Generics

Bok, Jong Soon
javaexpert@nate.com
https://github.com/swacademy/JavaSE

Generics

Generic programming is a programming method that is based in finding the most abstract representations of efficient algorithms.

-Alexander Stepanov

http://en.wikipedia.org/wiki/Generic

- cf) C++ template and STL (Standard Template Library)
- A ***generic type***, which is also referred to as a ***parameterized type***, is a class or interface type definition that has one or more type parameters.

Generics (Cont.)

```
2  public class GenericsDemo {
3     public static void main(String[] args) {
4        java.util.Vector<String> vector;
5        vector = new java.util.Vector<String>();
6        vector.addElement("Hello");
7        vector.addElement("World");
8        //vector.addElement(5);   //compile error
9        for(String str : vector){
10          System.out.println(str);
11       }
12    }
13 }
14 /*
15 Hello
16 World
17 */
```

```
1  public class GenericsDemo{
2     public static void main(String[] args) {
3        java.util.Vector vector;
4        vector = new java.util.Vector();
5        vector.add("Hello");
6        vector.add("World");
7        //vector.add(5);   //compile시 발견할 수 없슴
8
9        int size = vector.size();
10       for(int i = 0 ; i < size ; i++){
11          String str = (String)vector.elementAt(i);
12          System.out.println(str);
13       }
14    }
15 }
```

```
---------- Java Compiling ----------
Note: GenericsDemo.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

Output completed (2 sec consumed) - Normal Termination
```

```
C:\JavaRoom>javac -Xlint:unchecked GenericsDemo.java
GenericsDemo.java:5: warning: [unchecked] unchecked call to add(E) as a member o
f the raw type java.util.Vector
                vector.add("Hello");
                          ^
GenericsDemo.java:6: warning: [unchecked] unchecked call to add(E) as a member o
f the raw type java.util.Vector
                vector.add("World");
                          ^
2 warnings
```

Generics – in java 1.4

```
2  public class GenericsDemo {
3      public static void main(String[] args) {
4          // TODO Auto-generated method stub
5          //A Library containing only Books
6          Library myBooks = new Library();
7          myBooks.addMedia(new Book());
8          myBooks.addMedia(new Video());
9          Book lastBook = (Book)myBooks.retrieveLast();
10     }
11 }
12 class Library{
13     private java.util.List resource = new java.util.ArrayList();
14     public void addMedia(Media x){
15         resource.add(x);
16     }
17     public Media retrieveLast(){
18         int size = resource.size();
19         if(size > 0) return resource.get(size -1);
20         return null;
21     }
22 }
23 class Media{}
24 class Book extends Media{}
25 class Video extends Media{}
26 class Newspaper extends Media{}
```

Generics – in java 1.4 (Cont.)

- Type casting the return value from the `retrieveLast()` method to `Book` is necessary.
- The compiler knows :
  - What kind of object is returned.
  - What kinds of operations can be performed on that object.
- Even though the programmer may be certain that only `Book` objects will be returned, the compiler does not know this.
- A `Video` being stored in `myBooks` and then returned here, the cast would cause an unexpected exception to be thrown at runtime.

Generics – since Java 1.5

```
 2 public class GenericsDemo {
 3    public static void main(String[] args) {
 4       // TODO Auto-generated method stub
 5       //A Library containing only Books
 6       Library<Book> myBooks = new Library<Book>();
 7       myBooks.addMedia(new Book());
 8       //myBooks.addMedia(new Video());
 9       Book lastBook = myBooks.retrieveLast();
10    }
11 }
12 class Library<Book>{
13    private java.util.List<Book> resource = new java.util.ArrayList<Book>();
14    public void addMedia(Book x){
15       resource.add(x);
16    }
17    public Book retrieveLast(){
18       int size = resource.size();
19       if(size > 0) return resource.get(size -1);
20       return null;
21    }
22 }
```

