Generics

1. Introduction
2. Generic classes
3. Bounded Types
4. Generics, methods & Wild-card character (?)
5. Communication with Non-Generic code
6. Conclusions

Introduction

→ The main objectives of generics are to provide Type-Safety & to ~~provide~~ ~~Type~~ Resolve Type-casting problems.

→ **Case-I : [Type-Safety]**

- Arrays are Type-safe i.e we can give the guarantee for the type of elements present inside Array, for example if our programming requirement is to hold only string type of object, we can choose String[]. By mistake, if we are trying to add any other type of objects we will get C.E.

```
String[] s = new String[10...0];
s[0] = "durga";          ✓
s[1] = "ravi";           ✓
s[2] = new Integer(10);  ✗
```

C.E. Incompatible Types
found : j.l.Integer
required : j.l.String

```
s[2] = "shiva";          ✓
```

- Hence, String[] can contain only string type of objects, due to this we can give the guarantee for the type of elements present inside Array. Hence Arrays are safe to use w.r.t type. i.e arrays are type-safe.

- But Collections are not type-safe i.e we can't give the guarantee for the type of elements present inside collection.

Ex. If our programming req. is to hold only string type of objects, and it choose ArrayList, By mistake if we are trying to add any other type of object

we won't get any compile time error, But the program may fail at runtime

```
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);
```

→ R.E: ClassCastException

- Hence, we can't give the gaurantee for the type of elements present inside collection, Due to this collections are not safe to use w.r.t type i.e collections are not Type-safe.

→ Case-II [Type-Casting]

- In the case of arrays, at the time of retrieval, it is not required to perform Type-casting, bcz there is a gaurantee for the type of elements present inside array

```
String[] s = new String[10000];
s[0] = "durga";
;
String name1 = s[0];
```

↓

Type-casting Not Required

- But in the case of collections, at the time of retrieval, compulsory we should perform Type-casting bcz there is no gaurantee for the type of elements present inside collection.

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

↓

C.E: Incompatible Types
found: java.lang.Object
required: java.lang.String

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

↓

Type casting Mandatory

- Hence Type-casting is a bigger headache in collection.

→ To overcome above problems of collection SUN people introduced "Generics" concept in 1.5V

→ Hence the main objectives of "Generics" are to provide Type-safety & To resolve Type-casting problem.

→ Ex: To hold only String type of object we can create Generic version of Arraylist object as follows:

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

For this Al we can add only String type of object, By mistake if we are trying to add any other type then we will get C.E.


```
l.add("durga");   ✓
l.add("ravi");    ✓
l.add(new Integer(10)); ✗ C.E.
l.add("shiva");   ✓
```

- Hence, through generics we are getting Type-Safety.

- At the time of retrival, we are not required to perform Type-casting

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

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

Type-casting not Required

Bez compiler knows that Arraylist can contain only String values.

- Hence, through generics we can solve Type-casting problem.

AL l = new AL();	AL<String> l = new AL<String>();
(1) Non-generic version of AL object	Generic version of AL object
(2) Not Type-safe	Type-safe
(3) Type-casting Required	Type-casting not required

→ Conclusion 1 :-

- polymorphism concept applicable only for the Base-Type but not for parameter-Type.

[what is polymorphism ?
usage of parent Reference to hold child object]

Ex.

parameter Type

Base Type

AL<String> l = new AL<String>(); ✓

List<String> l = new AL<String>(); ✓

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

AL<Object> l = new AL<String>(); ✗

C.E. incompatible type
found : AL<String>
required : AL<Object>

→ Conclusion 2 :-

- For the "parameter Type" we can provide any class or interface name but not primitives

- If we are trying to provide then we will get C.E.

Ex.

AL<int> al = new AL<int>();

C.E. Unexpected Type
found : int
required : reference

Generic Classes

→ Untill 1.4V, a non-generic version of Arraylist class is declared as follows:

1.4V

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

→ The argument to add() method is Object & hence we can add any type of object to the Arraylist, due to this we are missing Type-Safety.

→ The return type of get() is Object, at the time of retrieval we have to perform Type-casting.

→ But in 1.5V, a generic version of Arraylist class is declared as follows:

1.5V

Type-parameter

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

→ Based on our run-time requirement T will be replaced with our provided type.

→ For Example, to hold only string type of object a generic version of Arraylist object can be created as follows:

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

→ For this requirement, compiler consider Arraylist class is as follows:

```
class Arraylist<String>
{
    add(String s)
    String get(int index)
}
```

→ The argument to add() method is String type, hence we can add any String type of Object. By mistake if we are trying to add any other type we will get C.E.

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

C.E- cannot find symbol
Symbol : method add(j.l.Integer)
location : class Arraylist<String>

→ Hence, through generics we are getting Type-safety.

→ Return type of get() method is String & hence at a time of retrieval we are not required to perform Type-casting.

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

Type-casting Not Required

→ In generics, we are associating a Type-parameter to the class such type of parameterized class 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>();
```

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 main(Str)
    {
        Gen<String> g1 = new
            Gen<String>("durga");
        g1.show();
        s.o.p(g1.getOb());
```

//The type of ob: J.L.String

// durga

```
        Gen<Integer> g2 = new
            Gen<Integer>(10);
        g2.show(); //The Type of ob: J.L.Integer
        s.o.p(g1.getOb()); //10

        Gen<Double> g3 = new
            Gen<Double>(10.5);
        g3.show(); //The type of ob: J.L.Double
        s.o.p(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 a 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>
{
}
```

- where X can be either class / Interface.

