Java Programmingn Tutorial Generics

TABLE OF CONTENTS (HIDE)

1. Introduction to Generics (JDK 1.5)

You are certainly familiar with passing arguments into methods. You place the arguments inside the round bracket `()` and pass them to the method. In generics, instead of pass arguments, we pass *type information* (inside the angle brackets <>).

JDK 1.5 introduces *generics*, which allows us to *abstract over types* (or *parameterized types*). The class designers can be *generic about types in the definition*, while the users can be *specific in the types during the object instantiation or method invocation*.

The primary usage of generics is to abstract over types when working with *collections* (Read "The Collection Framework" if necessary).

For example, the class `ArrayList` is designed (by the class designer) to take a generics type <E> as follows:

```
public class ArrayList<E> implements List<E> .... {
   // Constructor
   public ArraList() { ...... }

   // Public methods
   public boolean add(E e) { ...... }
   public void add(int index, E element) { ...... }
   public boolean addAll(int index, Collection<? extends E> c)
   public abstract E get(int index) { ...... }
   public E remove(int index)
   .......
}
```

To instantiate an `ArrayList`, the users need to provide the actual type for <E> for this particular instance. The actual type provided will then substitute all references to `E` inside the class. For example,

```
ArrayList<Integer> lst1 = new ArrayList<Integer>(); // E substituted with Integer
lst1.add(0, new Integer(88));
lst1.get(0);

ArrayList<String> lst2 = new ArrayList<String>();   // E substituted with String
lst2.add(0, "Hello");
lst2.get(0);
```

The above example showed that the class designers could be *generic* about type; while the class users provide the *specific* actual type information during instantiation. The type information is passed inside the angle bracket <>, just like method arguments are passed inside the round bracket `()`.

Pre-Generic Collections are not Type-Safe

If you are familiar with the pre-JDK 1.5's collections such as `ArrayList`, they are designed to hold `java.lang.Object`. Via polymorphism, any subclass of `Object` can be substituted for `Object`. Since `Object` is the common root class of all the Java's classes, a collection designed to hold `Object` can hold any Java classes. There is, however, one problem. Suppose, for example, you wish to define an `ArrayList` of `String`. In the `add(Object)` operation, the `String` will be upcasted implicitly into `Object` by the compiler. During retrieval, however, it is the programmer's responsibility to downcast the `Object` back to an `String` explicitly. If you inadvertently added in a non-`String` object. the compiler cannot detect the error, but the downcasting will fail at runtime (`ClassCastException` thrown). Below is an example:

```
// Pre-JDK 1.5
import java.util.*;
public class ArrayListWithoutGenericsTest {
   public static void main(String[] args) {
      List strLst = new ArrayList();  // List and ArrayList holds Objects
      strLst.add("alpha");            // String upcast to Object implicitly
      strLst.add("beta");
      strLst.add("charlie");
      Iterator iter = strLst.iterator();
      while (iter.hasNext()) {
         String str = (String)iter.next(); // need to explicitly downcast Object back to String
```

```
            System.out.println(str);
         }
         strLst.add(new Integer(1234));       // Compiler/runtime cannot detect this error
         String str = (String)strLst.get(3);  // compile ok, but runtime ClassCastException
      }
   }
```

We could use an `instanceof` operator to check for proper type before downcasting. But again, `instanceof` detects the problem at runtime. How about compile-time type-checking?

2. Generics

Let's write our own "type-safe" `ArrayList`

We shall illustrate the use of generics by writing our own *type-safe* resizable array for holding a particular type of objects (similar to an `ArrayList`).

Let's begin with a version without generics called `MyArrayList`:

```
// A dynamically allocated array which holds a collection of java.lang.Object - without generics
public class MyArrayList {
   private int size;     // number of elements
   private Object[] elements;

   public MyArrayList() {         // constructor
      elements = new Object[10];  // allocate initial capacity of 10
      size = 0;
   }

   public void add(Object o) {
      if (size < elements.length) {
         elements[size] = o;
      } else {
         // allocate a larger array and add the element, omitted
      }
      ++size;
   }

   public Object get(int index) {
      if (index >= size)
         throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + size);
      return elements[index];
   }

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

The `MyArrayList` is not *type-safe*. For example, if we create a `MyArrayList` which is meant to hold `String`, but added in an `Integer`. The compiler cannot detect the error. This is because `MyArrayList` is designed to hold `Object`s and any Java classes can be upcast to `Object`.