Generics – since Java 1.5

- `Library` is now a *generic type* with a single *type parameter E*.
- A `Library` containing only `Book` objects, for example, would be written as `Library<Book>`.
- The code snippet now looks like this :

```
Library<Book> myBooks = new Library<Book>();
    //
Book lastBook = myBooks.retrieveLast();
```

Using Generic Types

- You'll usually se generics when dealing with collections of some kind.
- The ***Collections Framework*** was a major motivation.
- Enable compile-time checking of the type safety of operations on a collection.
- When you specify the type of object stored in a collection :
  - The compiler can verify any operation that adds an object to the collection.
  - The type of an object retrieved from a collection is known, so there's no need to cast it to a type.

Type Parameter Conventions

- The angle bracket <, > and single or more letter notation used to represent a type parameter.
- A type parameter is a single, uppercase letter – this allows easy identification and distinguishes a type parameter from a class name.
- The most common type parameters you will see are :
  - `<T>` -- Type
  - `<S>` -- for type, when T is already in use
  - `<E>` -- Element (used extensively by the Java Collections Framework)
  - `<K>` -- Key
  - `<V>` -- value
  - `<N>` -- Number

Sample Code I

```
2  public class ValueWrapper<T> {
3     private T value;
4     public ValueWrapper(T value){
5        this.value = value;
6     }
7     public T value(){
8        return this.value;
9     }
10    public static void main(String[] args) {
11       // TODO Auto-generated method stub
12       ValueWrapper<String> sf = new ValueWrapper<String>("Hello");
13       System.out.println(sf.value());
14
15       ValueWrapper<Integer> sf1 = new ValueWrapper<Integer>(5);
16       System.out.println(sf1.value());
17
18    }
19 }
20 /*
21 Hello
22 5
23 */
```

Generics and Relationship between Types

- You might expect that `ArrayList<Object>` is a supertype of `ArrayList<String>`, because `Object` is a supertype of `String`.
- No such relationship exists for instances of generic types.

```
List<String> ss = new ArrayList<String>(1);
List<Object> os = ss; //error
```

- This causes a compile error.
- ***The compiler does not allow you to make any assignment that may compromise type safety***.

Generics and Type Erasure

- When a generic type is instantiated, the compiler translates those types.
- A process where the compiler removes all information related to type parameters and type arguments within a class or method.
- ***Type erasure*** means that Java applications that use generics maintain binary compatibility with Java libraries and applications created before generics.

Generics and Type Erasure (Cont.)

- e.g. `Iterator<String>` is translated to type `Iterator`, which is called the *raw type*.
  - A *raw type* is a class without a type argument.

```
public class MyClass<E> {
   public static void myMethod(Object item) {
      if(item instanceof E) { //compile error
        ...
      }
      E item2 = new E();  //compile error
      E [] iArray = new E[10];  //compile error
      E obj = (E)new Object(); //Unchecked cast warning
   }
}
```

Sample Code II

```
1  import java.util.*;
2  public class GenericsDemo1 {
3      public static void main(String[] args) {
4          // TODO Auto-generated method stub
5          Vector<String> names;
6          Vector<Integer> scores;
7          Vector<Object> obj;
8          Vector<?> wild;
9          Vector raw;
10         names = new Vector<String>();
11         scores = new Vector<Integer>();
12         //scores = names;                    //compile error
13         //obj = names;                       //compile error
14         //obj = (Vector<Object>)names;       //compile error
15         wild = names;
16
17         //names = wild;                      //compile error
18         names = (Vector<String>)wild;        //warining message
19         raw = names;
20         System.out.println("Program is Over...");
21     }
22 }
```

