Table of Contents

1 Generics Introduction

The main objective of Generics is **to provide "Type-Safety" and to resolved "Type casting" problems**

1.1 Type- Safety

Naming Dictionary, to do this, sharing naming is not a problem. All the names can be shared in the form of Array or ArrayList.

1.1.1 Using Arrays

Arrays are type safe; that is we can give guarantee for the elements present inside array. If our programing requirement is to hold only string type of objects, we can choose string array. By mistake if you are trying to add any other type of objects, we will get compile time error.

Let's say we have 10000 names then –

```
String[] s = new String[10000];

S[0] = "Naren";

S[1] = "Kumar";

// S[2] = 10; // Incompatible type found - java.lang.Integer required java.lang.String.

S[2] = "shiva"; // correct the value at index "2"
```

String array can contains 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 with respect to "Type". That means arrays are type safe.

1.1.2 Using ArrayList

But collections are not type safe, that is we can't give the guarantee for the type of elements present inside collection.

For example, If our programming requirement is to hold only String type of objects, and we if choose ArrayList, By mistake if you are trying to add any other type objects we won't get any compile time error. But the program may fail at runtime.

```
ArrayList al = new ArrayList();

Al.add("naren");

Al.add("Kumar");

Al.add(new Integer(10));
```

At run time

String name = (String)al.get(0)

String name1 = (String)al.get(1)

String name3 = (String)al.get(2) // ClassCastException

Hence we can't give the guarantee for the type of elements present inside collection. Due to this collections are not safe to use with respect to "Type". That is collections are not type safe.

Arrays are not growable which are fixed size, so we may need to use collections which supports growable size.

1.2 Type - Casting

In the case of arrays at the time of retrieval it's not required to perform type casting because, there is a guarantee for the type of elements present inside array.

String[] s = new String[10000];

S[0] = "Naren";

S[1] = "Kumar";

String name = s[0];

But in the case of collections, at the time of retrieval compulsory we should perform type casting. Because there is no guarantee for the type of elements present inside collections.

ArrayList al = new ArrayList();

Al.add("naren");

Al.add("Kumar");

String name = al.get(0); **// Incompatible type found - java.lang.Integer required java.lang.String.**

String name = (String) al.get(0); // Type casting is mandatory.

Hence Type casting is required.

To overcome above problems of collections, Generic concepts in 1.5 versions, hence main objects of generics are to Provide type safety, and to resolve Type casting problems.

1.3 Non-Generics vs Generic Collections

For example, to hold only string type of objects, we can create generic version of Array List object as follows.

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

For this array list, we can add only string type of objects, by mistake if we are trying to add any other types then we will get compile time error, Through Generics we can achieve type safety like below.

```
al.add("Naren")

// al.add(new Integer(10)); // Compile time error.

al.add("Sri")
```

At the time of retrieval, we are not required to perform type casting, So Through Generics we can achieve type casting like below., Hence through Generics, we can solve type casting problem.

```
String name =  al.get(0);
```

ArrayList = new ArrayList ()	**ArrayList <String> = new ArrayList <>**
It is non-generic version of AarrayList Object	It is generic version of AarrayList Object
For this array list we can add any type of objects, hence it's not type safe	For this array list we can add only String type of objects, hence it's type safe
At the time of retrieval, we need to perfume type casting.	At the time of retrieval, No need to perfume type casting

1.4 Conclusions

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

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

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

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

ArrayList <Object> al =  new ArrayList<String>(); // Not Valid,

        In compatible types, found AL<String> but required Al<Object>
```

- For the type parameter , we can provide any class or interface name but not primitives, if we are trying to provide primitive then we will get compile error

```
ArrayList <int> al =  new ArrayList<int>(); // Not Valid,
```

2 Generics Classes/Interface

Until 1.4 version a non-generic version of Arraylist class id declared as follows,

```
Class ArrayList {

        add(Object 0){…}

        Object get(int index) {..}

}
```

- 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 perform type casting.

But in 1.5 version, a generic version of arrayList class can be declared as follows –

```
Class ArrayList <T> {

        add(T  0){…}

        T get(int index) {..}

}
```

- Based on our runtime requirement T will be replaced with provided Type, For example to hold only String type of objects a generic version of ArrayList object can be created as follows

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

- For this requirement, compiler consider version of ArrayList class is as follows –

```
Class ArrayList <String> {

        add(String  0){…}

        String get(int index) {..}

}
```

- Here 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.Hence, we are getting type safety.

