Generic (1)

* The main objectives of generics are to provide type safty & to resolve type casting problems.

Case 1:- Type Safty:-

Arrays are Type safe that is we can give the gaurenty for the type of elements presents inside arrays.

for example if our programing requirments is only holds the string type of objects we can choose string array by mistake if we are trying to add any other type of object we will get compile type Error.

```
String[] s = new String[1000];
s[0] = "durga";
s[1] = "Ravi";
* s[2] = new Integer(10);   CE: Incompatible type
                            found: java.lang.Integer
                            required: java.lang.String
s[2] = "shiva";
```

hence string array can contain only string type of objects. due to this we can give the gaurenty for the type of Elements presents inside array hence arrays are safe to use with respects to type that is Arrays are type safe.

* But Collections are not Type Safe that is we can't give the gaurenty for the type of Element present inside collection.

for example:- if our programming requirement is to hold only string type of object and we choose arraylist, by mistake if we are trying to add any other type of object, we will not get any compile time error. but program may fail at run time.

```
ArrayList l = new ArrayList();
l.add("durga");
l.add("Ravi");
l.add(new Integer(10));
```

Retriving object from Array List.

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

hence we can not gaurenty for the type of elements presents inside collection due to this collections are not safe to use with respect to type. that is collections are not TYPE safe.

Type Casting:- (Case:2)

On the case of Arrays at the time of retrival it is not required to perform type casting because there is a guarenty for the Elements presents inside array.

```
String [] s = new String[10000];
s[0] = "durga";
```

at the time of retrival

```
String name1 = s[0];
```

(Type Casting not required)

* but in the case of Collections at the time of retrival compulsory we should perform type casting because there is no guaranty for the elements presents inside collections.

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

at the time of retrival:-

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

CE : Incompatible type
found : java.lang.Object
required : java.lang.String

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

(Type-casting is mandatory)

hence the Type-casting is bigger headache problem in Collections.

To overcome these problem of collections SUN people introduced generics concepts in 1.5 version.

hence the main objective of Generics are;

1. To provide Type-safty
2. To resolve Type-casting problem.

for example to add only String type of objects we can create generic version of ArrayList object as follows:-

```
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 then we will get compile time error.

```
l.add("durga");
l.add("Ravi");
// l.add(new Integer(10)); ⟶ CE X
l.add("Shiva");
```

hence through generics we are getting Type safty.

at the time of retrival we are not required to perform type casting: 3

```
ArrayList<String> l = new ArrayList<String>();
l.add("durga");
:
```

While retrival:- `String name1 = l.get(0);`

Type-casting is not required

Through Generic we can solve type casting problem.

Difference Between normal AL version & Generic AL version.

AL l = new AL();	AL<String> l = new AL<String>();
• It is a non generic version of AL object	• It is a generic version of AL object
• For this AL we can add any type of of object & hence it is not Type-safe	• For this AL we can add only String type of objects & hence it is Type-Safe
• At the time of retrival Compulsory we have to perform Type-casting	• At the time of retrival 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 Parrents reference to hold child Object is the Concept of Polymorphism

Base Type ← AL<String> l = new AL<String>(); ✓ (→ Parameter Type)

List<String> l = new AL<String>(); ✓

Collection<String> l = new AL<String>(); ✓

AL<Object> l = new AL<String>(); X

CE : incompatible type
found : AL<String>
required : AL<object>

Conclusion - 2:- For the type Parameter we can provide any class or interface name but not Primitive. If we are trying to provide primitive then we will get CE.

Example:- AL<int> l = new AL<int>();

CE: Unexpected Type
found : int
required : reference.

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

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

the argument to add method is Object and hence we can add any type of object to the ArrayList. due to this we are missing type safty

the return type of get method is Object hence at the time of retrival we have to perform Type-casting.

But in 1.5 version generic version of ArrayList class is declared as follows:

```
class AL<T>        // T → Type Parameter
{
    add(T t);
    T get(int index);
}
```

based on our runtime requirement T will be replaced with our provided type.

for Example to hold only String type of objects a generic version of ArrayList object can be created as follows:

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

for this requirement compiler considered version of ArrayList class is as follows:-

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

The argument to add method is String type hence 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.

