**Generics – Part\_05**

* **Declaring Type-Parameters:**

We can declare type parameter either at class level or 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();

public <T extends Number> void m1();

<T extends Runnable>

<T extends Number & Runnable>

<T extends Comparable & Runnable>

<T extends Number & Comparable & Runnable>

<T extends Runnable & Number> // Invalid

[first class has to come and then interface]

<T extends Number & Thread> // Invalid

[ 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 starts behaving like generic object. That is the location in which object present based on that behavior will be defined.

Note: Recall about Srinivas Iyer story.

Example:

class Test{

public static void main(String[] args){

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

l.add(“durga”);

l.add(“ravi”); // Complete block area is generic-area

l.add(10); // CE

m1(l);

System.out.println(l); [durga, ravi, 10, 10.5, true]

l.add(10.5); // CE

}

public static void m1(ArrayList l){

l.add(10);

l.add(10.5); // This complete block is non-generic area.

l.add(true);

}

}

* **Conclusions:**

1. The main purpose of Generics is to provide type-safety and to resolve type-casting problems.
2. Type-safety and type-casting both are applicable at compile-time, hence Generics concept also applicable only at compile-time but not at runtime.
3. At the time of compilation as a last step, Generics syntax will be removed and hence for the JVM generics syntax won’t be available.

Example:

ArrayList l = new ArrayList<String>();

l.add(10);

l.add(10.5);

l.add(true);

System.out.println(l); [10, 10.5, true]

Note: Compile-time compiler will check the reference type and at the runtime JVM will check the objects.

1. Because of all the above statements the below declarations are equal.

ArrayList l = new ArrayList<String>();

ArrayList l = new ArrayList<Integer>();

ArrayList l = new ArrayList<Double>();

ArrayList l = new ArrayList();

1. The following declarations are equal.

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

ArrayList<String> l = new ArrayList();

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

1. Another example:

class Test{

public void m1(ArrayList<String> l){

}

public void m1(ArrayList<Integer> l){

}

}

CE: 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.