karena kita sudah memberitahu kepada compiler bahwa Collection yang akan diterima harus berisikan object String, dan jika kalau bukan, maka akan error.

Enhanced for Loop

Setiap kali kita meggunakan Collection object kita secara otomatis kebanyakan akan memakai Iterator untuk mengiterate dan mengambil element yang tersimpan di dalam Collection object tersebut, dan setelah itu kita harus mencast object yang didapatkan ke dalam object yang kita inginkan, belum kalau kita salah mencast object tersebut, maka terjadilah error. Dengan cara lama kita biasa melakukan iterasi seperti dibawah ini,

```
public void iterateCollection(Collection coll)
{
  for (Iterator i=c.iterator(); i.hasNext();)
  {
    String str = (String) i.next();
    System.out.println("Content = " + str);
  }
}
```

dengan menggunakan enhanced for loop kita mengiterasi dengan cara,

```
public void iterateCollection(Collection coll)
{
  for (String str : coll)
  {
    System.out.println("Content = " + str);
  }
}
```

jumlah baris yang kita harus maintain dalam program kita akan jauh lebih sedikit dan juga lebih gampang dibaca dengan menggunakan enhanced for loop ini.

Autoboxing/Unboxing

Tipe primitive seperti *int* tidak dapat disimpan ke dalam *Collection*, container tanpa harus melalui konversi ke wrapper classnya yaitu Integer. Dalam versi 1.5, kita tidak perlu melakukan konversi, hanya cukup deklarasikan apa yang akan diterima oleh Collection. Dibawah ini contoh dengan cara lama,

```
class TestAutoBoxing
{
 public static void main(String[] args)
 {
  Hashtable h = new Hashtable();
  int j = 2;
```

```
    h.put(new Integer(j), "value");
  }
}
```

cara baru kita menggunakan sebagai berikut,

```
class TestAutoBoxing
{
  public static void main(String[] args)
  {
    Hashtable<Integer, String> h = new Hashtable<Integer, String>();
    int j = 2;

    h.put(j, "value");
  }
}
```

seperti kita lihat, kita tidak perlu melakukan konversi yang kita perlukan, compiler secara otomatis akan melakukannya sendiri. Tipe wrapper class (Integer, dll) bisa dipakai untuk deklarasi object yang akan disimpan di dalam Hashtable, tidak dapat menggunakan tipe primitive seperti int,

```
Hashtable<Integer, String> h = new Hashtable<Integer, String>();
                               [ Diperbolehkan ]

Hashtable<int String> h = new Hashtable<int    , String>();
                               [ Tidak diperbolehkan ]
```

Typesafe Enums

Typesafe enum memberikan fasilitas linguistic untuk Typesafe Enum pattern. Caranya mirip dengan bagaimana kita deklarasi enum di C/C++. Pada versi JAVA sekarang ini, kita membuat enum dengan menggunakan pattern dengan cara sebagai berikut (menggunakan Typesafe Enum pattern)

```
public class CoinRupiahEnum {
   private final int coin;

   private CoinRupiahEnum (int coin) { this.coin = coin; }

   public String toString()  { return "" + coin; }

   public static final CoinRupiahEnum
                                LIMAPULUH = new CoinRupiahEnum (50);

   public static final CoinRupiahEnum
                                DUAPULUHLIMA  =  new CoinRupiahEnum(25);

   public static final CoinRupiahEnum
                                SERATUS       =  new CoinRupiahEnum (100);
}
```

dan class CoinRupiahEnum tersebut dapat kita pakai dari class lain dengan CoinRupiahEnum.LIMAPULUH.

Dalam “Tiger” kita membuat enum dengan cara

```
public enum CoinRupiahEnum
{
```

```
  LIMAPULUH(50), DUAPULUHLIMA(25), SERATUS(100);

  CoinRupiahEnum (int Coin)
  {  this. Coin= Coin;  }

  private final int Coin;

  public int Coin()
  {  return Coin;  }
}
```

Kalau kita bandingkan lebih mudah dan lebih praktis kalau kita gunakan Typesafe Enum di Java 1.5

Static Import

Pada umumnya pada saat kita ingin menggunakan static method atau variable dalam suatu class kita menggunakan cara

```
<fully qualified class name>.<method/variable>
```

Sebagai contoh,

```
package Test.Static;

public class TestStatic
{
  public static void printMe()
  {
        System.out.println("Print Me");
  }
}
```

Static method kita dalam package Test.Static

```
public class MainStatic
{
  public static void main(String[] args)
  {
        Test.Static.TestStatic.printMe();
  }
}
```

Class yang akan memakai static method di package Test.Static

Kita rubah supaya memakai Static Import dengan cara sebagai berikut,

```
package tiger;

public class TestStatic
{
 public static void printMe()
 {
   System.out.println("Print Me");
 }
}
```

Static method kita dalam package tiger

```
import static tiger.TestStatic.*;

public class MainStatic_1_5
{
 public static void main(String[] args)
```

Class yang akan memakai static method di class tiger.TestStatic

```
  {
    printMe();
  }
}
```

Jadi kita bisa lihat, kita tidak perlu menuliskan tiger.TestStatic.printMe() untuk memanggil method kita, hanya printMe() saja sudah cukup.

Metadata

Dalam prototype yang didownload belum tersedia fasilitas Metadata ini, mungkin di release berikutnya akan diikutsertakan. Metadata membantu developer supaya tidak perlu melakukan coding untuk code-code yang dianggap sebagai “boilerplate” code. Kita ambil contoh di bawah ini

```
import javax.xml.rpc.*;

public class CoffeeOrder
{
  @Remote public Coffee [] getPriceList()
  {
    ...
  }

    @Remote public String orderCoffee(String name, int quantity)
  {
    ...
  }
}
```

Dengan menggunakan code annotations @Remote secara otomatis tools akan mengenerate class yang akan dipakai oleh class CoffeeOrder diatas tersebut