Generics in Java

Generics means **parameterized types**. The idea is to allow type (Integer, String, ... etc., and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types. An entity such as class, interface, or method that operates on a parameterized type is a generic entity.

Why Generics?

The **Object** is the superclass of all other classes, and Object reference can refer to any object. These features lack type safety. Generics add that type of safety feature. We will discuss that type of safety feature in later examples.

Generics in Java are similar to templates in C++. For example, classes like HashSet, ArrayList, HashMap, etc., use generics very well. There are some fundamental differences between the two approaches to generic types.

Types of Java Generics

Generic Method: Generic Java method takes a parameter and returns some value after performing a task. It is exactly like a normal function, however, a generic method has type parameters that are cited by actual type. This allows the generic method to be used in a more general way. The compiler takes care of the type of safety which enables programmers to code easily since they do not have to perform long, individual type castings.

Generic Classes: A generic class is implemented exactly like a non-generic class. The only difference is that it contains a type parameter section. There can be more than one type of parameter, separated by a comma. The classes, which accept one or more parameters, are known as parameterized classes or parameterized types.

Generic Class

Like C++, we use <> to specify parameter types in generic class creation. To create objects of a generic class, we use the following syntax.

```
// To create an instance of generic class
BaseType <Type> obj = new BaseType <Type>()
```

Note: *In Parameter type we can not use primitives like 'int','char' or 'double'.*

Java

```java
// Java program to show working of user defined
// Generic classes

// We use < > to specify Parameter type
class Test<T> {
    // An object of type T is declared
    T obj;
    Test(T obj) { this.obj = obj; } // constructor
    public T getObject() { return this.obj; }
}
```

```
// Driver class to test above
class Main {
    public static void main(String[] args)
    {
        // instance of Integer type
        Test<Integer> iObj = new Test<Integer>(15);
        System.out.println(iObj.getObject());

        // instance of String type
        Test<String> sObj
            = new Test<String>("GeeksForGeeks");
        System.out.println(sObj.getObject());
    }
}
```

Output

```
15
GeeksForGeeks
```

We can also pass multiple Type parameters in Generic classes.

Java

```
// Java program to show multiple
// type parameters in Java Generics

// We use < > to specify Parameter type
class Test<T, U>
{
    T obj1;  // An object of type T
    U obj2;  // An object of type U

    // constructor
    Test(T obj1, U obj2)
    {
        this.obj1 = obj1;
        this.obj2 = obj2;
    }

    // To print objects of T and U
    public void print()
    {
        System.out.println(obj1);
        System.out.println(obj2);
    }
}
```

```
// Driver class to test above
class Main
{
    public static void main (String[] args)
    {
        Test <String, Integer> obj =
            new Test<String, Integer>("GfG", 15);

        obj.print();
    }
}
```

Output

```
GfG
15
```

Generic Functions:

We can also write generic functions that can be called with different types of arguments based on the type of arguments passed to the generic method. The compiler handles each method.

Java

```
// Java program to show working of user defined
// Generic functions

class Test {
    // A Generic method example
    static <T> void genericDisplay(T element)
    {
        System.out.println(element.getClass().getName()
                           + " = " + element);
    }

    // Driver method
    public static void main(String[] args)
    {
        // Calling generic method with Integer argument
        genericDisplay(11);

        // Calling generic method with String argument
        genericDisplay("GeeksForGeeks");

        // Calling generic method with double argument
```

```
        genericDisplay(1.0);
    }
}
```

Output

```
java.lang.Integer = 11
java.lang.String = GeeksForGeeks
java.lang.Double = 1.0
```

Generics Work Only with Reference Types:

When we declare an instance of a generic type, the type argument passed to the type parameter must be a reference type. We cannot use primitive data types like **int**, **char.**

```
Test<int> obj = new Test<int>(20);
```

The above line results in a compile-time error that can be resolved using type wrappers to encapsulate a primitive type.

But primitive type arrays can be passed to the type parameter because arrays are reference types.

```
ArrayList<int[]> a = new ArrayList<>();
```

Generic Types Differ Based on Their Type Arguments:

Consider the following Java code.

Java

```
// Java program to show working
// of user-defined Generic classes

// We use < > to specify Parameter type
class Test<T> {
    // An object of type T is declared
    T obj;
    Test(T obj) { this.obj = obj; } // constructor
    public T getObject() { return this.obj; }
}
```

```
// Driver class to test above
class Main {
    public static void main(String[] args)
    {
        // instance of Integer type
        Test<Integer> iObj = new Test<Integer>(15);
        System.out.println(iObj.getObject());

        // instance of String type
        Test<String> sObj
            = new Test<String>("GeeksForGeeks");
        System.out.println(sObj.getObject());
        iObj = sObj; // This results an error
    }
}
```

Output:

```
 error:
  incompatible types:
  Test cannot be converted to Test
```

Even though iObj and sObj are of type Test, they are the references to different types because their type parameters differ. Generics add type safety through this and prevent errors.