```
l.add("durga");
l.add(new Integer(10));
```

CE: Cannot find Symbol
Symbol : method add(I.d.Integer)
location: class AL<String>

Hence through generic we are getting Type safty.
The return type of get method is String and hence at the time of retrival we are not required to perform type casting.

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

Type casting is not required

* In generic we are associating a Type Parameter to the class Such type of Parameterized classes are nothing but Generic classes or Template classes.

* Based on our requirement we can define our own Generic classes also.

```
Class Account<T>
{
    :
}

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

Example:-

```
Class Gen<T>
{
   T ob;
  Gen (T ob)
  {
  this.ob = ob;
  }
  Public Void show ( )
  {
     System.out.println("The type of ob:"+
                              ob.get(class().getName());
     }
   Public T getOb()
   {
    Return ob;
  }
}
Class GenDemo
{
  P s v main (String[] args)
  {
  Gen<String> g1 = new Gen<String>("durga");
       g1.show(); // The type of ob; Java.Lang.String
       sopln( g1.getOb()); // durga.

  Gen<Integer> g2 = new Gen<Integer>(10);
    g2.show();//The type of ob: Java.Lang.Integer
    Sopln (g2.getOb()); // 10

  Gen<Double> g3 = new Gen<Double>(10.5);
    g3.show(); // The type of ob; Java.lang.Double
    Sopln( g3.getOb()); 10.5

  }
}
```

Bounded Types :-

we can bound the Type parameter for a particular range by using extends Keyword. Such types are called Bounded Types.

```
class Test<T>
{
  :
}
```

as the type parameter we can pass any Type and there are no restrictions and it is unbounded type.

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

Syntax for Bounded Types :-

```
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 a interface then as the type parameter we can pass either X type or its implementation classes.

```
class Test<T extends Number>
{

}
```

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

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

```
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 java.lang.Integer is not within its bound

* We can define bounded type even in combination also

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

as the Type Parameter we can take any thing which should be child class of Number and should implements Runnable Interface

✓ Class Test<T extends Runnable & Comparable>

✓ Class Test<T extends Number & Runnable & Comparable>

✗ Class Test<T extends Runnable & Number> [Because we have to take class first followed by interface next] ①

✗ Class Test<T extends Number & Thread> [Because we can't extends more than one classes simultaneously.]

Note:- ① We can define bounded type only by using extends keyword and we can't use implement & super keywords but we can replace implements keyword purpose with extends keyword.

```
Class Test<T extends Number>
{
}
```
✓

```
Class Test<T implements Runable>
{
}
```
✗ ②

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

```
Class Test<T super String>
{
}
```
✗

② as the Type Parameter 'T' we can Take any valid java Identifier but it is convention to use 'T'.

```
Class Test<T>
{
}
```
✓

```
Class Test<X>
{
}
```
✓

```
Class Test<A>
{
}
```
✓

```
Class Test<durga>
{
}
```
✓

③ Depedon our requirement we can declare any number of Type Parameters and all these type Parameter should seprated with ','.

```
Class Test<A,B>
{
}
```
✓

```
Class Test<X,Y,Z>
{
}
```
✓

```
class HashMap<K, V>
{
}
```
K: Key Type, V: Value Type ✓

```
HashMap<Integer, String> h =
        HashMap<Integer, String>();
```

Generic Method & Wild-card character (?)

① m1(AL<String> l) :-

We can call this method by Passing ArrayList of only String type. but with in the method we can add only String type of Object to the List by mistake if we are trying to add any other type then we get Compile time error.

