Tutorial Link https://course.testpad.chitkara.edu.in/tutorials/Java:Generics/5b767c2f4227cf4f24a30e2a

TUTORIAL

Java: Generics

Topics

JDK 1.5 introduces generics, which supports abstraction 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. 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 <>.

We shall illustrate the use of generics by writing our own *type-safe* resizable array for holding a particular type of objects.

Java

```
class CQArrayList {
  private int size;     // number of elements
  private Object[] elements;
  public CQArrayList()   {
    elements = new Object[10];
    size = 0;
  }
  public void add(Object o)   {
    if (size < elements.length)
      elements[size] = o;
    else{
      // allocate a larger array and add the elements here
    }
```

```
14      ++size;
15    }
16    public Object get(int index)   {
17      if (index >= size)
18        throw new IndexOutOfBoundsException("Index: " +
index + ", Size: " + size);
19      return elements[index];
20    }
21    public int size() { return size; }
22  }
23
24  class Main {
25    public static void main(String[] args)   {
26      String str;
27      CQArrayList lst1 = new CQArrayList();
28      lst1.add("Code");   // We inserted an String
29      lst1.add("Quotient");
30      for (int i = 0; i < lst1.size(); ++i)
31        System.out.println((String)lst1.get(i));   //
compile ok, runtime ClassCastException
32
33      // Inadvertently added a non-String object will
cause a runtime ClassCastException.
34      lst1.add(new Integer(12));   // compiler cannot
detect this error that it is integer so it will
compile, but at runtime, there will be an Exception.
35      // Comment the above line to make the program work
correctly
36
37      for (int i = 0; i < lst1.size(); ++i)
38        System.out.println((String)lst1.get(i));   //
compile ok, runtime ClassCastException
39    }
40  }
```

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?

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 Convention

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

For example,

Java

```
class CQGeneric<E> // Generic class
{
  private E content;
  // Constructor
  public CQGeneric(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() + ")";
  }
}

class Main
{
  public static void main(String[] args)
  {
    CQGeneric<String> box1 = new CQGeneric<String>
("Code Quotient");
    String str = box1.getContent();  // no explicit
downcasting needed
    System.out.println(box1);
    CQGeneric<Integer> box2 = new CQGeneric<Integer>
(5587);  // autobox int to Integer
    int i = box2.getContent();       // downcast to
Integer, auto-unbox to int
    System.out.println(box2);
    CQGeneric<Double> box3 = new CQGeneric<Double>
(22.33);  // autobox double to Double
    double d = box3.getContent();     // downcast to
Double, auto-unbox to double
    System.out.println(box3);
  }
}
```

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 bytecode are compatible with those without generics. This process is called *type erasure*.

The previous SafeArray list example can be done using Generics as below: -

Java

```
class CQGenericArray<E>
{
   private int size;
   private Object[] elements;

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

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

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

      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(22));  // 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.

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`).

Java

```
class Main
{
   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(22, 23));     // int -> Integer
      System.out.println(add(5.8f, 7.6f)); // float -> Float
      System.out.println(add(23.5, 54.6));   // double -> Double
```

```
    }
}
```