L61

Monday, February 7, 2022 6:33 PM

Generics - Introduction, Generics and Collections, Correct usage of Generics

Introduction to Generics

Q1. In order to understand Generics and its advantages, we need to understand the importance of types.

Java is a strongly typed language. Meaning, Java language mandates that we clearly declare the data type of a variable/reference before it is used for the first time.

For example, in the below code:
int age;
age = 3;
Variable age is declared to be of data type int. Java compiler uses this information to verify and flag an error if some other type value, say, for example a String is being assigned. For example, the below code will not compile.
int age;
age = "Hello"; *//Compiler will flag this as an **error**, saying incompatible types.*
Until **Java 5** (version 1.5), all the collection classes used to work on Object as the data type, so that any kind of object could be stored in **Lists**, **Sets** etc.

However, this approach had two disadvantages which were solved by the inclusion of Generics in **Java 5**.

Prior to generics, an ArrayList could include objects of any type meaning, a developer could store an **Integer** and a **String** object in the same ArrayList.

For example:
ArrayList numbersList = new ArrayList();
numbersList.add(new Integer(72));
numbersList.add(new Integer(78));
numbersList.add("Alfa"); *// **Statement 1***
In the above code compiler will not flag Statement 1 as an error, since numbersList can accept any type of object.

However, Statement 1 will cause a runtime error during code execution, if the code is trying to calculate the sum of all integers stored in the numbersList.

In such situations, Generics allows us to specify the type of elements that can be stored in the ArrayList during declaration. For example, if we want the ArrayList to only accept Integers, we would declare the ArrayList as given below:
ArrayList<**Integer**> numbersList = new ArrayList<**Integer**>();
The first advantage of using the above Generic syntax to specify the type parameter as **<Integer>** is that compiler will allow only elements of type **Integer** to be added to numbersList. Compiler will flag an error if an attempt is made to add an object of type **String** or any other type other than **Integer**.

The second advantage of using the above syntax is that we need not **type cast** the elements from Object to Integer when we retrieve them. For example we can directly say:
Integer number1 = numbersList.get(0);
See and retype the below code to gain familiarity with Generic syntax.

Note the usage of < and > to specify the type argument **Integer**. Note that there should not be any spaces before and after < and >.

Also note that in **Java 7** (version 1.7) and later versions the below statement
List<**Integer**> numbersList = new ArrayList<**Integer**>();
can also be written as
List<**Integer**> numbersList = new ArrayList<>();

```java
package q11388;
import java.util.*;
public class GenericListDemo {
    public static void main(String[] args) {
        List<Integer> numbersList = new ArrayList<Integer>();
        numbersList.add(new Integer(72));
        numbersList.add(78);
        numbersList.add(81);
        int total = 0;
        for (int number : numbersList) {
            total = total + number;
        }
        System.out.println("total = " + total);
    }
}
```

Q2. All the collections in Java are parameterized using generic syntax. For example when we see the List interface we will find :
public interface List<**E**> extends Collection<**E**>{
    public boolean add(**E e**);
public **E** get(int index);
    ...
    ...
}
In the above example, List is called **generic interface**. Similarly we can have **generic classes**. The E surrounded by < and > is called the type parameter.

In the below code:
List<**String**> namesList = new ArrayList<**String**>();
String is called type argument passed to **List** and **ArrayList**.

Any **class** or **interface** which accepts parameterized types is called a **generic class** or a **generic interface** respectively. Select all the correct statements for the below code:
class A { *// statement 1*}
class B<**T**> { *// statement 2*}
B b1 = new B(); *// statement 3*B<**String**> b1 = new B<**String**>(); *// statement 4*

- [x] In **statement 1**, class A is called a non-generic class.

- [x] In **statement 1**, class `A` is called a non-generic class.
- [x] In **statement 2**, class `B` is called a generic class.
- [ ] **Statement 3** will result in compilation error. Since class `B` is a generic class, we should pass some **type argument** during instantiation.
- [ ] In **statement 4**, **String** is called `type parameter`.

Q3. It is very important to know the difference between type parameter and type argument.

A class or an interface is of a *generic type* when it uses parameterized types. For example:

```
public class Calculator<T> {
        public T sum(T number1, T number2) {
                return number1 + number2;
        }
}
```

In the above example, Calculator is a **generic class** even if one of its methods is parameterized using generic type parameter. The T surrounded by < and > (angular brackets) is called the type parameter. Note that it is not mandatory that the class should be parameterized for the individual methods to be parameterized.

In the below code:

```
Calculator<Integer> calculator = new Calculator<Integer>();
Calculator<Float> floatCalculator = new Calculator<Float>();
int intTotal = calculator.sum(3, 7);
float floatTotal = calculator.sum(3.2f, 7.2f);
```

In the above example, **Integer** and **Float** are called type arguments passed to **Calculator** class. You will notice that method **sum** will be valid only if the type arguments are subclasses of Number. In such situations we use **bounded type parameters**, which we will learn later.

The type argument can be any one of the following **non-primitive** types:

1. any **class** type - eg: ArrayList**<Integer>**, HashMap**<String, String>**
2. any **interface** type - eg: ArrayList**<CharSequence>**
3. any **array** type - eg: ArrayList**<int[ ]>**, HashMap**<String, boolean[ ]>**
4. **nested generic type** arguments - eg: ArrayList**<Set<String>>**, HashMap**<String, List<Integer>>**

Type parameter names are usually single character uppercase letters. The convention used in Java is given below:

1. **E** - is used while working with **elements**. Almost all classes in collection framework which work with elements use this name as the type parameter name.
2. **K** - is used to denote the **key** in a key-value pair. Almost all classes in the Map hierarchy in collection framework use this name to denote a key.
3. **V** - is used to denote the **value** in a key-value pair. Almost all classes in the Map hierarchy in collection framework use this name to denote a value.
4. **T** - is used to denote a class or interface of any type.
5. **N** - is used to denote a Number.
6. We can use **S, U, V** and so on when we want to denote different types after the first type.