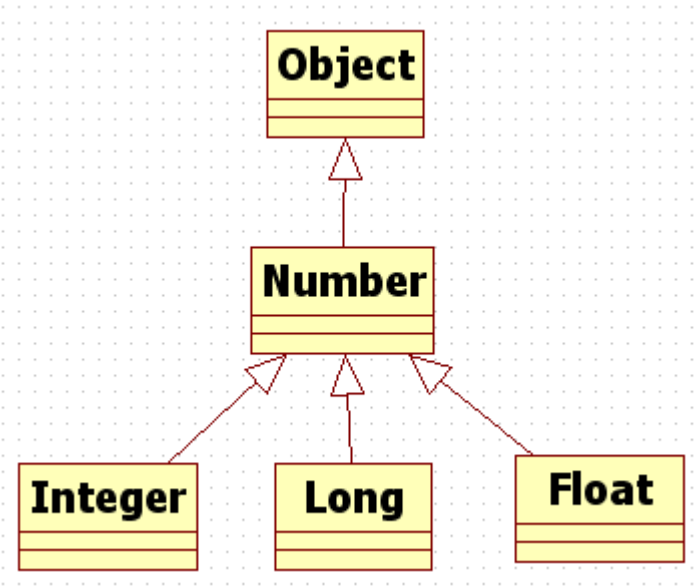
Wildcard Types

- Might see type argument notation that uses a question mark :

  e.g. `reverse (List<?> list)`
- The question mark is called the ***wildcard type***.
- Represents ***some*** type, but one that is not known at compile time.
- You might think that `List<?>` is the same as `List<Object>`. ***It is not***.
- If the `reverse` method had been defined as accepting type `List<Object>` then the compiler would not allow a `List<Integer>` to be passed to the method.

Constraining a Type with a Bound

- It is also possible to constrain the wildcard type with an ***upper*** or ***lower bound*** (but not both).
- To illustrate how to define a bound, look at the following classes from the JDK API :



Constraining a Type with a Bound (Cont.)

- Here is the syntax for constraining a wildcard type with a bound :
  - **`super className`**

    The type is constrained with a ***lower bound***.

    e.g. "**`List<? super Number>`**" → the **`List`** must contain either **`Number`**s of **`Object`**s.
  - **`extends className`**

    The type is constrained with an ***upper bound***.

    e.g. "**`List<? extends Number>`**" the **`List`** must contain **`Number`**s, **`Integer`**s, **`Long`**s, **`Float`**s or one of the other subtypes of **`Number`**.

Sample Code III

```
2  public class Pair<T extends Number> {
3     private T v1, v2;
4     public Pair(T v1, T v2){
5        this.v1 = v1;
6        this.v2 = v2;
7     }
8     public T first(){
9        return this.v1;
10    }
11    public T second(){
12       return this.v2;
13    }
14    public static void main(String[] args) {
15       // TODO Auto-generated method stub
16       Pair<Integer> su = new Pair<Integer>(3,4);
17       System.out.println(su.first());
18       Pair<Double> d = new Pair<Double>(3.0, 4.0);
19       System.out.println(d.second());
20    }
21 }
```

Using Generic Methods

- Defines one or more type parameters in the method signature, ***before the return type*** :

```
static <T> boolean myMethod (
                        List<? Extends T>, T obj)
```

- A type parameter is used to express dependencies between :
  - The types of the method's arguments
  - The type of the method's argument and the method's return type
  - both

Sample Code IV

```
2  public class Sorter {
3      public static void main(String[] args) {
4          // TODO Auto-generated method stub
5          String [] array = { "jkl", "ghi", "pqr", "abc", "def", "mno", };
6          Sorter.sort(array);
7          for(int i = 0; i<array.length; i++){
8              System.out.println(array[i]);
9          }
10     }
11     static <T extends Comparable<T>> void sort(T [] a){
12         for(int i=0; i< a.length; i++){
13             for(int j = 0; j < i ; j++){
14                 if(a[j].compareTo(a[i]) > 0){
15                     swap(a, i, j);
16                 }
17             }
18         }
19     }
20     static <T> void swap(T [] a, int i, int j){
21         T t = a[i];
22         a[i] = a[j];
23         a[j] = t;
24     }
25 }
```