```
public class MyArrayListTest {
   public static void main(String[] args) {
      // Intends to hold a list of Strings, but not type-safe
      MyArrayList strLst = new MyArrayList();
      // adding String elements - implicitly upcast to Object
      strLst.add("alpha");
      strLst.add("beta");
      // retrieving - need to explicitly downcast back to String
      for (int i = 0; i < strLst.size(); ++i) {
         String str = (String)strLst.get(i);
         System.out.println(str);
      }

      // Inadvertently added a non-String object will cause a runtime
      // ClassCastException. Compiler unable to catch the error.
      strLst.add(new Integer(1234));   // compiler/runtime cannot detect this error
      for (int i = 0; i < strLst.size(); ++i) {
         String str = (String)strLst.get(i);   // compile ok, runtime ClassCastException
         System.out.println(str);
      }
   }
}
```

If you intend to create a list of `String`, but inadvertently added in an non-`String` object, the non-`String` will be upcasted to `Object` implicitly. The compiler is not able to check whether the downcasting is valid at compile-time (this is known as late binding or dynamic binding). Incorrect downcasting will show up only at runtime, in the form of `ClassCastException`, which could be too late. The compiler is not able to catch this error at compiled time. Can we make the compiler to catch this error and ensure *type safety* at runtime?

2.1 Generics Classes

JDK 1.5 introduces the so-called *generics* to resolve this problem. *Generics* allow you to *abstract over types*. You can design a class with a *generic type*, and provide the *specific type information* during the instantiation. The compiler is able to perform the necessary type checking during compile time and ensure that no type-casting error occurs at runtime. This is known as *type-safety*.

Take a look at the declaration of interface `java.util.List<E>`:

```
public interface List<E> extends Collection<E> {
   boolean add(E o);
   void add(int index, E element);
   boolean addAll(Collection<? extends E> c);
   boolean containsAll(Collection<?> c);
   ......
}
```

`<E>` is called the *formal "type" parameter*, which can be used for passing "type" parameters during the actual instantiation.

The mechanism is similar to method invocation. Recall that in a method's definition, we declare the *formal parameters* for passing data into the method. For example,

```
// A method's definition
public static int max(int a, int b) {  // int a, int b are formal parameters
   return (a > b) ? a : b;
}
```

During the invocation, the formal parameters are substituted by the *actual parameters*. For example,

```
// Invocation: formal parameters substituted by actual parameters
int maximum = max(55, 66);   // 55 and 66 are actual parameters
int a = 77, b = 88;
maximum = max(a, b);         // a and b are actual parameters
```

Formal type parameters used in the class declaration have the same purpose as the formal parameters used in the method declaration. A class can use *formal type parameters* to receive type information when an instance is created for that class. The actual types used during instantiation are called *actual type parameters*.

Let's return to the `java.util.List<E>`, in an actual invocation, such as `List<Integer>`, all occurrences of the formal type parameter `<E>` are replaced by the actual type parameter `<Integer>`. With this additional type information, compiler is able to perform type check during compile-time and ensure that there won't have type-casting error at runtime.

Formal Type Parameter Naming Convection

Use an uppercase single-character for formal type parameter. For example,

- `<E>` for an element of a collection;
- `<T>` for type;
- `<K, V>` for key and value.
- `<N>` for number
- `S`,`U`,`V`, etc. for 2nd, 3rd, 4th type parameters

Example of Generic Class

In this example, a class called `GenericBox`, which takes a generic type parameter `E`, holds a `conetent` of type `E`. The constructor, getter and setter work on the parameterized type `E`. The `toString()` reveals the actual type of the `content`.

```
public class GenericBox<E> {
   // Private variable
   private E content;

   // Constructor
   public GenericBox(E content) {
      this.content = content;
   }

   public E getContent() {
      return content;
   }

   public void setContent(E content) {
      this.content = content;
   }

   public String toString() {
      return content + " (" + content.getClass() + ")";
   }
}
```

The following test program creates `GenericBox`es with various types (`String`, `Integer` and `Double`). Take note that JDK 1.5 also introduces autoboxing and unboxing to convert between primitives and wrapper objects.