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

② m1(AL<?> l) :-

We can call this method by Passing ArrayList of any unknown (Any) type. But with in the method we can't add anything to the List except null, Because we don't know the type exactly. null is allowed because it is valid value for any type.

```
m1(AL<?> l)
{
    l.add(10.5);     X
    l.add("A");      X
    l.add(10);       X
    l.add(null);     ✓
}
```

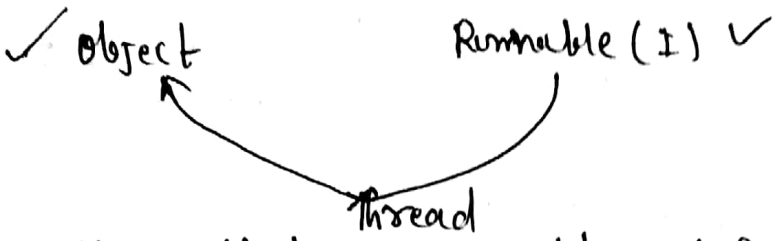
This type of methods are best suitable for read only operation.

③ m1(AL<? extends X> l) :-

X can be either Class or Interface. 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 classes. But with in the method we can't add anything to the List except null. Because we don't Know Type Exactly. This method also best Suitable for Read only operation.

④ m1(AL<? super X> l) ;- X can be either Class or Interface. If X is a Class then we can Call this method

By Passing ArrayList of either X type OR its Super Class.
If X is an interface then we can call this method by Passing ArrayList of either X type OR Super Class of Implementation Class of X.



But with in the method we can add X type of object and 'null' to the List.

Example:-

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

CE: Incompatible types
found: AL<String>
required: AL<? extends Number>

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

CE: unexpected type
found: ?
required: class or interface without bounds

✗ 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 Method level.
- Declare Type Parameter at class level.

```
class Test <T>
{
  we can use 'T' with in this class
  Based on our requirement.
}
```

- Declaring Type Parameter at Method level.

We have to declare Type Parameter Just before return Type.

```
class Test
{
  Public <T>Void m1(T ob)
  {
    we can use 'T' any where with in this
    Method based on our requirement.
  }
}
```

We can define Bounded Type Even at Method level also -

✓ Public <T> void m1();
✓ Public <T etends Number> m1();
✓ Public <T extends Runnable> m1();
✓ Public <T extends Number & Runnable> m1();
✓ Public <T extends Comparable & Runnable> m1();
✓ Public <T extends Number & Comparator & Runnable> m1();
✗ Public <T extends Runnable & Number> m1();
[First we have to take class and then interface]

✗ Public <T extends Number & Thread> m1();
[we can't extend more than one class]

Communication with non Generic Code:

If we send Generic object to non Generic area then its start behaving like non Generic object.
Similarly if we send non Generic object to Generic Area then its start behaving like Generic Object that is the location in which object present Based on that Behaviour will be defined.

```
Class Test
{
  Public Static void main(String[] args)
  {
     ArrayList <String> l = new ArrayList <String>();
      l.add("durgei");
      l.add("Ravi");
      l.add(10); // CE
        m1(l);
      sopln(l); // [durgei, Ravi, 10, 10.5, true]
     } l.add(10.5), → CE

    Public S V m1(ArrayList l)
    {
       l.add(10);
       l.add(10.5);
       l.add(true);
    }
  }
```

} Non generic Area

Conclusion;- The main purpose of Generics is To Provide Type Safty and To resolve Type casting Problem. Type Safty & Type Casting both are applica

at compile time hence Generic concept also applicable only at compile time but not at Run time,
At the time of compilation as last step Generic Syntax will be remove and hence for the JVM Generic Syntax will not be available.
hence the following declaration are equals:-

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

```
AL l = new AL<String>();
l.add(10);
l.add(10.5);
l.add(true);
sopln(l);   [10, 10.5, true
```

The following declarations are equals

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

for these ArrayList object we can only String type of object.

- class Test
{
  Public void m1(AL<String> l) ⇒ m1(AL l) {
  }
  Public void m1(AL<Integer> l) ⇒ m1(AL l);
  {
  }

}

CE: name clash: Both methods have same Erasure.

Le Concept

(1) Compile code Normally by considering generic Syntax.

(2) Remove Generic syntax;

(3) Compile once again resultant code.