Generics

- Introduction
- Generic classes
- Bounded Types
- Generic methods
- Valid card Character?
- Communication with non-Generic code
- Conclusion

Introduction

- Arrays are always safe with respect to type.

Example:

If our programmer requirement is to add only String objects then we can go for String[] array for this array 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.

Example:

```
String[] s=new String[600];

s[0]=”company”;

s[1]=”cernsystem”;

s[2]=new Student();  // incompatable types

                        found:Student

                        Requriment:String
```

- Hence in the code of Arrays we can always give the guarantee about The type of elements. String[] array contains only String objects (i.e.String)due to this arrays are always safe to use with respect to type.
- But collections are not safe to use w.r.t type.
  Example: If our program requirement is to hold only String object & if we are using ArrayList,By mistake and are trying to add any others type to the List we won’t get any Compile time error But program may fail at Runtime.
  Example:

```
ArrayList l=new  ArrayList();
    l.add(“Priya”);
    l.add(“Shaila”);
    l.add(new Student());
      |
    |
String name1=(String)l.get(0);
String name2=(String)l.get(1);
String name3=(String)l.get(2);//R.E. ClassCastException
```

- There is no guarantee that collection can hold a particular type. Hence w.r.t Type collection are not safe to use.
  Case 2: In the case of Arrays at the time of retrieval it is not required to perform any type casting.

  Example:

```
String[] s=new String[100];
          s[]="Priya"
            |
            |
 String name1=s[0];//Typecasting is not required.
```

- But in the case of Collection at the time of retrieval compulsory we should perform Typecasting is not mandatory which is a bigger headache to the programmer.
- To over come the above problems of collection (Type safe & TypeCasting) sun people introduced Generics concept in 1.5 version. Hence the main objective of Generic concept is.

1) To provide Type Safety to the Collections. So that they can hold only a particular type of object.
2) To resolve type casting problem.

Example:

To hold only String type of objects

a generic version of ArrayList we can declear as follows

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

ArrayList //returntype

<string> //parameter type
```

- For this Arraylist 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.
  i.e. we are getting Type-safety.

  ```
  l.add("Priya");
  l.add("Sujata");
  l.add("10");
  l.add(10); //C.E. cannot find symbol
                          symbol:method add(int)
                         location:class ArrayList<String>
  ```
- At the time of retrieval it is not required to perform any Typecasting.
  String name= l.get(0);//Type casting is not Required.

➢ Conclusion 1:

- Usage of parent class reference to hold child class object is considered as polymorphism.
- Polymorphism concept is applicable only for base type. but not for parameter type.

  Example:

  ```
     ArrayList<Integer> l= new ArrayList<Integer>();
      List<Integer> l=new ArrayList<Integer>();
     Collection<Integer> l=new ArrayList<Integer>();
      List<Objective>l=new ArrayList<Integer>();//C.E.incompatible Types
    required:AL<Integer>
     found:List<Object>
  ```

➢ Conclusion 2:

- For the parameter type we can use any class or interface name & we can't use primitive type validation that leads to Compile Time Error.
  Example:

  ```
   ArrayList<int> l= New ArrayList<int>();//C.E.  Unexpected type
   found:int
   required:reference
  ```

Generic Classes

- Untill 1.4 version non-Generic version of ArrayList class is declared as follows

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

- The argument to the add() method is object. Hence we can add any type of object due to this we are not getting Type Safety.
- The return type of get() method is object. Hence Erythematic Exception occur at the time of retrieval so it is compulsory to perform Typecasting.
- But in 1.5 version a generic class of ArrayList class is declared as follows.

```
     Class ArrayList<T>
      {
        Add<T t>
         T get (int index)
      }
```

- Based on our runtime requirement Type Parameter will be replaced with corresponding provided Type.

  Example: To hold only String Type of objects we have to create Generic version of ArrayList Object as follows

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

- For this requirement the corresponding loaded version of ArrayList Class is

```
class ArrayList<String>
{

add(String s)
```

```
String get(int index)
}
```

- Add() method can take String as the argument hence we can add only String type of object. By mistake if we are trying to add any other type we will get compile time Error.i.e.we are getting Type-safety.
- The return type of get() method is String, Hence at the time of retrieval we can assign directly to the String type variable it is not required to perform any Type Casting.

Note:

1) As the type parameter we can use any valid java identifier but it is Convention to use 'T'.

   Example:

```
1)  class AL<x>
      {
      }  //valid


2) class AL<priya>
      {
      }  //valid
```

2) We can pass any no. of parameter & need not be one.

   Example:

```
   class HashMap<K,V>
    {


    }
 HashMap<String,Integer>m=new     HashMap<String,Integer>();
String //Key type
 Integer//Value type
```

- Through generic we are associating a type parameter to the classes.such type of parameterized classes are called generic classes.
- We can define our own generic classes also.

  Example:

```
 Class Gen<T>
```

```
            {
             T ob;
            Gen<T ob>
            {
             this.ob=ob;
          }
            public void show()
            {
               System.out.println("The type of ob
          is:"+ob.getClass().getvalue());
          }
           Public T getOb()
           {
              return ob;
           }
          }
Example:
  Class GenDemo
    {
     public static void main(String arg[])
      {
       Gen<String> g1=new Gen<String>("durga");
   1     g1.show();//the type of object
                          is:java.lang.String
       System.out.println(g1.getob());durga

