JAVA GENERICS -Vikash Kumar

**1) Objectives of Generics:-**

a) To Provide Type Safety

b) To resolve Type Casting problems

**2)** **Arrays are Type Safe**, i.e. we can give guarantee for the type of elements present inside array.

**Collections are not type safe**.

String[] s = new String[1000];

s[0] = “durga”

s[1] = “aman”

s[2] = new Integer(10); //Gives CE (Compilation Error)

**Compilation Error**: Incompatible types. Found: java.lang.Integer, required: java.lang.String

ArrayList list = new ArrayList();

list.add(“Ram”);

list.add(“Navin”);

list.add(new Integer(10));

.

.

String name1 = (String) list.get(0);

String name2 = (String) list.get(1);

String name3 = (String) list.get(2); // Give RE (Runtime Error)

**Runtime Error**: ClassCastException

**Note**: It’s better to find error during compile time as compared to runtime. Since Collections give runtime error instead of compile time error, they are not safe to use w.r.t type.

**3) In case of Arrays, type casting is not required at the time of retrieval whereas in case of Collections, it is mandatory to type caste at the time of retrieval.**

String[] list = new String[100];

list[0] = "Shyam";

String name = list[0]; // Type casting not required (guarantee for type of elements present inside array

--------------------------------------------------------

ArrayList list2 = new ArrayList();

list2.add("Abhay");

String name2 = list2.get(0); // Gives RE

**Runtime Error**: incompatible types: Object cannot be converted to String.

**Fix**: String name2 =(String) list2.get(0); //add (String) to line 3 (type casting required)

**4) New syntax introduced to provide type safety and to resolve type-casting problem:-**

Ex- To hold only String type of objects we can create generic version of Arraylist as follows :

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

list.add(“Shivam”);

list.add(“Karan”);

list.add(new Integer(10)); // Gives CE

**Hence through Generics, we are getting Type Safety.**

String name = list.get(0); // Type casting not required.

**Hence through Generics, we can solve Type Casting problems.**

**5) Difference between Non-Generic and Generic version of ArrayList**

|  |  |
| --- | --- |
| ArrayList list = new ArrayList(); | ArrayList<String> list = new ArrayList<String>(); |
| Non- Generic version of ArrayList object | Generic version of ArrayList object. |
| Not Type Safe (as we can add any type of object) | Type Safe (we can add only String type objects) |
| Type casting is required (at the time of retrieval) | Type casting is NOT required (at the time of retrieval) |

**6) Polymorphism is only applicable to Base Type not Parameter Type.**

Base Type (Type Parameter/Type Variable)

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

Ex: List<String> list = new ArrayList<String>(); //Correct

Collection<String> list = new ArrayList<String>(); //Correct

List<Object> list = new ArrayList<String>(); // Gives CE: Incompatible types

Polymorphism concept only applicable to base type but not for parameter type.

(Usage of parent reference to hold child object is the concept of Polymorphism)

**7) For Type Parameter we can provide any Class or Interface name but NOT Primitives.**

Ex: ArrayList<int> x = new ArrayList<int>(); // Gives CE

Compilation Error: Unexpected type, found int , required reference(i.e. Integer is required).

**8) Till Java 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. Hence we can add any type of object to 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 preform type casting.

**From Java 1.5, a generic version of ArrayList Class is declared as follows :**

Class ArrayList<T>{ Class ArrayList<String>{

add (T t) add (String s)

T get(int index) String get(int index)

} }

Based on our Runtime requirement T will be replaced with our provided type. For ex- To hold only String type of objects a generic version of ArrayList can be created as above.

