01/05/11

Generics (1.5v)

1) Introduction
2) Generic classes
3) Bounded types
4) Generic methods
5) Wild Card Character
6) Communication with non-Generic Code.
7) Conclusions.

Introduction :-

→ Arrays are always Safe w.r.t type.

→ for Example, if our programme requirement is to add only String objects then we can go for String[] array. for this array we can add only String type of objects, by mistake if we are taying to add any other type we will get Compiletime-Error.

Ex :- String[] s = new String[600];

s[0] = "dorga"; ✓

s[1] = "pavan"; ✓

s[2] = new Student[]; ✗

Type-Safe.

C.E :- incompatable types
found : Student
required : String.

→ Hence In the Case of Arrays we can always give the guarantee about the the Type of elements. String[] array Contains only String objects, (i.e Strings) due to this arrays are always Safe to use w.r.t type.

→ But Collections are not Safe to use w.r.t type. for Example if our programme requirement 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 Compiletime-Error But program may fail at Runtime.

Eg:

```
ArrayList l = new ArrayList();
l.add("durga");
l.add("Sainu");
l.add(new Student());
:
String name1 = (String)l.get(0);
String name2 = (String)l.get(1);
String name3 = (String)l.get(2);
```

R.E:. ClassCastException.

→ There is no guarantee that Collection can Hold a particular type of objects. Hence w.r.t type Collections are not Safe to use.

Case2 :-

→ In the Case of Arrays at the time of retrieval it is not required to perform any TypeCasting.

Ex:-

```
String[] s = new String[600];
s[0] = "durga";
:
String name1 = s[0];
```

TypeCasting is not required.

→ But in the Case of Collections at the time of retrival compulsary we should perform TypeCasting otherwise we will get CompileTimeError.

Ex:-

```
ArrayList l = new ArrayList();
l.add("durga");
:
String name1 = l.get(0);
```

C.E:- Incompatable types
found : object
required : String

But

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

→ Hence, in the Case of Collections TypeCasting is mandatory which is a bigger headache to the programmer.

→ To overcome the above problems of Collections (TypeSafe & TypeCasting) Sun people introduced Generics Concepts in 1.5Version. Hence the main objectives of Generic Concepts are,

Hence the main objectives of Generics Concepts are,

1) To provide Type safety to the Collections. So that they can hold only a particular Type of objects.

2) To resolve Type Casting problems.

→ for Example — to Hold only String Type of objects a Generic version of ArrayList we can declare as follows.

parameter-type.

Basetype → `ArrayList<String> l = new ArrayList<String>();`

→ 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 Compiletime Error. i.e, we are geting Type-Safety.

```
l.add("durga");  ✓
l.add("Sunny");  ✓
l.add("10");     ✓
l.add(10);       ✗  C.E:- Cannot find Symbol
                        Symbol : method add(int)
                        location : class ArrayList<String>
```

→ At the time of retrival it is not required to perform any Type Casting.

```
String name1 = l.get(0);  ✓
```