Gen<Integer> g2=new Gen<Integer>(10);
      g1.show();//the type of object
                          is:java.lang.integer
       System.out.println(g2.getob());10
        }
    }
```

Bounded Types:

- We can bound the type parameter for a particular range by using extends keyword
  Example:1)
  ```
    class Test<T>
     {
     }
  ```
- As the type parameter we can pass any type hence it is unbounded type
  Test<String>t1=new Test<String>();
  Test<String>t2=new Test<Integer>();
  Example:2)
  ```
    class Test<T>
     {
     }
  ```
- As the type of parameter we can pass either Numbers type or its child classes. it is Bounded type.

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

Test<String>t2=new Test<Integer>();//C.E: Type parameter

java.lang.String is not with in bound

- We can't bound type parameter by using implements & super keywords
  Example 1)
  ```
      class Test<T implements Runnable>
      {
      } //invalid
  ```

  Example 2)
  ```
      class Test<T super integer>
      {
      } //invalid
  ```

Example

```
 class Test<T extends X>//class/interface
  {
}
```

- x $\longrightarrow$ can be either class/interface.
- If x is a class then as the type parameter we can provide either X type as its implementation classes.

  Example:

  ```
    class Test<T extends Runnable>
     {
   }
   Test<Runnable> t1=new Test<Runnable>();
   Test<Thread> t2=new Test<Thread>();
    Test<String> t3=new Test<String>();//C.E=type parameter
  java.lang.String is not within it's Bound
  ```

- We can bound the parameter even in combination also

  Example:

  ```
    class Test<T extends Number & runnable>
  ```

- As the type of parameter we can pass any type which is the child class Of number & implements & Run able interface.

  Example:

  1) class Test<T extends Number & comparable>
  2) class Test<T extends Number & Run able comparable>
  3) class Test<T extends Number & Thread>//we can't extends more than one class at a time
  4)class Test<T extends Run able & Number>//we have to take first & then interface

Generic Methods & Valid Class Character

1) m1(ArreyList<String> l) //valid

- This method is applicable for arraylist<String>(Arraylist of only String type)
- Within the method we can add String type objects and null to the List if we are trying to add any other type we will get compile time error.
- Example:m1(ArreyList<String> l)
  ```
  {
   l.add("A");     //valid
  l.add(null);     //valid
  l.add(10);       //Invalid
  }
  ```

  2)m1(ArrayList <? extends x> l) //valid
- We can call this method by passing Arraylist of any type,but within the method we can't add any type except null to the List. Because we don't know the type exactly.
- Example:m1(ArreyList<?> l)
  ```
  { l.add(null);     //valid
  l.add("A");     //Invalid
  l.add(10);       //Invalid
  }
  ```

  3)m1(ArrayList <? extends x> l) //valid
- 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 class.
- In this case also we can't add any type of elements to the list except null.

4)m1(ArrayList <? supers x> l) //valid

- If x is a class then this method is applicable for ArrayList of either x type or its super classes.
- If x is an interface then this method is applicable for ArrayList of either x type or super classes of implementation class of x.
- With in the method we can add only x type object and null to the list.

Qu)Which of the following declaration are valid?

1)AL<String> l=new AL<String>(); //Valid

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

3)AL<? extends String> l=new AL<String>(); //Valid

4)AL<? super String> l=new AL<String>(); //Valid

5)AL<? extends Object> l=new AL<String>(); //Valid

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

7)AL<? extends Number> l=new AL<String>();

//C.E.Incompatible types

required:AL<? extends Number>

found:AL<String>

8)AL<?> l=new AL<? extends Number>(); //Invalid

9)AL<?> l=new AL<?>(); //C.E.unexpected type:

required:class or interface without bounds

found:?

- We can define the type parameter either at class level or at method level.

- **Declaring type parameter at class level:**

```
class Test<T>

{

T ob;

public T get()

{

return ob;

}

}
```

- **Declaring type parameter at method- level:**

  We have to declare the type paremeter just before return type .

  1) class level:

```
class Test<T>

{

T ob;

public T get()

{

return ob;

}

}
```

2)method level:

```
class Test

{

public <T > void m1(T)

{

T ob;

}

}
```

```
1)<T extends Number>                    //Valid

2)<T extends Runnable>                  //Valid

3)<T extends Number and Runnable>       //Valid

4)<T extends Number and compareble> //Valid

5)<T extends Number and Thread>         //Invalid

6)<T extend Runnable and Thread>        //Invalid
```

Communication with non-Generic code:

- To provide compatibility with old version SUN people compromised
  The concept of generics in very few areas.
- The following is one such area.
  Example:
  Class test
  {

  public static void main(String ar())

```
                {
                ArrayList<String> l=new ArrayList<String>();
                l.add("A");
                l.add(10);
                m1(l);
                System.out.println(l);      [A,10,10.5,true]
                //l.add(10);                //   C.E.
                }

        Public static void m1(Arraylist l)

        {

        l.add(10);

        l.add(10.5);

        l.add(true);

         }

}
```

Conclusion:

Generic concept is applicable only at compile time to provide the safety and to resolve type casting problems. At runtime there is no such type of concept.

Hence the following declarations are equal:

Example:

1)ArrayList al=new ArrayList();

2)ArrayList al=new ArrayList<String>();

3)ArrayList al=new ArrayLis<Integer>();

Example:

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

              l.add("A");

              l.add(10);

l.addtrue);

         System.out.println(l);
```

The following two declarations are equal and there is no difference

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

Both are equal.