```
public class TestGenericBox {
   public static void main(String[] args) {
      GenericBox<String> box1 = new GenericBox<String>("Hello");
      String str = box1.getContent();  // no explicit downcasting needed
      System.out.println(box1);
      GenericBox<Integer> box2 = new GenericBox<Integer>(123);  // autobox int to Integer
      int i = box2.getContent();       // downcast to Integer, auto-unbox to int
      System.out.println(box2);
      GenericBox<Double> box3 = new GenericBox<Double>(55.66);  // autobox double to Double
      double d = box3.getContent();     // downcast to Double, auto-unbox to double
      System.out.println(box3);
   }
}
```

```
Hello (class java.lang.String)
123 (class java.lang.Integer)
55.66 (class java.lang.Double)
```

Type Erasure

From the previous example, it seems that compiler substituted the parameterized type `E` with the actual type (such as `String`, `Integer`) during instantiation. If this is the case, the compiler would need to create a new class for each actual type (similar to C++'s template).

In fact, the compiler replaces all reference to parameterized type E with Object, performs the type check, and insert the required downcast operators. For example, the `GenericBox` is compiled as follows (which is compatible with codes without generics):

```
public class GenericBox {
   // Private variable
   private Object content;

   // Constructor
   public GenericBox(Object content) {
      this.content = content;
   }

   public Object getContent() {
      return content;
   }

   public void setContent(Object content) {
      this.content = content;
   }

   public String toString() {
      return content + " (" + content.getClass() + ")";
   }
}
```

The compiler also inserts the required downcast operator in the test codes:

```
GenericBox box1 = new GenericBox("Hello");  // upcast is type-safe
String str = (String)box1.getContent();     // compiler inserts downcast operation
System.out.println(box1);
```

In this way, the same class definition is used for all the types. Most importantly, the bytecodes are compatible with those without generics. This process is called *type erasure*.

Continue with our "type-safe" `ArrayList`...

Let's return to the `MyArrayList` example. With the use of generics, we can rewrite our program as follows:

```
// A dynamically allocated array with generics
public class MyGenericArrayList<E> {
   private int size;     // number of elements
   private Object[] elements;

   public MyGenericArrayList() {  // constructor
      elements = new Object[10];  // allocate initial capacity of 10
      size = 0;
   }

   public void add(E e) {
      if (size < elements.length) {
         elements[size] = e;
      } else {
         // allocate a larger array and add the element, omitted
      }
      ++size;
   }
```

```

   public E get(int index) {
      if (index >= size)
         throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + size);
      return (E)elements[index];
   }

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

Dissecting the Program

MyGenericArrayList<E> declare a generics class with a *formal type parameter* <E>. During an actual invocation, e.g., MyGenericArrayList<String>, a specific type <String>, or *actual type parameter*, replaced the formal type parameter <E>.

Behind the scene, generics are implemented by the Java compiler as a front-end conversion called *erasure*, which translates or rewrites code that uses generics into non-generic code (to ensure backward compatibility). This conversion erases all generic type information. For example, ArrayList <Integer> will become ArrayList. The formal type parameter, such as <E>, are replaced by Object by default (or by the upper bound of the type). When the resulting code is not type correct, the compiler insert a type casting operator.

Hence, the translated code is as follows:

```
// The translated code
public class  MyGenericArrayList {
   private int size;     // number of elements
   private Object[] elements;

   public MyGenericArrayList() {  // constructor
      elements = new Object[10];  // allocate initial capacity of 10
      size = 0;
   }

   // Compiler replaces E with Object, but check e is of type E, when invoked to ensure type-safety
   public void add(Object e) {
      if (size < elements.length) {
         elements[size] = e;
      } else {
         // allocate a larger array and add the element, omitted
      }
      ++size;
   }

   // Compiler replaces E with Object, and insert downcast operator (E<E>) for the return type when invoked
   public Object get(int index) {
      if (index >= size)
         throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + size);
      return (Object)elements[index];
   }

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

When the class is instantiated with an actual type parameter, e.g. MyGenericArrayList<String>, the compiler ensures add(E e) operates on only String type. It also inserts the proper downcasting operator to match the return type E of get(). For example,