↙

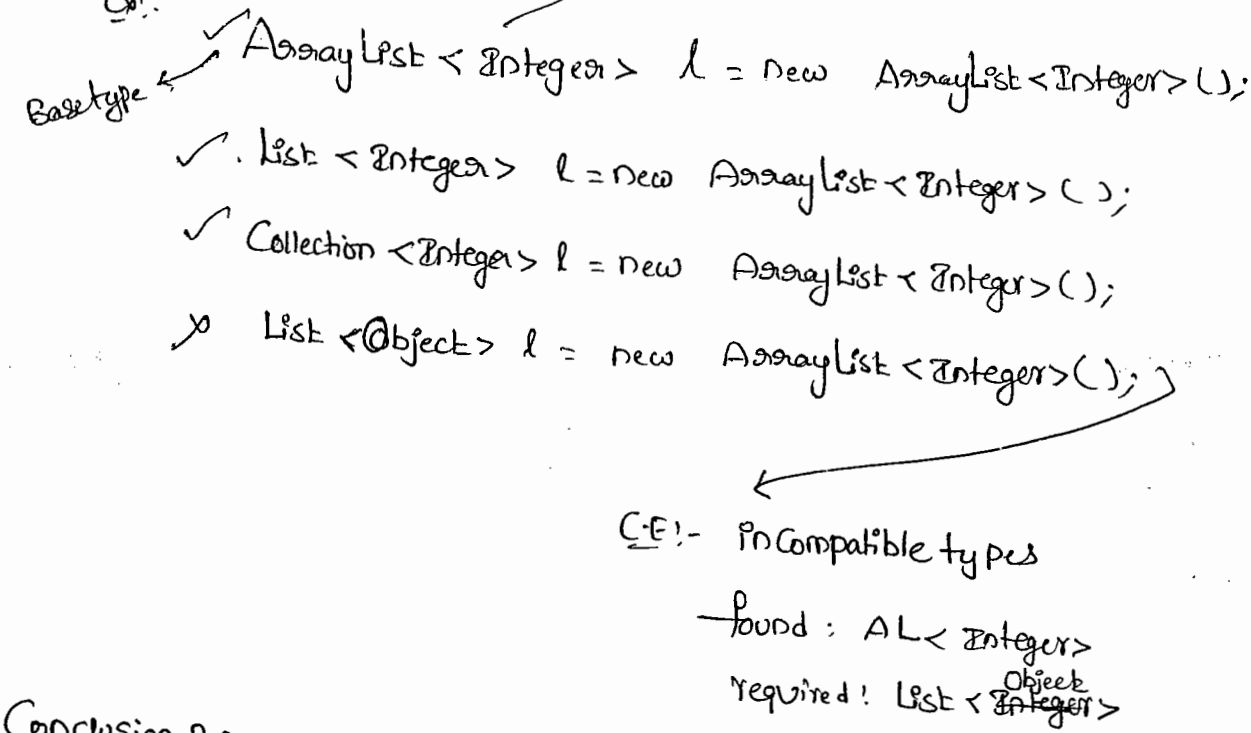
Type Casting is not required

Conclusion 1 :-

→ Usage of parent class reference to hold child class objects is considered as polymorphism.

→ polymorphism concept is applicable only for base type. but not for parameter type.

Ex:-



Conclusion 2 :-

→ for the parameter-type we can use any class or interface name & we can't use primitive type. vilation leads to CompileTime Error.

Ex:- ArrayList<int> l = new ArrayList<int>();

C.E:-
unexpected type
found : int
required : reference

C.E:-
Unexpected type
found : int
required : reference

Generic - classes :-

→ Until 1.4V a non-Generic Version of ArrayList class is declared as follows.

```
class ArrayList
{
    add(Object o);
    Object get(int index)
}
```

→ The argument to the add() method is Object. Hence we can add any type of object due to this we are not getting Type-Safety.

→ The return type of get() method is Object, Hence At the time of retrieval Compulsary we should perform Type Casting.

→ But in 1.5V a Generic Version of ArrayList class is declared as follows.

```
class ArrayList<T>        // T → Type parameter.
{
    add<T t>
    T get(int index)
}
```

→ Based on our runtime requirement Type parameter 'T' will be replaced with Corresponding provided Type.

→ for Example, To hold only String type of object we have to Create Generic Version of ArrayList Object as follows.

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

→ for this requirement the corresponding loaded version of ArrayList Class is,

```
Class ArrayList<String>
{
    add(String s)
    String get(int index)
}
```

→ add() method can take String as the argument hence we can add only String type of objects. By mistake if we are trying to add any other type we will get CompileTimeError. i.e, we are getting Type-Safety.

→ The return type of get() method is String, Hence at the time of retrival we can assign directly to the String type variable it is not required to perform any TypeCasting.

Note :-

1) As the Type parameter we can use any valid java identifiers but it is Convension to use "T".

Eg:- ✓ Class AL<X>
{
}

Class AL<Durga>
{
}

2) We can pass any no. of type parameters but it need not be one. class

Ex:-

```
class HashMap<K,V>
{
}
```

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

Key type

Value type

→ Through Generics we are associating a type-Parameter to the classes. Such type of parameterized classes are called Generic-classes.

→ We can define our own Generic classes also.

