Generics (1.5V)

1. Introduction
2. Generic Classes
3. Bounded Types
4. Generic Methods
5. Wildcard Characters
6. Communication with non-generic code
7. Conclusion

Case 1

- Arrays are always safe w.r.t type .
- For example , if our programming requirements is to add only String objects then we can go for String[] array . For this array we can add only String type object , by mistake if we are trying to add any other type we will get Compile Time Error .

Example:

```
String[] s = new String [600];
s[0]="prolog" ;
s[1]="Academy" ;
s[2]=new Student() ; //It will give CE
        CE:         Incompatible Type
        Found :     Student
        Required :  String
```

- Hence in the case of array we can always give guarantee about the type of elements .
- String[] array contains only String Objects due to this array are always safe to use w.r.t type .
- But Collection are not safe to use w.r.t type .
- For example if our programing requirements is to hold only String Objects & if we are using ArrayList , By mistake if we are trying to add any other type to the list we won't get any Compile Time Error , But program may fail at runtime .

Example:

```
ArrayList  al  =  new ArrayList( );
al.add("Prolog");
al.add("Academy");
al.add( new Student() );
                .
String name1 = (String) al.get(0) ;    (Correct)
String name2 = (String) al.get(1) ;    (Correct)
String name3 = (String) al.get(2) ;    (Wrong)
```

RE: ClassCastException

- There is no guarantee that collection can hold a particular type of objects . Hence w.r.t type Collection are not safe to use .

Case 2

- In the case of Arrays at the time of retrieved it is not required to perform any Typecasting .

Example:

```
String[] s = new String [600];
s[0]="prolog" ;
s[1]="Academy" ;
        …
        …
String name1 = s[0] ;  //TypeCasting is not Required
```

- But in the case of Collection at the time of retrieval compulsory we should perform Typecasting otherwise we will get compile Time Error .

Example:

```
ArrayList  al  =  new ArrayList( );
al.add("Prolog");
        .
String name1 = al.get(0) ;
```

CE: Incompatible Type
Found : Object
Required : String

But

```
String name1 = (String) al .get(0) ;
```

(Correct)

- Hence in the case of Collection Type Casting is mandatory which is a bigger headache to the programmer .
- To overcome the above problem of collection (Type Safe & Type Casting) Sun people introduced Generics concept in 1.5 Version .
- Hence the main objective of generics concepts are
    1. To provide Type safety to the collection . So that they can hold a particular type of objects .
    2. To Resolve Type Casting Problem .

Example:

- To hold only String type of Objects a generic version of ArrayList we can declare as follows.

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

Base Type

Parameter

- For this ArrayList we can add only String type of Objects , by mistake if we are trying to add any other type we will get Compile Time Error .

  i.e We are getting Type-Safety

```
import java.util.*;
public class Test1 {
        public static void main(String[] args) {
                ArrayList<String> al = new ArrayList<String>();
                al.add("Prolog");
                al.add("Academy");
                al.add("10");
                //al.add(10);
                System.out.println(al);

        }

}
```

CE : Cannot Find Symobol
Symbol :Method add(int) ;
Location : Class ArrayList<String>

- At the time of retrieval it is not required to perform any TypeCasting .

Example:

```
String name = al.get(0) ;
```

Conclusion 1

- Usage of parent class reference to hold Child class Object is considered as polymorphism .
- Polymorphism concept is applicable only for base type but not for parameter type .

Example:

```
ArrayList<Integer> al = new ArrayList<Integer>( );    (Correct)
List<Integer> al = new ArrayList<Integer>( );         (Correct)
Collection<Integer> al = new ArrayList<Integer>( );   (Correct)
List<Object> al = new ArrayList<Integer>( );          (Wrong)
```

CE : Incompatable Type

Found :ArrayList<Integer>;

Required : List<Object>

Conclusion 2

- For the parameter we can use any classes or interface name & we can't use primitive type .
- Violation leads to Compile Time Error .

Example:

```
ArrayList<Int>  al  =  new ArrayList<int>( );
```

CE : Unexpected Type
Found :int
Required : Reference

CE : Unexpected Type
Found :int
Required : Reference

Generic Classes

- Until 1.4 version a non-generic version of ArrayList class is declared as follows .

```
class ArrayList
{
        void add(object o)
        {
                //doStuff
        }
        Object get(int  index)
        {
                //doStuff
        }
}
```

- The argument to add() method is Object . Hence we can add any type of Objects due to this we are not getting Type Safety .
- The Return Type of get() method is Object , Hence at the time of retrieval Compulsory we should Perform Type Casting .

