14. Generics (1.5 version)

1. Introduction
2. Generic classes
3. Bounded types
4. Generic methods & wild-card character (?)
5. Communication with non-Generic code
6. Conclusions

1) Introduction :—

→ The main purpose of Generics is to provide Type safety & to resolve type casting problems.

Case (i): Type safety :—

→ Arrays are always Type safe i.e., we can give the guarantee for the type of elements present inside array.

→ For example, our programming requirement is to store String objects. We can choose String[]. By mistake if we are trying to provide any other type we will get compile time error.

Ex: String[] s=new String[2500];

✓ s[0] = "durga";

✓ s[1] = "ravi";

✗ s[2] = new Integer(10); → CE: incompatible types found: j.l.Integer required: j.l.String

✓ s[2] = "shiva";

i.e., we can give the guarantee that String array can contain only String type of objects.

Hence with respect to type Arrays are safe to use i.e., Arrays are type safe.

→ Collections are not type safe i.e., we can't give the guarantee for the type of elements present inside Collection.

For example, if our programming requirement is to hold only String type of objects & if we choose ArrayList. By mistake if we are trying to add any other type then we won't get any CE, but the program may fail at runtime.

Ex:
```
ArrayList l = new ArrayList();
l.add("durga");
l.add("ravi");
l.add(new Integer(10));
```

```
String name1 = (String) l.get(0);
String name2 = (String) l.get(1);
String name3 = (String) l.get(2); → RE : ClassCastException
```

i.e., we can't give the guarantee for the type of elements present inside Collection.

Hence Collections are not safe to use w.r.t. type i.e., Collections are not type safe.

Case(ii): Type casting :—

→ In case of Arrays, at the time of retrieval we are not required to perform type casting.

Ex:
```
String[] s = new String[2500];
s[0] = "durga";
```

```
String name1 = s[0];
```
→ type casting is not required

→ But in case of Collection at the time of retrieval compulsory we have to perform type casting.

Ex:
```
ArrayList l=new ArrayList();
l.add("durga");
  :
```

✗ `String name1 = l.get(0);` → CE: incompatible types
found: j.l.Object
required: j.l.String

`String name1 = (String)l.get(0);` → Type casting is mandatory

→ Hence type casting is a bigger headache in Collections.

→ To overcome above problems we should go for Generics.

→ Hence the main purposes of Generics are

① To provide type safety

② To resolve type casting problems.

For example, if our programming requirement is to hold only String type of objects we can create Generic version of ArrayList object as follows.

```
ArrayList<String> l=new AL<String>();
```

→ For this AL object we can add only String type of objects. By mistake if we are trying to add any other type we will get CE.

Ex:
```
l.add("durga");
l.add("ravi");
✗ l.add(new Integer(10)); → CE
```

→ Hence through Generics we are getting type safety.

→ At the time of retrieval we are not required to perform any type casting.

Ex:
```
AL<String> l=new AL<String>();
l.add("durga");
String name1 = l.get(0);
```
→ Type casting is not required

→ Hence through Generics we can resolve type casting problems.

AL l=new AL();	AL<String> l=new AL<String>();
1. It is a non-Generic version of AL object.	1. It is a Generic version of AL object.
2. For this AL object we can add any type of objects and hence it is not type safe.	2. For this AL object we can add only String type of objects and hence it is type safe.
3. At the time of retrieval compulsory we should perform type casting.	3. At the time of retrieval we are not required to perform type casting.

Conclusion ①:-

→ Polymorphism concept applicable only for the Base type, but not for parameter type (Usage of parent reference to hold child object is called polymorphism).

Ex:
```
AL<String> l=new AL<String>();
List<String> l=new AL<String>();
Collection<String> l=new AL<String>();
```
(AL<String> : parameter type; AL : Base type)

AL<Object> l=new AL<String>(); → CE: incompatible types
found: AL<String>
required: AL<Object>

Conclusion②:-

→ For the type parameter we can provide any class or interface name, but not primitives, otherwise we will get CE.

Ex: AL<int> l=new AL<int>();

CE: unexpected type
found: int
required: reference

2) Generic classes:-

→ Until 1.4 version AL class is declared as follows.

Ex:
```
class AL
{
    add(Object o)
    Object get(int index)
}
```

→ The argument to the add(-) method is Object. Hence we can add any type of object to the AL.
→ Due to this we are missing type safety.
→ The return type of get(-) method is Object. Hence at the time of retrieval we should perform type casting.
→ But in 1.5 version a generic version of AL class is declared as follows.

Ex:
```
class AL<T>        ← Type parameter
{
    add(T ob)
    T get(int index)
}
```

→ Based on our requirement 'T' will be replaced with our provided type.

