CHAPTER

31

GENERIC TYPES

When we create a class with an instance variable to store an Integer object, it can be used to store Integer type data only. We cannot use that instance variable to store a Float class object or a String type object. This becomes a limitation. Let us take a class:

```
class A
{
     Integer x;
}
```

Now, it is not possible to store a String in this class A, since the instance variable x is of type Integer and it can store only an Integer object. To store String type value in class A, we can rewrite the class, as:

```
class A
{
     String x;
}
```

Now, can we store a Float class object into the instance variable x ? No, it is not possible. To do so, we can write another copy of class A, as:

```
class A
{
     Float x;
}
```

Like this, to store different types of data into a class, we have to write the same class again and again by changing the data type of the variables. This can be avoided if we use a 'generic class'.

Generic Class

A generic class represents a class that is type-safe. This means a generic class can act upon any data type. Similarly, a generic interface is also type-safe and hence it can use any data type. Generic classes and generic interfaces are also called 'parameterized types' because they use a parameter that determines which data type they should work upon.

When a generic class or generic interface is written, the programmer need not rewrite the same class or interface whenever he wants to use the class or the interface with a new data type. The

same class or interface can work with any data type. This becomes the greatest advantage for the programmers.

Important Interview Question

What is a generic type?

A generic type represents a class or an interface that is type-safe. It can act on any data type.

Generic types are designed to act upon objects. Hence they cannot work with primitive data types. Since, generic class acts on any data type, we cannot specify a particular data type when designing the class. In the place of the data type, we use a generic parameter like <T>, or <GT> and write the class as:

```
class Myclass<T>
{
     class code;
}
```

Here, <T> is called 'generic parameter' and it determines at the time of compilation, which data type actually the programmer wants to use with this class. In the remaining places where a data type is to be used, the programmer can use T, as shown below:

```
class Myclass<T>
{
     T obj;
}
```

Here, the programmer's intention is to store T type object where T represents any data type. The data type is specified by the programmer at the time of creating the object to Myclass, as:

```
Myclass<String> obj = new Myclass<String>();
```

Here, we are specifying <String> data type after Myclass name. This means, T assumes String data type and hence Java compiler creates the following class internally :

```
class Myclass
{
     String obj;
}
```

The above code is compiled by the compiler and is executed by JVM. The point is whenever a generic class or interface is written, Java compiler internally creates non-generic version of the class or interface by substituting the specified data type in place of generic parameter T. This is called 'erasure'.

Important Interview Question

What is erasure?

Creating non-generic version of a generic type by the Java compiler is called erasure.

Program 1: To understand how to create a generic class, let us write a program in which we will take a class by the name of 'Myclass'. In Myclass, we want to store an object of any data type.

```
//A generic class - to store any type of object
//here, T is generic parameter which determines the datatype
class Myclass<T>
```

```
{
        //declare T type object
        T obj;

        //a constructor to initialize T type object
        Myclass(T obj)
        {
                this.obj = obj;
        }

        //a method which returns T type object
        T getobj()
        {
                return obj;
        }
}
class Gen1
{
        public static void main(String args[])
        {

                //create Integer class object
                Integer i = 12; //This is same as: Integer i = new Integer(12);

                //create Myclass object and store Integer object in it
                Myclass<Integer> obj = new Myclass<Integer>(i);

                //retrieve Integer object by calling getobj()
                System.out.println("U stored: "+ obj.getobj());

                //In the same way, use Myclass for storing
                //Float object and retrieve it
                Float f = 12.123f;  //Same as:Float f = new Float(12.123f);
                Myclass<Float> obj1 = new Myclass<Float>(f);
                System.out.println("U stored: " +obj1.getobj());

                //we can use Myclass to store String type data also
                Myclass<String> obj2 = new Myclass<String>("Ravi Kumar");
                System.out.println("U stored: " +obj2.getobj());
        }
}
```

Output:

```
C:\> javac Gen1.java
C:\> java Gen1
U stored: 12
U stored: 12.123
U stored: Ravi Kumar
```

In Program 1, we created Myclass object 3 times and stored 3 different objects: Integer object, Float object and String objects into it.