Generics (1.5V)

- But in java 1.5 Version a generic Version of ArrayList class is declared as follows .

```
class ArrayList <T>
{
        void add(T   t)
        {
                //doStuff
        }
        T get(int  index)
        {
                //doStuff
        }
}
```

Type Parameter

- Based on our requirement Type Parameter “T” will be replaced with corresponding provided Type .

Example:

- To hold only String type of Objects we have to create Generic Version of ArrayList Object as follows.

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

- For this requirement the corresponding loaded version of ArrayList class is .

```
class ArrayList <String>
{
        void add(String   t)
        {
                //doStuff
        }
        String get(int  index)
        {
                //doStuff
        }
}
```

Generics (1.5V)

- add() method can take String as the parameter , hence we can add only String type of Object . But by mistake if we are trying to add any other type we will get Compile Time Error . i.e , We are getting Type Safety .

- The return type of get( ) method is String , hence at the time of retrieval we can assign directly to the String type Variable , it is not required to perform any type Casting ..

```
class ArrayList <String>
{
        void add(String   t)
        {
                //doStuff
        }
        String get(int  index)
        {
                //doStuff
        }
}
```

Note

1. As the type parameter we can use any valid java identifier but it is a convention to use <T> .

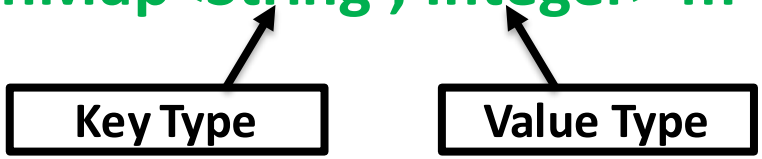
```
Example
Class Al <X>
{
        //doStuff;
}
```

```
Example
Class Al <ProLog>
{
        //doStuff;
}
```

2. We can pass any number of type parameter & need not to be one .

```
class HashMap < K , V >
{
        //doStuff;
}
```

```
HashMap<String , Integer> m = new HashMap<String,Integer>();
```



Generics (1.5V)

- Though in Generics we are associating a type parameter to the classes . Such type of parameterized classes are called Generic Class .

- We can define our own Generic classes also .

Generics (1.5V)

```
class Gen<T>
{
        T ob;
        Gen(T ob)
        {
                this.ob=ob;
        }
        public void show()
        {
        S.o.p("The type of ob is :"+ob.getClass());
        }
        public T getOb()
        {
                return ob;
        }
}
```

```
public class GenDemo
{
        public static void main(String[] args)
        {
                Gen<String> g1=new Gen<String>("prolog");
                g1.show();
                System.out.println(g1.getOb());

                Gen<Integer> g2=new Gen<Integer>(10);
                g2.show();
                System.out.println(g2.getOb());

        }

}
```

Bounded Types

- We can bound the type parameter for a particular range by using extend keyword .

```
class Test < T >
{
        //doStuff;
}
```

- As the type parameter we can pass any type hence it is unbounded type .

```
Test< String > m = new Test< String>();
Test< Integer > m = new Test< Integer >();
```

```
class Test < T  extends Number>
{
        //doStuff;
}
```

- As the type parameter we can pass either Number type or its child classes , hence it is Bounded type .

```
Test< Integer > m = new Test< Integer >();        (Correct)
Test< String > m = new Test< String>();           (Wrong)
```

CE : Type parameter java.lang.String is not within its bound

- We can't bound type parameter by using implements & super keyword .

```
class Test < T  implements Runnable>
{
        //doStuff;
}
```

Wrong

```
class Test < T  super Integer>
{
        //doStuff;
}
```

Wrong

- But , implements keyword purpose we can survive by using extend Keyword only .

```
class Test < T  extends  x>
{
        //doStuff;
}
```

- X can be either class / interface .
- If X is a class than , as the type parameter we can provide either x type or its child classes .
- If x is an interface than as the type parameter we can provide either x type or its implementation classes .

Example:

```
class Test < T  extends  Runnable>
{
        //doStuff;
}
```

```
Test< Runnable >  al  =  new Test< Runnable >( );   (Correct)
Test< Thread >  al  =  new Test< Thread >( );       (Correct)
Test< String >  al  =  new Test< String >( );       (Wrong)
```

CE : Type parameter java.lang.String is not within its bound

- We can also bound the parameter even in Combination also .

```
class Test < T  extends  Number & Runnable >
{
        //doStuff;
}
```

- As the type parameter we can pass any type which is the child class of Number & implements Runnable interface .

