SNS COLLEGE OF TECHNOLOGY

(AN AUTONOMOUS INSTITUTION)

COIMBATORE – 35

DEPARTMENT OF COMPUTER SIENCE AND ENGINEERING

UNIT IV

MULTITHREADING IN JAVA

Generic Programming

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 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 Method:

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

Bounded Types

There may be times when you want to restrict the types that can be used as type arguments in a parameterized type. For example, a method that operates on numbers might only want to accept instances of Numbers or their subclasses. This is what bounded type parameters are for.

- Sometimes we don't want the whole class to be parameterized. In that case, we can create a Java generics method. Since the constructor is a special kind of method, we can use generics type in constructors too.
- Suppose we want to restrict the type of objects that can be used in the parameterized type. For example, in a method that compares two objects and we want to make sure that the accepted objects are Comparables.
- The invocation of these methods is similar to the unbounded method except that if we will try to use any class that is not Comparable, it will throw compile time error.

How to Declare a Bounded Type Parameter in Java?

1. List the type parameter's name,
2. Along with the extends keyword
3. And by its upper bound. (which in the below example c is **A**.)

Syntax

<T extends **superClassName**>

Note that, in this context, extends is used in a general sense to mean either "extends" (as in classes). Also, This specifies that T can only be replaced by superClassName or subclasses of superClassName. Thus, a superclass defines an inclusive, upper limit.

Let's take an example of how to implement bounded types (extend superclass) with generics.

Java

```
// This class only accepts type parameters as any class

// which extends class A or class A itself.

// Passing any other type will cause compiler time error



class Bound<T extends A>

{
```

```
    private T objRef;



    public Bound(T obj){
        this.objRef = obj;
    }



    public void doRunTest(){
        this.objRef.displayClass();
    }
}



class A
{
    public void displayClass()
    {
        System.out.println("Inside super class A");
    }
}



class B extends A
{
```

```
    public void displayClass()
    {
        System.out.println("Inside sub class B");
    }
}


class C extends A
{
    public void displayClass()
    {
        System.out.println("Inside sub class C");
    }
}


public class BoundedClass
{
    public static void main(String a[])
    {


        // Creating object of sub class C and
        // passing it to Bound as a type parameter.
```

```
        Bound<C> bec = new Bound<C>(new C());

        bec.doRunTest();



        // Creating object of sub class B and

        // passing it to Bound as a type parameter.

        Bound<B> beb = new Bound<B>(new B());

        beb.doRunTest();



        // similarly passing super class A

        Bound<A> bea = new Bound<A>(new A());

        bea.doRunTest();



    }

}
```

Output

Inside sub class C

Inside sub class B

Inside super class A