For example, if our programming requirement is to store only String type of objects we can create generic version of AL object as follows.

```
AL<String> l=new AL<String>();
```

→ For this AL object compiler consider version of AL class is as follows.

```
class AL<String>
{
    add(String ob)
    String get(int index)
}
```

→ The argument to add() method is String & hence we can add only String type of objects. By mistake if we are trying to add any other type then we will get CE.

Ex: l.add("durga");

✗ l.add(new Integer(10));

CE: cannot find symbol
symbol: method add(Integer)
location: class AL<String>

→ Hence through Generics we are getting Type Safety.

→ The return type of get() method is String & hence at the time of retrieval we are not required to perform type casting.

Ex: String name1 = l.get(0);

type casting is not required

→ In Generics we are associating a type parameter for the classes. Such type of parameterized classes are called

Generic classes (Template classes).

→ Based on our requirement we can create our own Generic classes.

Ex:
```
class Account<T>
{
  ==
}

Account<Gold> a1=new Account<Gold>();
Account<Platinum> a2=new Account<Platinum>();
```

Ex:
```
class Gen<T>
{
  T ob;
  Gen(T ob)
  {
    this.ob = ob;
  }
  public void show()
  {
    S.o.p("The type of ob:"+ob.getClass().getName());
  }
  public T getOb()
  {
    return ob;
  }
}
class GenDemo
{
  p s v m(-)
  {
    Gen<String> g1=new Gen<String>("durga");
    g1.show(); ⇒ (O/P: The type of ob: j.l.String)
    S.o.p(g1.getOb()); ⇒ durga

    Gen<Integer> g2=new Gen<Integer>(10);
    g2.show(); ⇒ (O/P: The type of ob: j.l.Integer)
```

S.o.p(g2.getOb()); ⇒ o/p : 10

Gen<Double> g3=new Gen<Double>(10.5);
g3.show(); ⇒ o/p : The type of ob : j.l.Double
S.o.p(g3.getOb()); ⇒ o/p : 10.5
}
}

3) Bounded Types :—

→ We can bound type parameter for a particular range by using extends keyword. Such types are called Bounded Types.

Ex: class Test<T>
{
}

As the type parameter we can pass any type & there are no restrictions. Hence it is unbounded type.

Test<Integer> t1=new Test<Integer>(); ✓
Test<String> t2=new Test<String>(); ✓

Syntax for Bounded Type :—

class Test<T extends X>
{
≡
}

X can be either class or interface

→ If X is a class then as the type parameter we can pass either X type or its child classes.

→ If X is an interface then as the type parameter we can pass either X type or its implementation classes.

Ex: class Test<T extends Number>
{
≡
}

✓ Test<Integer> t1 = new Test<Integer>();

✓ Test<Double> t2 = new Test<Double>();

✗ Test<String> t3 = new Test<String>();

CE: Type parameter j.l.String is not in its bound

Ex: class Test<T extends Runnable>
{
≡
}

✓ Test<Runnable> t1 = new Test<Runnable>();

✓ Test<Thread> t2 = new Test<Thread>();

✗ Test<Integer> t3 = new Test<Integer>();

CE: Type parameter j.l.Integer is not in its bound

→ We can define bounded types in combination also.

Ex: class Test<T extends Number & Runnable>
{
≡
}

As the type parameter we can pass any type which should be child class of Number & implement Runnable interface.

Ex: class Test<T extends Runnable & Comparable> ✓
{
≡
}

<T extends Number & Runnable & Comparable> ✓

<T extends Runnable & Number> ✗ [we have to take class first followed by interface].

<T extends Number & Thread> ✗ [we can't extend multiple classes simultaneously].

Note:—① We can define bounded types only by using extends keyword and we can't use implements and super keywords. But implements keyword purpose we can replace with extends keyword.

Ex: class Test<T extends Number>
{
}
✓

class Test<T implements Runnable>
{
}
✗

class Test<T extends Runnable>
{
}
✓

class Test<T super String>
{
}
✗

② As the type parameter instead of T we can use any valid Java identifier, but it is convention to use T always.

Ex: class Test<T>
{
}
✓

class Test<X>
{
}
✓

class Test<ABC>
{
}
✓

③ Based on our requirement we can use any no. of type parameters & need not be one.

Ex: class Test<X,Y>
{
}
✓

class Test<A,B,C>
{
}
✓

class HashMap<K, V>
{
}
K → Key Type
V → Value Type

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

4) Generic Methods & wild-card character(?):—

1. m1(AL<String l):—

i) We can call this method by passing ArrayList of only String type.

ii) But within the method we can add only String type of objects &

null to the list.