Ex:-

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

```
Class GenDemo
{
  p.s.v.m(String[] args)
  {
    Gen<String> g1 = new Gen<String>("durga");
    g1.show();  // the type of ob is : java.lang.String
    S.o.pln(g1.getOb());  durga

    Gen<Integer> g2 = new Gen<Integer>(10);
    g1.show();  // the type of ob is : java.lang.Integer.
    S.o.pln(g2.getOb());  10
  }
}
```

① ②

Bounded Types :-

→ We can Bound the type parameter for a particular range By using extends keyword.

ex(1) :-

```
Class Test<T>
{
}
```

→ As the type parameter we can pass any type hence it is UnBounded type.

✓ `Test<String> t1 = new Test<String>();`

✓ `Test<Integer> t2 = new Test<Integer>();`

Eg:- Class Test<T extends Number>
{
}

→ As the type parameter we can pass either Number type or its child classes. it is bounded type.

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

✗ Test<String> t2 = new Test<String>();

C.E:- Type parameter java.lang.String is not with in its bound

→ We can't Bound Type Parameter By using implements & Super Keywords :

Ex:- ① Class Test<T implements Runnable>
{
}

② Class Test<T Super Integer>
{
}

But,

→ implements Keyword purpose we can Survive by using extends Keyword only

Ex:- Class Test<T extends X>
{
}
↳ class / interface.

→ X → Can be either class/interface.

→ if X is a class then as the Type parameter we Can provide either X type or it's child classes.

→ if X is an interface as the Type parameter we Can provide either X type or it's implementation classes.

ex!-

```
Class Test<T extends Runnable>
{
}
```

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

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

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

C.E!- Type parameter java.lang.String is not within its Bound

→ We Can Bound the Type parameter even in Combination also.

ex!-

Class Test<T extends Number & Runnable>

→ As the Type parameters we Can pass any type which is the child Class of Number & implements Runnable interface.

ex!-

✓ ① Class Test<T extends Runnable & Comparable)

✓ ② class Test<T extends Number & Runnable & Comparable)

✗ ③ Class Test<T extends Number & Thread>

→ We Can't extend more than one class at a time.

✗ ⑤ Class Test<T extends Runnable & Number>

→ we have to take first class & then interface.

Generic Methods & Wild Card Character ?

→ ① m1(ArrayList<String> l) ✓

→ this method is applicable for ArrayList<String> (ArrayList of only 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 Compiletime Error.

eg:- m1(ArrayList<String> l)
{
l.add("A"); ✓
l.add(null); ✓
l.add(10); ✗

② m1(ArrayList<? extends X> l) ✓

→ we can call this method by passing ArrayList of any type. But within the method we can't add any-type except null to the list. Because we don't know the type Exactly.

③ Eg:-
m1(ArrayList<?> l)
{
l.add(null); ✓
l.add("A"); ✗
l.add(10); ✗
}

3) m1(AL<? extends X> l) ✓

→ 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 also we can't add any type of elements to the list Except null

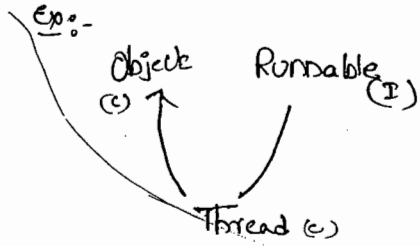
4) m1(ArrayList<? super X> l) ✓

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

→ If X is an interface then this method is applicable for ArrayList of either X type or Super classes of implementation class of X

→ With in the method we can add only X type objects & null to the List

Ex:-



Q) Which of the following declarations are valid?

① AL<String> l = new AL<String>(); ✓

② AL<?> l = new AL<String>(); ✓

③ AL<E, extends String> l = new AL<String>(); ✓

④ AL<? super String> l = new AL<String>(); ✓

⑤ AL<? extends Object> l = new AL<String>(); ✓

✓ ⑥ AL < ? extends Number > l = new AL < Integer > ();

✗ ⑦ AL < ? extends Number > l = new AL < String > ();

C.E:- Incompatible types

found : AL<String>

required: AL < ? extends Number >

✗ ⑧ AL < ? > l = new AL < ? extends Number > ();

✗ ⑨ AL < ? > l = new AL < ? > ();

C.E:- unexpected type

found : ?

required : Class or interface without bounds.

