Course code : **CSE2005**

Course title : **Object Oriented Programming**

Generic Mtheods Interfaces

Objectives

This session will give the knowledge about

- Generic Methods
- Generic Constructors
- Generic Interfaces
- Generic Hierarchies

Generic Methods

You can write a single generic method declaration that can be called with arguments of different types. Following are the rules to define Generic Methods:

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).
- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.

Generic Methods

- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.

- A generic method's body is declared like that of any other method.

- Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Generic Methods

```
class Avg{
        String intInput(int inp){  //Normal Method
                return inp+"".getClass().getName();
        }
        <Type> String IntegerInput(Type inp){ //Generic method
                return inp.getClass().getName();
        }
        static <Type> String staticInput(Type[] inp){ //static Generic method
                return inp.getClass().getName();
        }
}
```

Generic Methods

```
public class GenericDemo {
        public static void main(String[] args) {
                Avg obj=new Avg();
                System.out.println(obj.intInput(23));
                System.out.println(obj.IntegerInput(34.45));
                Double[] dary={34.23,45.23};
                System.out.println(Avg.staticInput(dary));
        }
}
```

Generic Methods with Bounded Types

```
class Avg{
        <Type extends Number> String IntegerInput(Type inp){ //upper bound
                return inp+" is now "+inp.getClass().getName();
        }
}
public class GenericDemo {
        public static void main(String[] args) {
                Avg obj=new Avg();
                System.out.println(obj.IntegerInput(23.34f));
                System.out.println(obj.IntegerInput(23));
                System.out.println(obj.IntegerInput(23d));
        }
}
```

Generic Methods with Bounded Types

```
class Avg{
        <Type extends Integer> String IntegerInput(Type inp){ //lower bound
                return inp+" is now "+inp.getClass().getName();
        }
}
public class GenericDemo {
        public static void main(String[] args) {
                Avg obj=new Avg();
                System.out.println(obj.IntegerInput(23.34f));//error
                System.out.println(obj.IntegerInput(23));
                System.out.println(obj.IntegerInput(23d));//error
        }
}
```

Generic Constructors

It is possible for constructors to be generic, even if their class is not.

```
class Avg{
        <Type> Avg(){
                System.out.println("constructor for "+this.getClass().getName());
        }

        <Type extends Number> Avg(Type inp){
                System.out.println("constructor for "+inp.getClass().getName());
        }
}
```

Generic Constructors

```
public class GenericDemo {
        public static void main(String[] args) {
                Avg obj1=new Avg();
                Avg obj2=new Avg(23);
                Avg obj3=new Avg(23f);
        }
}
```

Generic Interfaces

In addition to generic classes and methods, you can also have generic interfaces.

```
interface Sample <Type>{
        void show(Type t);  }
public class GenericDemo implements Sample {
        public static void main(String[] args) {
                new GenericDemo().show(10.45);
        }
        public void show(Object t) {
                System.out.println(t.getClass().getName());
        } }
```

Generic Interfaces

```
interface Sample <Type extends Number>{
        void show(Type t);
}
public class GenericDemo implements Sample {
        public static void main(String[] args) {
                new GenericDemo().show("34"); //Error
        }

        public void show(Number t) {
                System.out.println(t.getClass().getName());
        }
}
```

Generic Interfaces

```
interface Mathematics<T extends Number> {
   int square(T t);
}
class Demo<T extends Integer> implements Mathematics<T>{
         public int square(T t) {
                  return t.intValue()*t.intValue();
         }
}
public class GenericDemo {
         public static void main(String[] args) {
                  System.out.println(new Demo().square(12));
         }
}
```

Generic Class Hierarchies

Generic classes can be part of a class hierarchy in just the same way as a nongeneric class. Thus, a generic class can act as a superclass or be a subclass.

The key difference between generic and non-generic hierarchies is that in a generic hierarchy, any type arguments needed by a generic superclass must be passed up the hierarchy by all subclasses.

This is similar to the way that constructor arguments must be passed up a hierarchy.

Generic Class Hierarchies

```
class Base<T>{
        T t;
        Base(T t){
                this.t=t;
                System.out.println(t.getClass().getName().toString());
        }
}
class Derived<T> extends Base<T>{
        Derived(T t){
                super(t);
                System.out.println(t.getClass().getName().toString());
        }
```

Generic Class Hierarchies

```
}
public class GenericDemo {
        public static void main(String[] args) {
                Derived<Object> obj=new Derived<Object>(10.89);
        }
}
```

Generic Class Hierarchies

```
class Base<T>{
        T t;
        Base(T t){
                this.t=t;
                System.out.println(t.getClass().getName().toString());
        }
}
class Derived<T, V> extends Base<V>{
        Derived(T t, V v){
                super(v);
                System.out.println(t.getClass().getName().toString());
        }
```

Generic Class Hierarchies

```
}
public class GenericDemo {
        public static void main(String[] args) {
                Derived<Object,Object> obj=new Derived<>(10.89,23);
        }
}
```

Generic Class Hierarchies

```
class Base{
        Base(){
                System.out.println("i am non gen base");
        }
}
class Derived<T> extends Base{
        Derived(T t){
                super();
                System.out.println(t.getClass().getName().toString());
        }
}
```

Generic Class Hierarchies

```
public class GenericDemo {
        public static void main(String[] args) {
                Derived<Object> obj=new Derived<>(10.89);
        }
}
```

Generic Restrictions

Type Parameters Can’t Be Instantiated

```
class Gen<T>{
        T t;
        Gen(){
                t=new T(); //illegal
        }
}
```

Generic Restrictions

Restrictions on Static Members

```
class Gen<T>{
        static T t;      //illegal
        static T getT(){  //illegal
                return t;
        }
}
```

Generic Restrictions

Generic Array Restrictions

```
class Gen<T>{
        T[] t;
        Gen(){
                t=new T[10]; //illegal
        }
}
```

Generic Restrictions

Generic Exception Restriction: A generic class cannot extend Throwable. This means that you cannot create generic exception classes.

```
class Gen<T> extends Throwable{ //illegal
        T[] t;
        Gen(){
        }
}
```

Summary

We have discussed about

- Generic Methods
- Generic Constructors
- Generic Interfaces
- Generic Hierarchies