**Generics – Part\_02**

* **Generics:**

For example, to hold only String type of objects we can create Generic version of ArrayList 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.

Example:

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

l.add(“durga”); // valid

l.add(“ravi”); // valid

l.add(new Integer(10)); // Invalid – CE

l.add(“shiva”);

Hence, through Generics we are getting type safety.

At the time of retrieval, we are not required to perform type-casting.

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

l.add(“durga”);

String name1 = l.get(0); // type-casting is not required.

Hence, through Generics we can solve type-casting problem.

* **Non-Generic ArrayList vs Generic ArrayList.**

|  |  |  |
| --- | --- | --- |
| **S.No** | **AL l = new AL();** | **AL<String> l = new AL<String>** |
| 1 | It is a Non-generic version of ArrayList object. | It is a Generic version of ArrayList object. |
| 2 | For this ArrayList we can add any type of object and hence it is not type-safe. | For this ArrayList we can add only String type of object and hence its is type-safe. |
| 3 | At the time of retrieval compulsory, we have to perform type-casting. | At the time of retrieval, we are not required to perform type-casting. |

* **Conclusions:**

**Conclusion\_01:**

Polymorphism concept applicable only for the base type but not for parameter-type. (Usage of parent reference to hold child object is the concept of polymorphism).

Example:

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

ArrayList 🡪 Base Type

String 🡪 Parameter-type

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

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

ArrayList<Object> l = new ArrayList<String>(); // Invalid – CE

CE: incompatible types

found: ArrayList<String>

required: ArrayList<Object>

**Conclusion\_02:**

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 compile time error.

Example:

ArrayList<int> x = new ArrayList<in>();

CE: unexpected type

found:int

required: reference

* **Generic classes:**

Until 1.4 version, a non-generic version of ArrayList class is declared as follows.

class ArrayList{

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-safety.

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

But, in 1.5 version a generic version of ArrayList is declared as follows.

class ArrayList<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.

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

For this requirement, compiler considered version of 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 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”); // Valid

l.add(new Integer(10)); // Invalid – CE:

CE: cannot find symbol

symbol: method add(java.lang.Integer)

location: class ArrayList<String>

Hence, through Generics we are getting type-safety.

The return-type of get() method is String and hence, at the retrieval we are not required to perform type-casting.

String name1 = l.get(0); // type-casting is not required.

In generics, 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>();

* **Creating our own Generic class:**

class Gen<T>{

T ob;

Gen(T ob){

This.ob = ob;

}

public void show(){

System.out.println(“The type of ob:”+ob.getClass().getName());

}

public T getOb(){

return ob;

}

}

class GenDemo{

public static void main(String[] args){

Gen<String> g1 = new Gen<String>(“durga”);

g1.show();

System.out.println(g1.getOb());

Gen<Integer> g2 = new Gen<Integer>(10);

g2.show();

System.out.println(g2.getOb());

Gen<Double> g3 = new Gen<Double>(10.5);

g3.show();

System.out.println(g3.getOb());

}

}

Output:

The type of ob: java.lang.String

Durga

The type of ob: java.lang.Integer

10

The type of ob: java.lang.Double

10.5