To create an Integer object in this program, we have written:

```
Integer i = 12;
```

Observe that Integer is a wrapper class and we are storing directly 12 into its variable i. In this case, Java compiler creates Integer class object internally and stores a value 12 as shown below:

```
Integer i= new Integer(12);
```

This is called 'auto boxing'.

Important Interview Question

What is auto boxing?

Auto boxing refers to creating objects and storing primitive data types automatically by the compiler.

Generic Method

We can make a method alone as generic method, by writing the generic parameter before the method return type as:

```
<T> void display()
{
     method code;
}
```

In this case, the method becomes a generic method and can act on any data type. To represent the data type, we should use T inside the method.

Program 2: In this program, let us write a generic method that receives an array and displays the elements of the array. The array elements may be of any data type.

```
//A generic method - to read and display any type of array elements
class Myclass
{
     //This method accepts T type array
     static <T>void display(T[] arr)
     {
          //use for-each loop and read elements of array
          for(T i: arr)
               System.out.println(i);
     }
}
class Gen2
{
     public static void main(String args[])
     {
          //read elements from Integer type array using display()
          Integer arr1[] = {1,2,3,4,5,6};
          System.out.println("Reading Integer objects: ");
          Myclass.display(arr1);

          //read elements from Double type array using display()
          Double arr2[] = {1.1, 2.2, 3.3, 4.5};
          System.out.println("Reading Double objects: ");
          Myclass.display(arr2);

          //read elements from String type array using display()
          String arr3[] = {"Raju", "Rani", "Ravi", "Kiran"};
          System.out.println("Reading String objects: ");
          Myclass.display(arr3);

     }
}
```

Output:

```
C:\> javac Gen2.java
C:\> java Gen2
Reading Integer objects:
1
2
```

```
3
4
5
6
Reading Double objects:
1.1
2.2
3.3
4.5
Reading String objects:
Raju
Rani
Ravi
Kiran
```

Generic Interface

It is possible to develop an interface using generic type concept. An generic interface looks something like this:

```
interface Fruit<T>
{
      //method that accepts any object
      void tellTaste(T fruit);  //public abstract
}
```

Here, T represents any data type which is used in the interface. We know already that whenever there is an interface, we should also have implementation classes that implement all the methods of the interface. We can write an implementation class for the above interface, as:

```
class AnyFruit<T> implements Fruit<T>
{
      public void tellTaste(T fruit)
      {
            //write body for this method as needed
      }
}
```

Here, T represents generic parameter that shows any data type.

Program 3: In this program, let us see how to create a generic interface Fruit and a generic implementation class AnyFruit. Then the implementation class is used to decide the taste of the fruits: a Banana and an Orange.

```
//A generic interface
interface Fruit<T>
{
      //method that accepts any object
      void tellTaste(T fruit);  //public abstract

}
//this class implements Fruit interface
class AnyFruit<T> implements Fruit<T>
{
      public void tellTaste(T fruit)
      {
            //know the class name of the object passed to this method
            String fruitname = fruit.getClass().getName();

            //then decide the taste and display
            if(fruitname.equals("Banana"))
```

```
                System.out.println("Banana is sweet");
                else if(fruitname.equals("Orange"))
                System.out.println("Orange is sour");
        }
}
class Banana
{

}
class Orange
{

}
class Gen3
{
        public static void main(String args[])
        {
                //create Banana object and pass it to AnyFruit class
                Banana b = new Banana();
                AnyFruit<Banana> fruit1 = new AnyFruit<Banana>();
                fruit1.tellTaste(b);

                //create Orange object and pass it to AnyFruit class
                Orange o = new Orange();
                AnyFruit<Orange> fruit2 = new AnyFruit<Orange>();
                fruit2.tellTaste(o);

        }
}
```

Output:

```
C:\> javac Gen3.java
C:\> java Gen3
Banana is sweet
Orange is sour
```

Effect of generics on collections

After developing the concept of generic types in JDK1.5, JavaSoft people used the concept to re-write all the collection classes of `java.util` package. The classes are re-defined as:

```
HashSet<T>
LinkedHashSet<T>
Stack<T>
LinkedList<T>
ArrayList<T>
Vector<T>
HashMap<K,V>
Hashtable<K,V>
```

Where, T represents the type of the element. K and V represent types of elements marked as Key and Value pair.