```
public class MyGenericArrayListTest {
   public static void main(String[] args) {
      // type safe to hold a list of Strings
      MyGenericArrayList<String> strLst = new MyGenericArrayList<String>();

      strLst.add("alpha");   // compiler checks if argument is of type String
      strLst.add("beta");

      for (int i = 0; i < strLst.size(); ++i) {
         String str = strLst.get(i);   // compiler inserts the downcasting operator (String)
         System.out.println(str);
      }

      strLst.add(new Integer(1234));  // compiler detected argument is NOT String, issues compilation error
   }
}
```

With generics, the compiler is able to perform type checking during compilation and ensure type safety at runtime.

Unlike "template" in C++, which creates a new type for each specific parameterized type, in Java, a generics class is only compiled once, and there is only one single class file which is used to create instances for all the specific types.

2.2 Generic Methods

Methods can be defined with generic types as well (similar to generic class). For example,

```
public static <E> void ArrayToArrayList(E[] a, ArrayList<E> lst) {
   for (E e : a) lst.add(e);
}
```

A generic method can declare formal type parameters (e.g. `<E>`, `<K,V>`) *preceding the return type*. The formal type parameters can then be used as *placeholders* for return type, method's parameters and local variables within a generic method, for proper type-checking by compiler.

Similar to generics class, when the compiler translates a generic method, it replaces the formal type parameters using *erasure*. All the generic types are replaced with type `Object` by default (or the upper bound of type). The translated version is as follows:

```
public static void ArrayToArrayList(Object[] a, ArrayList lst) {  // compiler checks if a is of type E[],
                                                                  //   lst is of type ArrayList<E>
   for (Object e : a) lst.add(e);                                 // compiler checks if e is of type E
}
```

However, compiler checks that `a` is of the type `E[]`, `lst` is of type `ArrayList<E>`, and `e` is of type `E`, during invocation to ensure type-safety. For example,

```
import java.util.*;
public class TestGenericMethod {

   public static <E> void ArrayToArrayList(E[] a, ArrayList<E> lst) {
      for (E e : a) lst.add(e);
   }

   public static void main(String[] args) {
      ArrayList<Integer> lst = new ArrayList<Integer>();

      Integer[] intArray = {55, 66};  // autobox
      ArrayToArrayList(intArray, lst);
      for (Integer i : lst) System.out.println(i);

      String[] strArray = {"one", "two", "three"};
      //ArrayToArrayList(strArray, lst);   // Compilation Error below
   }
}
```

```
TestGenericMethod.java:16: <E>ArrayToArrayList(E[],java.util.ArrayList<E>) in TestGenericMethod
cannot be applied to (java.lang.String[],java.util.ArrayList<java.lang.Integer>)
      ArrayToArrayList(strArray, lst);
      ^
```

Generics have an optional syntax for specifying the type for a generic method. You can place the actual type in angle brackets `<>`, between the dot operator and method name. For example,

```
TestGenericMethod.<Integer>ArrayToArrayList(intArray, lst);
```

The syntax makes the code more readable and also gives you control over the generic type in situations where the type might not be obvious.

2.3 Wildcards

Consider the following lines of codes:

```
ArrayList<Object> lst = new ArrayList<String>();
```

It causes a compilation error "incompatible types", as `ArrayList<String>` is not an `ArrayList<Object>`.

This error is against our intuition on polymorphism, as we often assign a subclass instance to a superclass reference.

Consider these two statements:

```
List<String> strLst = new ArrayList<String>();   // 1
List<Object> objLst = strList;                   // 2 - Compilation Error
```

Line 2 generates a compilation error. But if line 2 succeeds and some arbitrary objects are added into `objLst`, `strLst` will get "corrupted" and no longer contains only `String`s. (`objLst` and `strLst` have the same reference.)

Because of the above, suppose we want to write a method called `printList(List<.>)` to print the elements of a List. If we define the method as `printList(List<Object> lst)`, then it can only accept an argument of `List<object>`, but not `List<String>` or `List<Integer>`. For example,

```
import java.util.*;
public class TestGenericWildcard {

   public static void printList(List<Object> lst) {  // accept List of Objects only,
                                                     // not List of subclasses of object
      for (Object o : lst) System.out.println(o);
   }
```

```

   public static void main(String[] args) {
      List<Object> objLst = new ArrayList<Object>();
      objLst.add(new Integer(55));
      printList(objLst);   // matches

      List<String> strLst = new ArrayList<String>();
      strLst.add("one");
      printList(strLst);  // compilation error
   }
}
```

Unbounded Wildcard `<?>`

To resolve this problem, a wildcard (`?`) is provided in generics, which stands for *any unknown type*. For example, we can rewrite our `printList()` as follows to accept a `List` of any unknown type.