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

- If X is an interface then as a 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>();

→
```
class Test<T extends Runnable>
{
   ≡
}
```

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

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

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

→ C.E. type parameter i.e. Integer is not within its bound

C.E. type parameter i.e. String is not within its bound

→ we can define Bounded Types even in combination also

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

- As a Type-parameter we can take anything which should be child class of Number & should implements Runnable interface.

→ class Test<T extends Runnable & Comparable> ✓

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

class Test<T extends Runnable & Number> ✗

Bcz we have to take class first followed by interface next

class Test<T extends Number & Thread> ✗

Bcz we can't extends more than one class simultaneously

Note :- ① we can define Bounded Types only by using "extends" keyword & we can't use "super" & "implements" keyword but we can replace "implements" keyword purpose 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 a Type-parameter(T) we can take any valid java identifier but it is convention to use (T)

class Test<T> ✓
{
}

class Test<durga>
{
}

③ Based on our requirement we can declare any No. of Type-parameters & all these Type-parameters should be separated by comma (,).

```
class Test<A,B>
{
}
```
✓

```
class Test<X,Y,Z>
{
}
```
✓

```
class HashMap<K,V>
{
}
```
(K → Key Type, V → Value Type) ✓

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

✱ Generic Methods & Wild card character (?)

① `m(ArrayList<String> l)`

- we can call this method by passing ArrayList of only "String" type

- But within a method we can add only "String" type of objects into the list, By mistake if we are trying to add any other type then we will get C.E.

```
m(AL<String> l)
{
    l.add("A");   ✓
    l.add(null);  ✓
    l.add(10);    ✗ C.E.
}
```

② `m(ArrayList<?> l)`

- we can call this method by passing ArrayList of any unknown type

- But within a method we can't add anything to the AL except null, bcz we don't know the type exactly.

- null is allowed bcz it is valid value for any type.

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

- This type of methods are best suitable for read-only operations.

(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 AL of either X type or its child classes. If X is an interface then we can call this method by passing AL of either X type or its implementation classes.

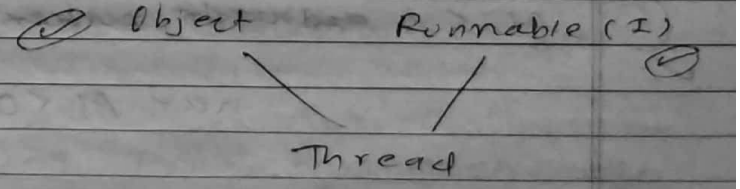
- But within a method, we can't add anything to the AL except null, bcz we don't know the

type exactly.

- This type of methods are also best suitable for read-only operations.

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



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

Ex.

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

(2) ✓ `ArrayList<?> l = new ArrayList<String>();`

(3) ✓ `ArrayList<?> l = new ArrayList<Integer>();`

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

(5) ✗ `ArrayList<? extends Number> l = new ArrayList<String>();`

C.E. Incompatible types
found : ArrayList<String>
required : ArrayList<? extends Number>

(6) ✓ `ArrayList<? super String> l = new ArrayList<Object>();`

(7) ✗ `ArrayList<?> l = new ArrayList<?>();`

C.E. Unexpected type
found : ?
required : class or interface without bounds

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

C.E. 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 :-

```
class Test<T>
{
    we can use "T" within 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"
        anywhere within this
        method based on
        our requirement
    }
}
```

→ we can define "Bounded types" even at method level also.

```
public <T> void m1()
```

✓ <T extends Number>
✓ <T extends Runnable>
✓ <T extends Number & Runnable>
✓ <T extends Comparable & Runnable>
✓ <T extends Number & (Comparable & Runnable)>
X <T extends Runnable & Number>

[∵ first we have to take class & then interface]

X <T extends Number & Thread>

[∵ we can't extend more than one class]

(*) Communication with Non-Generic code :-

→ If we send Generic object to Non-generic area then it starts behaving like Non-generic object

→ similarly, if we send Non-Generic object to generic area then it starts behaving like generic object

→ i.e the location in which object present, based on that behaviour will be defined

→ Ex:-

```
class Test
{
    p s v main(sa)
    {
        ArrayList<String> l =
            new ArrayList<String>();
        l.add("durga");
        l.add("ravi");
        // l.add(10); ⟶ C.E.
```

Generic Area

```
m1(l);
s.o.p(l);  [durga, ravi, 10, 10.5, true]

// l.add(10.5); → C.E.
}

p s v m(ArrayList l)
{
    l.add(10);
    l.add(10.5);
    l.add(true);
}
}
```

Non-Generic Area

Conclusions

~> The main purpose of generics is to provide Type-safety & To resolve Type-casting problems.

~> Type-safety & Type-casting both applicable at compile-Time, so Generic concept also applicable at Compile-Time not Run-Time.

- At the time of compilation, as a last step generic syntax will be removed & hence for the JVM, Generic syntax won't be available.

- Hence the following declarations are equal.

Equal Declarations

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

Ex:

```
AL l = new AL<String>();
l.add(10);
l.add(10.5);
l.add(true);
s.o.p(l);  [10, 10.5, true]
```

~> The following declarations are equal.

Equal declarations

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

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

- For this AL object we can add only string type of object.

~ class Test
{
    public void m(AL<String> l)
    {
        ≡
    }
    public void m(AL<Integer> l)
    {
        ≡
    }
}

m(AL l) ⇐ m(AL l)

∴ C.E: name clash: Both methods have same erasure

~ At compile time

(1) Compile code normally by considering generic syntax

(2) Remove Generic syntax

(3) Compile once again resultant code.

~ so, Adding generic syntax after new student<> is not worthy

End of Generics

Start of Assertions