**9) Example of our own Generic Class**

class Gen<T>{

T obj;

Gen(T obj){

this.obj = obj;

}

public void show(){

System.out.println("The type of object: "+obj.getClass().getName());

}

public T getObj(){

return obj;

}

}

public class Main

{

public static void main(String[] args) {

Gen<String> list1 = new Gen<String>("Vicky");

list1.show(); //

System.out.println(list1.getObj());

Gen<Integer> list2 = new Gen<Integer>(10);

list2.show();

System.out.println(list2.getObj());

}

}

**Output:**

The type of object: java.lang.String

Vicky

The type of object: java.lang.Integer

10

**10) Bounded Types: We can bound the Type parameters for a particular range only by using ‘extends’ keyword.**

**Such types are called Bounded types.**

Ex: If we want to restrict to have only numbers.

Class Test<T extends Number> { //Number is parent for byte, short, int, long, float and double

}

**11) We can use extends keyword for interface in type parameter.**

Ex: class Test<T extends Runnable> //Here note that Runnable is an interface not class.

**12) Syntax for Bounded type:**

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 an Interface, then as the type parameter we can pass either X type, or its implementing classes.

**13)**

Class Test< T extends Number> {

}

Test<Integer> t1= new Test<Integer>(); //Correct

Test<String> t2 = new Test<String>(); //Wrong, It gives CE

**Compilation Error**: Type Parameter java.lang.String is not within its bound.

Class Test< T extends Runnable> {

}

Test<Runnable> t1= new Test<Runnable>(); //Correct

Test<Thread> t2 = new Test<Thread>(); //Correct

Test<Integer> t3 = new Test<Integer>(); //Wrong, It gives CE

**Compilation Error**: Type Parameter java.lang.Integer is not within its bound.

**14) We can define Bounded Types even in combination also.**

Ex: class Test< T extends Number & Runnable> { //Correct Syntax

}

As a type parameter we can take anything which should be child class of Number and implements Runnable Interface.

Ex: class Test< T extends Number & Runnable & Comparable> { //Correct Syntax

}

Ex: class Test< T extends Runnable & Number & Comparable > { //Incorrect Syntax

}

**Reason**: First we need to take classes then interfaces, if any(Like we used to do class A extends B implements C).

Ex: class Test< T extends Number & Thread > { //Incorrect Syntax

}

**Reason**: We can’t extend more than one classes simultaneously in Java. (Multiple Inheritance not allowed in Java)

**15) As a Type Parameter ‘T’, we can take any valid Java identifier, but it is convention to use ‘T’.**

Ex: class Test< T > { // Valid

}

Ex: class Test< X > { // Valid

}

Ex: class Test< A > { // Valid

}

Ex: class Test< Vicky > { // Valid

}

**16) We can take any number of Type Parameters separated by comma, based on our requirement.**

Ex: class Test< A,B > { // Valid

}

Ex: class Test< X,Y,Z > { // Valid

}

Inbuilt example:- class HashMap< K,V > { // K = Key Type, V = Value Type

}

HashMap<Integer,String> map = new HashMap<Integer,String>();

**17) Generics wildcard character (?)**

1. m1(ArrayList<String> list)

2. m1(ArrayList< ? > list1)

3. m1(ArrayList< ? extends X > list2) // X can be class or interface

4. m1(ArrayList< ? super X > list3)

We can call (1) method by only passing ArrayList of String type.

We can call (2) method by passing ArrayList of **any type**.

We can call (3) method by passing ArrayList of either X type or its child classes (or its implementing classes)

We can call (4) method by passing ArrayList of either X type of super classes of X

**Note: With ‘ ? ’ symbol at method level super keyword is allowed but at class level with ‘ T ’, super keyword is not allowed.**

For (1), we can add only String type values.

Ex: m1(ArrayList<String> list){

list.add(“A”); // Valid

list.add(null); // Valid (null is valid value for any type including String)

list.add(10); // Gives CE (since it is other than String type i.e. Integer here)

}

For (2) Within a method we can’t 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(ArrayList< ? > list) {

list.add(10.5); // Invalid

list.add(“A”); // Invalid

list.add(10); // Invalid

list.add(null); // Valid

}

This type of method are best suitable for READ ONLY operation.