```
public static void printList(List<?> lst) {
  for (Object o : lst) System.out.println(o);
}
```

Upperbound Wildcard `<? extends type>`

The wildcard `<? extends type>` stands for `type` and its sub-`type`. For example,

```
public static void printList(List<? extends Number> lst) {
  for (Object o : lst) System.out.println(o);
}
```

`List<? extends Number>` accepts `List` of `Number` and any subtype of `Number`, e.g., `List<Integer>` and `List<Double>`.

Clearly, `<?>` can be interpreted as `<? extends Object>`, which is applicable to all Java classes.

Another example,

```
// List<Number> lst = new ArrayList<Integer>();  // Compilation Error
List<? extends Number> lst = new ArrayList<Integer>();
```

Lowerbound Wildcard `<? super type>`

The wildcard `<? super type>` matches `type`, as well as its super-`type`. In other words, it specifies the lower bound.

Read Java Online Tutorial "More Fun with Wildcards".

[TODO] Example

2.4 Bounded Generics

A bounded parameter type is a generic type that specifies a bound for the generic, in the form of `<T extends ClassUpperBound>`, e.g., `<T extends Number>` accepts `Number` and its subclasses (such as `Integer` and `Double`).

Example

The method `add()` takes a type parameter `<T extends Number>`, which accepts `Number` and its subclasses (such as `Integer` and `Double`).

```
public class MyMath {
   public static <T extends Number> double add(T first, T second) {
      return first.doubleValue() + second.doubleValue();
   }

   public static void main(String[] args) {
      System.out.println(add(55, 66));     // int -> Integer
      System.out.println(add(5.5f, 6.6f)); // float -> Float
      System.out.println(add(5.5, 6.6));   // double -> Double
   }
}
```

How the compiler treats the bounded generics?

As mentioned, by default, all the generic types are replaced with type `Object`, during the code translation. However, in the case of `<? extends Number>`, the generic type is replaced by the type `Number`, which serves as the *upper bound* of the generic types.

Example

```
public class TestGenericsMethod {
   public static <T extends Comparable<T>> T maximum(T x, T y) {
      return (x.compareTo(y) > 0) ? x : y;
   }

```

```
    public static void main(String[] args) {
       System.out.println(maximum(55, 66));
       System.out.println(maximum(6.6, 5.5));
       System.out.println(maximum("Monday", "Tuesday"));
    }
 }
```

By default, `Object` is the *upper-bound* of the parameterized type. `<T extends Comparable<T>>` changes the upper bound to the `Comparable` interface, which declares an `abstract` method `compareTo()` for comparing two objects.

The compiler translates the above generic method to the following codes:

```
public static Comparable maximum(Comparable x, Comparable y) {   // replace T by upper bound type Comparable
                                                                 // Compiler checks x, y are of the type Comparable
                                                                 // Compiler inserts a type-cast for the return value
   return (x.compareTo(y) > 0) ? x : y;
}
```

When this method is invoked, e.g. via `maximum(55, 66)`, the primitive `int`s are auto-boxed to `Integer` objects, which are then implicitly upcasted to `Comparable`. The compiler checks the type to ensure type-safety. The compiler also inserts an explicit downcast operator for the return type. That is,

```
(Comparable)maximum(55, 66);
(Comparable)maximum(6.6, 5.5);
(Comparable)maximum("Monday", "Tuesday");
```

We do not have to pass an actual type argument to a generic method. The compiler infers the type argument automatically, based of the type of the actual argument passed into the method.

LINK TO JAVA REFERENCES & RESOURCES

More References

1. Java Online Tutorial on "Generics" @ http://docs.oracle.com/javase/tutorial/extra/generics/index.html.
2. Java Online Tutorial on "Collections" @ http://docs.oracle.com/javase/tutorial/collections/index.html.

Latest version tested: JDK 1.7.0_03
Last modified: May, 2012