To understand how the collection classes are re-written, let us write a program using Hashtable. We will begin by writing the code using versions prior to JDK1.5 version. Then we will rewrite the code using generic type concept added since JDK1.5 version.

Program 4: In this program, we have created a Hashtable with cricket players' names and their scores in a match. Here, we do not use generic type concept.

```
//Hashtable before jdk1.5 - no generic type concept
import java.util.*;
class HT1
{
```

```
        public static void main(String args[])
        {
                //create Hashtable object
                Hashtable ht = new Hashtable();  //generic types not used

                //store String type key and Integer type value
                ht.put("Ajay", new Integer(50));  //auto boxing is not used
                ht.put("Sachin", new Integer(90));
                ht.put("Dhoni", new Integer(75));

                //retrieve Sachin's score
                String s = "Sachin";
                Integer score =(Integer)ht.get(s);  //casting is required here
                System.out.println("Score= "+ score);

        }
}

C:\>javac HT1.java
```

Important Interview Question

HT1.java uses unchecked or unsafe operations.
Recompile with –Xlint:unchecked for details.

Please observe the messages given by the Java compiler above. It is saying unchecked operations are being performed. When creating Hashtable object we wrote:

```
Hashtable ht = new Hashtable();
```

Here, we did not mention what type of data being stored into ht. So the compiler was unaware of which type of data might be stored in the Hashtable. Hence it gives a message that says 'it is an unchecked operation'. If Hashtable is used as a generic type, then the programmer specifies the data type clearly, as:

```
Hashtable<String, Integer> ht = new Hashtable<String,Integer>();
```

This represents the Hashtable will be used to store a String as key and an Integer object as its value. In this case, the compiler would not make any complaint.

```
C:\>java HT1
Score= 90
```

The same program can be rewritten using generic type concept. In this case, the programmer informs the compiler which data types he is going to store in the Hashtable, at the time of creating the object, as:

```
Hashtable<String, Integer> ht = new Hashtable<String,Integer>();
```

Once, the compiler knows that the programmer wishes to store String type key and Integer type value, it will check the Hashtable object. This is done to verify if the programmer is actually storing the same data types or not. This is shown in Program 5.

Program 5: Here, we create a Hashtable with generic type concept so that it can be used to store any data type.

```
//Hashtable is rewritten using generic types in jdk1.5
//as Hashtable<K,V>
```

```
import java.util.*;
class HT2
{
      public static void main(String args[])
      {
            //create Hashtable object using String and Integer types
            Hashtable<String,Integer> ht = new Hashtable<String,Integer>();

            //store string type key and Integer type value
            ht.put("Ajay", 50);  //auto boxing is done
            ht.put("Sachin", 90);
            ht.put("Dhoni", 75);

            //retrieve Sachin's score
            String s = "Sachin";
            Integer score = ht.get(s);  //casting is not required
            System.out.println("Score= "+ score);

      }
}
```

Output:

```
C:\> javac HT2.java
C:\> java HT2
Score= 90
```

Please observe the differences between Program 4 and Program 5 where Hashtable is written using and without using generics.

After discussing generics, we can conclude the following points:

- A generic class or a generic interface represents a class or an interface which is type-safe.
- A generic class, generic interface or a generic method can handle any type of data.
- Generic types are defined as sub types of the class 'Object'.
- So, they act on objects of any class.
- They cannot act on primitive data types.
- Java compiler creates a non-generic version of the class by substituting the specified data type in a generic class. This is called erasure.
- By using generic types, it is possible to eliminate type casting in many cases.
- All the classes of java.util package have been rewritten using generic types.
- We cannot create an object to a generic parameter. For example,

```
class Myclass<T>  //T is generic parameter
T obj = new T(); //invalid
```

Conclusion

Generic classes in Java are similar to templates in C++ but there is considerable difference between them. The main aim of generic types is to provide type-safety for the classes and interfaces. A generic class, generic interface or a generic method eliminates the need of being re-written every time there is a change in a data type. The programmer can use same generic type on any data type and hence generics are very flexible and useful in writing the programs.