For (3) Within a method we can’t add anything to the list except null because we don’t know the type exactly. This type of methods also best suitable for READ ONLY operations. X can be either Class or Interface.

For (4) If X is a Class then we can call this method by passing ArrayList of either X type or its super classes.

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.**

**Here we have a flexibility to add X type of Object** (which we didn’t have in above two scenarios).

m1(ArrayList< ? super X > list) {

list.add(X);

list.add(null);

}

**18) Some important declarations to check our understanding which are valid/invalid.**

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

ArrayList< ? > l = new ArrayList<String>(); // Valid

ArrayList< ? > l = new ArrayList<Integer>(); // Valid

ArrayList< ? extends Number > l = new ArrayList<Integer>(); // Valid

ArrayList< ? extends Number > l = new ArrayList<String>(); // Gives CE: incompatible types

ArrayList< ? super String > l = new ArrayList<Object>(); // Valid

ArrayList< ? > l = new ArrayList< ? >(); // Gives CE: Unexpected Type

ArrayList< ? > l = new ArrayList< ? extends Number >(); //Gives CE: Unexpected Type

**19) 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 requirements.

}

**Declaring type parameter at Method Level** :-

Note: We have to declare type parameter before return type.

class Test

{

public < T > void m1( T obj)

{

We can use ‘T’ anywhere within this method based on our requirement.

}

}

**20) We can define bounded types at method level also.**

public < T > void m1()

< T extends Number > // Valid

< T extends Runnable > // Valid

< T extends Number & Runnable > // Valid

< T extends Comparable & Runnable > // Valid

< T extends Number & Comparable & Runnable > // Valid

< T extends Runnable & Number > // CE: (First we have to take Class then Interface)

< T extends Runnable & Thread > //CE: (We can’t extend more than one Class)

**21) 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 is present, behaviour will be defined.**

class Test

{

public static void main(String[] args)

{

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

list.add("Vicky");

list.add("Shiva");

// list.add(10); //Gives CE **Generic Area**

m1(list);

System.out.println(list);

// list.add(10.5); //Gives CE

}

public static void m1(ArrayList list)

{

list.add(10);

list.add(10.5); **Non – Generic Area**

list.add(true);

}

}

**Output**: [Vicky, Shiva, 10, 10.5, true]

**22) Conclusions :-**

* The main purpose of Generics is to provide Type Safety and to resolve Type Casting problems.
* Type Safety and Type Casting both are applicable at compile time. Hence Generics concept also applicable only at compile time but not at runtime.
* At the time of compilation at the last step, Generics syntax will be removed and hence for the JVM Generics syntax won’t be available.

Hence the following declarations are equal:

ArrayList list = new ArrayList<String>();

ArrayList list = new ArrayList<Integer>();

ArrayList list = new ArrayList<Double>();

ArrayList list = new ArrayList ();

**Example**: ArrayList list = new ArrayList<String>();

list.add(10);

list.add(10.5);

list.add(true);

System.out.println(list);

Output: [10, 10.5, true]

**Reason**: Compiler is always going to check reference side, JVM will check runtime object. Since at the time of compilation Generics syntaxes are removed and hence for runtime Generics syntax won’t be available. So, there are no errors I this example. If Generics concept was present at runtime then we would certainly get Runtime Exceptions, because object is of type <String>.

The following declarations are equal:

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

ArrayList<String> list = new ArrayList();

For these ArrayList objects, we can add only String type objects.

**23)**

class Test{ After Compilation

public static void m1(ArrayList<String> list){ m1(ArrayList list)

}

public static void m1(ArrayList<Integer> list){ m1(ArrayList list)

}

}

**Compilation Error:** name clash: Both methods have the same erasure.

**At Compile Time :-**

* Compile code normally by considering Generic syntax
* Remove Generic Syntax
* Compile once again resultant code. (Compilation failed at this step)

==XX=================================XX====================================XX===