→ If we are trying to add any other type we will get CE.

Ex: m1(AL<String> l)
{
l.add("A"); ✓
l.add(null); ✓
l.add(10); X
}

2. m1(AL<?> l):—

i) We can call this method by passing AL of any type.

ii) But within the method we can't add anything to the list except null becoz we don't know the type of l exactly.

ex: m1(AL<?> l)
{
l.add("A"); X
l.add(10); X
l.add(null); ✓
}

↓
null is allowed becoz it is valid value for any type

→ This type of methods are best suitable for read operations.

3. m1(AL<? extends X> l):—

→ X can be either class or interface.

→ If X is a class then this method is applicable for AL of either X type or its child classes.

→ If X is an interface then this method is applicable for AL of either X type or its implementation classes.

→ Within the method we can't add anything to the list except null, becoz we don't know the type of AL exactly.

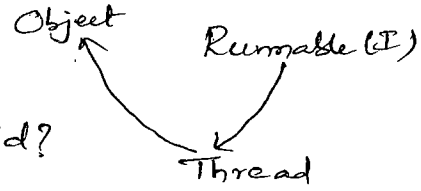
4. m1(AL<? super X> l):—

→ X can be either class or interface.

→ If X is a class then this method is applicable for AL of either X type or its super classes.

→ If X is an interface then this method is applicable for AL of either X type or super classes of implementation class of X.

→ But within the method we can add X type of objects & null to the list.



Q: Which of the following declarations are valid?

① `AL<String> l=new AL<String>();`

② `AL<?> l=new AL<String>();`

③ `AL<?> l=new AL<Integer>();`

④ `AL<? extends Number> l=new AL<Integer>();`

⑤ `AL<? extends Number> l=new AL<String>();` → CE: incompatible types found: AL<String> required: AL<? extends Number>

⑥ `AL<? extends Runnable> l=new AL<Thread>();`

⑦ `AL<? super String> l=new AL<Object>();`

⑧ `AL<?> l=new AL<?>();` → CE: unexpected type found: ? required: class or interface without bounds

⑨ `AL<?> l=new AL<? extends Number>();` → CE: unexpected type found: ? extends Number required: class or interface without bounds

→ We can declare type parameter either at class level or at method level.

Declaring Type parameter at class level:—

Ex:
```
class Test<T>
{
    we can use 'T' anywhere
    within this class based on
    our requirement.
}
```

Declaring Type parameter at method level:—

Ex:
```
class Test
{
    public <T>void m1(T t)
    {
        we can use 'T' anywhere
        within this method based on
        our requirement.
    }
}
```

✓ <T extends Number>
✓ <T extends Runnable>
✓ <T extends Number & Runnable>
✓ <T extends Runnable & Comparable>
✓ <T extends Number & Runnable & Comparable>
✗ <T extends Runnable & Number>
✗ <T extends Number & Thread>

5. Communication with non-Generic code:—

→ If we send Generic object to non-Generic area then Generic properties will be lost i.e., it starts behaving like non-Generic object.

→ Similarly if we are sending non-Generic object to Generic area then it will start behaving like Generic object.

Ex:
```
class Test
{
    psvm(-)
    {
        AL<String> l=new AL<String>();
        l.add("A");
        //l.add(10); → CE
        m1(l);
        S.o.p(l); ⇒ o/p: [A, 10, 10.5, true]
        //l.add(10); → CE
    }
    psvm1(AL l)
    {
        l.add(10);
        l.add(10.5);
        l.add(true);
    }
```
Generic Area

non-Generic Area

6) Conclusions :—

→ The main purpose of Generics is to provide type safety & to resolve type casting problems, but type safety & type casting applicable only at compile time.

→ Hence Generics concept is also applicable only at compile time & at runtime there is no such type of concept. i.e., while compiling generic syntax will be removed.

① Compile with Generic syntax
② Remove Generic syntax
③ Validate code once again
} By compiler

Ex①: The following declarations are equal.

```
AL l=new AL<String>();
AL l=new AL<Integer>();
AL l=new AL();
```
} equal

Ex:
```
AL l=new AL<String>();
l.add(10);
l.add(10.5);
l.add(true);
S.o.p(l) ⇒ o/p: [10, 10.5, true]
```

→ The following declarations are equal.

```
AL<String> l = new AL<String>();   } equal
AL<String> l = new AL();
```

→ For the above list objects we can add only String type of objects.

Ex:
```
class Test
{
    P S V m1(AL<String> l) => m1(AL l)
    {
    }
    P S V m1(AL<Integer> l) => m1(AL l)
    {
    }
}
```

CE: name clash: both methods having same erasure