Type Parameters in Java Generics

The type parameters naming conventions are important to learn generics thoroughly. The common type parameters are as follows:

- T – Type
- E – Element
- K – Key
- N – Number
- V – Value

Advantages of Generics:

Programs that use Generics has got many benefits over non-generic code.

1. Code Reuse: We can write a method/class/interface once and use it for any type we want.

2. Type Safety: Generics make errors to appear compile time than at run time (It's always better to know problems in your code at compile time rather than making your code fail at run time). Suppose you want to create an ArrayList that store name of students, and if by mistake the programmer adds an integer object instead of a string, the compiler allows it. But, when we retrieve this data from ArrayList, it causes problems at runtime.

Java

```
// Java program to demonstrate that NOT using
// generics can cause run time exceptions

import java.util.*;

class Test
{
    public static void main(String[] args)
    {
        // Creatinga an ArrayList without any type specified
        ArrayList al = new ArrayList();

        al.add("Sachin");
        al.add("Rahul");
        al.add(10); // Compiler allows this

        String s1 = (String)al.get(0);
        String s2 = (String)al.get(1);

        // Causes Runtime Exception
        String s3 = (String)al.get(2);
    }
}
```

Output :

```
Exception in thread "main" java.lang.ClassCastException:
   java.lang.Integer cannot be cast to java.lang.String
    at Test.main(Test.java:19)
```

How do Generics Solve this Problem?

When defining ArrayList, we can specify that this list can take only String objects.

Java

```
// Using Java Generics converts run time exceptions into
// compile time exception.
import java.util.*;

class Test
{
    public static void main(String[] args)
    {
        // Creating a an ArrayList with String specified
        ArrayList <String> al = new ArrayList<String> ();

        al.add("Sachin");
        al.add("Rahul");

        // Now Compiler doesn't allow this
        al.add(10);

        String s1 = (String)al.get(0);
        String s2 = (String)al.get(1);
        String s3 = (String)al.get(2);
    }
}
```

Output:

```
 15: error: no suitable method found for add(int)
         al.add(10);
           ^
```

3. Individual Type Casting is not needed: If we do not use generics, then, in the above example, every time we retrieve data from ArrayList, we have to typecast it. Typecasting at every retrieval operation is a big headache. If we already know that our list only holds string data, we need not typecast it every time.

Java

```
// We don't need to typecast individual members of ArrayList

import java.util.*;

class Test {
    public static void main(String[] args)
    {
        // Creating a an ArrayList with String specified
```

```
        ArrayList<String> al = new ArrayList<String>();

        al.add("Sachin");
        al.add("Rahul");

        // Typecasting is not needed
        String s1 = al.get(0);
        String s2 = al.get(1);
    }
}
```

4. Generics Promotes Code Reusability: With the help of generics in Java, we can write code that will work with different types of data. For example,

Let's say we want to Sort the array elements of various data types like int, char, String etc.

Basically we will be needing different functions for different data types.

For simplicity, we will be using Bubble sort.

But by using **Generics,** we can achieve the code reusability feature.

Java

```
public class GFG {

    public static void main(String[] args)
    {
```



```
        Character[] c = { 'v', 'g', 'a', 'c', 'x', 'd', 't' };

        String[] s = { "Virat", "Rohit", "Abhinay", "Chandu","Sam", "Bharat", "Kalam" };

        System.out.print("Sorted Integer array :  ");
        sort_generics(a);

        System.out.print("Sorted Character array :  ");
        sort_generics(c);

        System.out.print("Sorted String array :  ");
        sort_generics(s);

    }

    public static <T extends Comparable<T> > void sort_generics(T[] a)
    {

         //As we are comparing the Non-primitive data types
```

```
          //we need to use Comparable class

        //Bubble Sort logic
        for (int i = 0; i < a.length - 1; i++) {

            for (int j = 0; j < a.length - i - 1; j++) {

                if (a[j].compareTo(a[j + 1]) > 0) {

                    swap(j, j + 1, a);
                }
            }
        }

        // Printing the elements after sorted

        for (T i : a)
        {
            System.out.print(i + ", ");
        }
        System.out.println();

    }

    public static <T> void swap(int i, int j, T[] a)
    {
        T t = a[i];
        a[i] = a[j];
        a[j] = t;
    }

}
```

Output

```
Sorted Integer array :  6, 22, 41, 50, 58, 100,
Sorted Character array :  a, c, d, g, t, v, x,
Sorted String array :  Abhinay, Bharat, Chandu, Kalam, Rohit, Sam, Virat,
```

Here, we have created a generics method. This same method can be used to perform operations on integer data, string data, and so on.

5. Implementing Generic Algorithms: By using generics, we can implement algorithms that work on different types of objects, and at the same, they are type-safe too.

This article is contributed by **Dharmesh Singh**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to review-