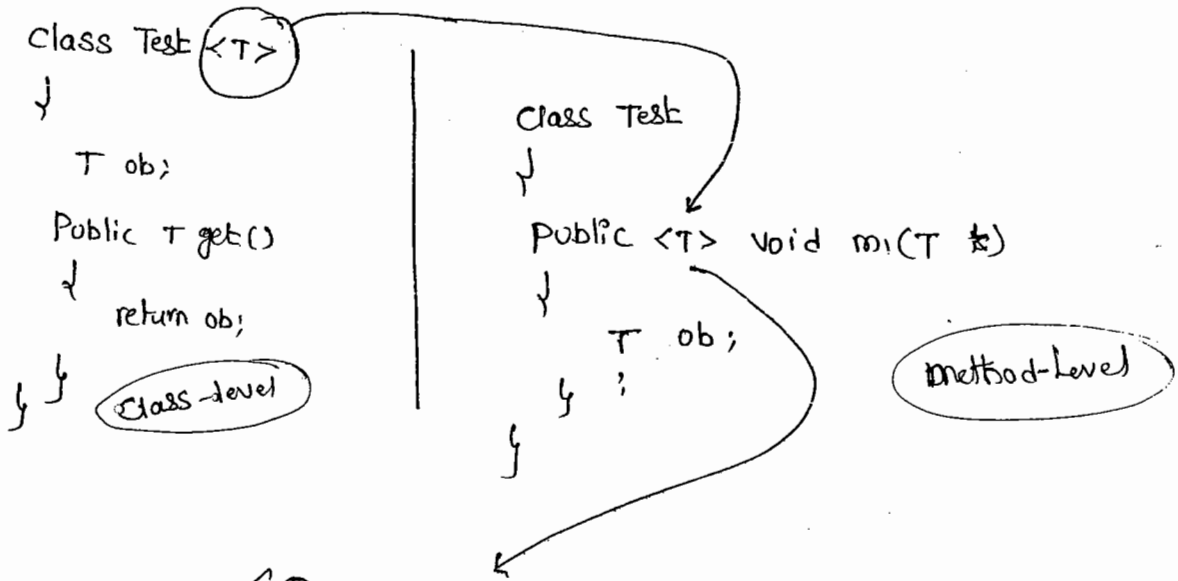
→ We can define the type parameter either at Class-Level or at method-Level.

Declaring type parameter at class level:-

```
Class Test <T>
{
    T ob;
    Public T get()
    {
        return ob;
    }
}
```

Declaring Type parameter at method-Level:-

→ we have to declare the type parameter just before returntype.



✓ ① <T extends Number>
✓ ② <T u Runnable>
✓ ③ <T u Number & Runnable>
✓ ④ <T extends Runnable & Comparable>
✗ ⑤ <T extends Number & Thread>
✗ ⑥ <T extends Runnable & Thread>

Communication with non-Generic Code :-

→ To provide Compatability with old version SUN people compromized the Concept of Generics in very few areas. the following is one such area.

```
eg:-    Class Test
        {
          P.S.V.m(String[] args)
          {
            AL<String> l = new AL<String>();
            l.add("A");
```

Eg:-

```
class Test
{
  p s v m (---)
  {
    AL<String> l = new AL<String>();
    l.add("A");
    l.add(10);  C.E
    m1(l);
    S.o.pln(l);   [A, 10, 10.5, true]
    // l.add(10);  C.E
  }
  public static void m1(AL l)
  {
    l.add(10);
    l.add(10.5);
    l.add(true);
  }
}
```

Generic area

non-Generic area

Conclusions :-

④ Generics Concepts is applicable only at Compiletime to provide type Safety & to resolve type casting problems. At Runtime there is no Such type of Concept. Hence the following declarations are equal,

Eg:-

all are equal

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

ex:- ArrayList l = new ArrayList<String>();

```
l.add("A");
l.add(10);
l.add(true);
S.o.pln(l);  [A, 10, true]
```

→ The following two declarations are equal & there is no difference

both are equal

1) AL<String> l = new AL<String>();
2) AL<String> l = new AL();