1. class Test < T extends Runnable & Comparable > (Correct)
2. class Test < T extends Number & Runnble & Comparable > (Correct)
3. class Test < T extends Number & Thread > (Wrong)

   We can't extend more than one class at a time .
4. class Test < T extends Runnable & Number > (Wrong)

   We have to take first class & then interface .

Generic Method & Wildcard Characters

Example

```
void m1(ArrayList<String>  l) {        }
```

- **This method is applicable for ArrayList<String> (ArrayList of any String Type) .**
- **Within the method we can add String type Objects & null to the List , if we are trying to add any other type we will get Compile Time Error .**

```
void m1(ArrayList<String>  l)
{
        l.add("A");
        l.add(null);
        l.add(10);      //CE
}
```

Example

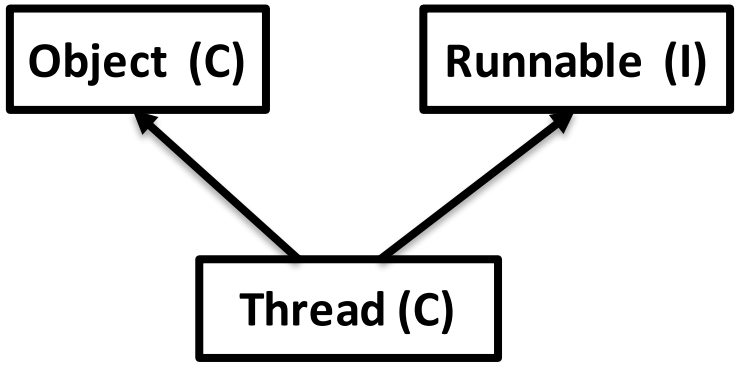
```
void m1(ArrayList<? extends  X>  l)
{
        doStuff();
}
```

- If X is a class then we can call this method by passing ArrayList of either X type or its child classes
- If X is an interface then we can call this method by passing ArrayList of either X type or its implementation class .
- In this case we can't add any type of the element to the List except null .

Example

```
void m1(ArrayList<? super X>  l)
{
        doStuff();
}
```

- If X is a class then we can call this method by passing ArrayList of either X type or its super classes
- If X is an interface then we can call this method by passing ArrayList of either X type or super classes of implementation class of x .



Example

```
void m1(ArrayList<? > l)
{
        doStuff();
}
```

- It simply means “anyType” . It can be <Dog> , <Integer> , <JButton> etc .

```
public void m1(List<?>  l) { }
public void m1(List<Object>  l) { }
```

Both Are Same ?

- **No , List<?> which is the wildcard <?> without the keyword extend or super simply means "anything" . Eg List<Dog> , List<Integer> .**
- **But List<Object> means that the method can take only a List<Object> not a List<Dog> , or List<Integer> .**

Which of the following declarations are valid

1. ArrayList<String> l=new ArrayList<String>();
2. ArrayList<?> l=new ArrayList<String>();
3. ArrayList<? extends String> l=new ArrayList<String>();
4. ArrayList<? extends Object> l=new ArrayList<String>();
5. ArrayList<? extends Number> l=new ArrayList<String>();
6. ArrayList<? extends Number> l=new ArrayList<Integer>();
7. ArrayList<?> l=new ArrayList<? extends Number>();
8. ArrayList<?> l=new ArrayList<?>();

Correct : - 1, 2, 3, 4,6 .

Wrong :- 5,7,8 .

- We can define the type parameter either at class level or at method-level .

Declaration of type parameter at class level

```
class Test <T>
{
        T  ob;
        public T get()
        {
                return ob ;
        }
}
```

Declaration of type parameter at Method level

- We have to declare the type parameter just before return type .

```
class Test
{
        public <T> void  get(T  t)
        {
                T ob ;
        }
}
```

At declaration we can have

1. < T extends Number >
2. < T extends Runnable>
3. < T extends Number & Runnable>
4. < T extends Runnable & Comparable >
5. < T extends Number & Thread >
6. < T extends Runnable & Thread>

Ans: 5 and 6 are Wrong

Communication with non-generic code

- To provide compatibility with old version SUN people compromised the concept of Generic in very few are .

```
import java.util.ArrayList;
public class Test2
{
        public static void main(String[] args)
        {
                ArrayList<String> al = new ArrayList<String>();
                al.add("Prolog");
                //al.add(10);     //CE:
                m1(al);
                System.out.println(al);
                //al.addAll(10)  //CE
        }
        public static void m1(ArrayList l)
        {
                l.add(10);
                l.add(10.5);
                l.add(true);

        }
}
```

Generic Area

Non Generic Area