Select all the correct statements for the below code:

```
class A { // statement 1}
class B<T> { // statement 2}
class C<T> { // statement 3}
B<A> b1 = new B<A>(); // statement 4C<B<A>> c1 = new C<B<A>>(); // statement 5C c2 = new C(); // statement 6C<int> c4 = new C<int>(); // statement 7C<int[]> c3 = new C<int[]>(); // statement 8
```

- [ ] **Statement 3** will result in a compilation error. Since class B is already using a **type parameter** `T`, class `C` cannot use a **type parameter** with same name.
- [ ] **Statement 4** will result in a compilation error because only `String` or an `Integer` can be used as **type argument** and not a custom class such as `A`.
- [ ] **Statement 5** will result in compilation error. Since class `C` is a generic class that accepts only one type parameter and not a nested type parameter.
- [x] **Statement 6** will not result in compilation errors.
- [ ] **Statement 7** will not result in compilation error.
- [ ] **Statement 8** will result in compilation error.

Generics and Collections

Q1. All the collections in Java are parameterized using generic syntax. For example when we see the List interface we will find :

```
public interface List<E> extends Collection<E>{
        public boolean add(E e);
public E get(int index);
        ...
        ...
}
```

In the above example, List class is declared with a **type parameter** E.

As mentioned earlier collections which work with elements use E for the **type parameter** name, as a convention. It could have been Z or X or Y, however E is more intuitive to denote an element type.

In the below code:

```
List<String> namesList = new ArrayList<String>();   //Statement 1namesList.add("Hyderabad");
namesList.add("Bangalore");
namesList.add("Chennai");
for (String name : namesList) {                      //Statement 2System.out.println(name.substring(0, 3)); //Statement 3}
```

String is called type argument passed to **List** and **ArrayList**.

Since in **statement 1** the **type argument** is provided as String, in statement 2 we are able to directly iterate over the elements as type String instead of receiving it as an Object and later type casting the Object reference to String.

If the **type argument** String is omitted in statement 2, elements will be of type Object forcing us to type cast to appropriate type before using it.

Note that unless there is a strong reason to do it otherwise, collection classes should always be used by passing appropriate argument type.

Fill the missing code in the given program using the instructions given.

```java
package q11391;
import java.util.*;
public class SimpleArrayListDemo {
    public static void main(String[] args) {
        List<String> namesList = new ArrayList<String>();
        namesList.add("Hyderabad");

        //Add Bangalore to the namesList
        namesList.add("Bangalore");

        //Add Chennai to the namesList
        namesList.add("Chennai");

        for (String name : namesList) {

        // Print the String up to 3 characters using substring method
        System.out.println(name.substring(0,3));

        }
    }
}

```

Q2. Usage of Map interface with generics

Below code shows how to use generics while using a **Map**. Observe how we iterate only keys of the **Map**.

```java
package q11392;
import java.util.*;
public class SimpleMapDemo {
    public static void main(String[] args) {
        Map<String, String> countryCodesMap = new HashMap<String, String>();
        countryCodesMap.put("IN", "India");
        countryCodesMap.put("CA", "Canada");
        countryCodesMap.put("AG", "Argentina");
        countryCodesMap.put("BR", "Brazil");
        Set<String> codesSet = countryCodesMap.keySet();
        for (String code : codesSet) {
            String countryName = countryCodesMap.get(code);
            System.out.println(code + " is the code for : " + countryName);
        }
    }
}

```

Correct Usage of Generics

Q1. When a generic class or interface is used without passing any parameters as a normal class or interface, it is called a raw type.

For example:
class A {
}
class B<T> {
}
A a = new A(); *// statement 1*B<String> b1 = new B<String>(); *// statement 2*B b2 = new B(); *// statement 3*
In the above code, A is a normal class and class B is a generic type.

In **statement 3**, class B is called a raw type for the generic type B**<T>**.

In **statement 1**, class A is **not** called a raw type. Since A is a normal class.

We can assign a **parameterized type** to a **raw type**. For example:
b2 = b1; *// this is perfectly valid and allowed from the above code*
However, when we assign a **raw type** to a **parameterized type** we get a warning. For example:
b1 = b2; *// compiler will warn of **unchecked conversion***
Select all the correct statements in the below code.
List aList = new ArrayList(); *// statement 1*List<String> bList = new ArrayList<String>(); *// statement 2*List<String> cList = new ArrayList<>(); *// statement 3*List<String[]> dList = new ArrayList<String[]>(); *// statement 4*List<String> eList = new ArrayList(); *// statement 5*List fList = new ArrayList<String>(); *// statement 6*

- [x] **Statement 1** is an example of **raw type**.
- [x] **Statement 2** and **Statement 3** mean the same.
- [ ] **Statement 4** will result in compilation errors, since both List and ArrayList are parameterized with `E` as **type parameter**, meaning they can accept only individual **elements** and not a `String array`.
- [x] Compiler will produce a type conversion warning for **Statement 5**.
- [x] Compiler will not produce a type conversion warning for **Statement 6**.