```
al.add(“Naren”)

// al.add(new Integer(10)); // Compile time error.
```

Compile time Error: cannot find Symbol: Method add(j.l.Integer) location ArrayList<String>

- The return type of get method is “String” hence at the time of retrieval we are not perform type casting.

  String name = al.get(0);

In generics, we are associating a type parameter to the class, such type of parameterize classes are nothing but **Generic classes or Template classes.**

Based on our requirement we can define iur own generic classes also, like below –

Account<Gold> a = new Account<Gold> ();

Account<Platinum> p = new Account< Platinum> ();

2.1 Custom Generics Class

```
Class Gen <T> {
        T Obj;
        Gen (T Obj) {
                This.obj = obj;
        }
        Public void show() {
                Sout(“The type of object” + obj.getClass().getName());
        }
        Public T  getObj() {
                Return obj;
        }
}
Class GenDemo {
        Public static void main(String[] args) {
                Gen<String> strGen = new Gen<String>(“test”);
                strGen.show();
                SOUT(strGen.getObj());
                Gen<Integer> intGen = new Gen< Integer >(10);
                intGen.show();
                SOUT(intGen.getObj());
                Gen<Double> doubleGen = new Gen< Double >(10.5);
                doubleGen.show();
                SOUT(doubleGen.getObj());
        }
}
```

3 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> {

}
```

For the above Gen class as a type parameter we can pass any type parameter, hence it is **unbounded type.**

```
Test<String> t1 = new Test<>();

Test<Integer> t2 = new Test<>();
```

3.1 Syntax for Bounded type

X Can be either class or 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 as a type parameter either X type or its implementation classes.

```
Class Test<T extends X> {

}
```

Example 1 –

```
Class Test<T extends Number> {

}

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

Test<String> t1 = new Test< String >(); // Not valid
```

Compile time error : Type parameter j.l.String is not with in its bound

Example 2 –

```
Class Test<T extends Runnable> {

}

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

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

Test<integer> t1 = new Test<Integer>(); // Not valid
```

Compile time error : Type parameter j.l.Integer is not with in its bound

3.2 Bounded type with combination

We can define bounded type even in combination also.

For example, As a type parameter, which should be child class of Number and should implement Runnable interface (Both should satisfy).

Below are the possible Declarations –

Class Test<T extends Runnable & Comparable> {}

Class Test<T extends Number & Runnable & Comparable> {}

Class Test<T extends Runnable & Number > { } // Invalid.

Class should be declared first followed by Interface

Class Test<T extends Number & Thread> {} // Invalid,

Can't extend more than one class.

Creating object:

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

Test<**Thread**> t1 = new Test< **Thread** >();

Test<integer> t1 = new Test<Integer>(); // valid

3.3 Conclusions

- We can define bounded types only by using "extends" keyword and we can't use implements and super keywords. But we can replace implements purpose with "extends" keyword.

  Below declaration are **valid**.

  Class Test<T extends Runnable> {.....}

  Class Test<T extends String> {.....}

  Below declaration are **not valid**.

  Class Test<T implements Runnable> {.....}

  Class Test<T super String> {.....}

- As the type parameter "T" we can take any valid java identifier, but it is convention to use "T"

  Class Test<T> {.....}
  Class Test<Naren> {.....}

- Based on our requirement, we can declare any number of type parameters and all these type parameters should be separated with “,”

  Class Test<A, B> {……}

  Class Test<X, Y, Z> {……..}

  Class HashMap<K, V> {……..}

4 Generic Methods and Wild Card Characters (?)

4.1 Generic method First Type of flavor

Consider following method –

m1(ArrayList<String> al)

With what parameters we can call a method?

We can call this method by passing only arrayList<String>, that is with only string type.

```
ArrayList <String> al = new ArrayList < String >();
    m1(al)
```

what are the things we can do?

But within the method we can add we add only string type of objects to the list. By mistake if we are trying to add any other types, then we will get compile time error.

```
m1(ArrayList<String> al) {
    l.add("a");
    l.add(null);
    l.add(10); // Invalid
}
```

4.2 Generic method Second Type of flavor

Consider following method –

m1(ArrayList<?> al)

With what parameters, we can call method?

We can call this method by passing ArrayList of any type

```
ArrayList <String> al = new ArrayList < String >();
m1(al)
ArrayList <Integer> al = new ArrayList <Integer>();
m1(al)
```

what are the things we can do?

But within the method we can't add anything to the list except "null", because we don't know the type exactly. Null is allowed because it's a valid value for any type.

```
m1(ArrayList<?> al) {
    l.add("a"); // Invalid
    l.add(null); -- // Valid
    l.add(10); // Invalid
}
```

Note: This is type of method best suitable only for read-only operations, but not for write.

4.3 Generic method Third Type of flavor

Consider following method –

m1(ArrayList<? extends X> al)

With what parameters, we can call method?

- X can be either class or interface,
- 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 classes.

what are the things we can do?

But within the method we can't add anything to the list except "null", because we don't know the type exactly. Null is allowed because it's a valid value for any type.

```
m1(ArrayList<? extends X > al) {
    l.add("a"); // Invalid
    l.add(null); -- // Valid
    l.add(10); // Invalid
}
```

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

4.4 Generic method Fouth Type of flavor

Consider following method –

m1(ArrayList<? super X> al)

With what parameters, we can call method?

- X can be either class or interface,
- 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 Its **super class** of implementation class of X.

Example - ***Object is super class of implementation class (Thread) of Runnable type.***

what are the things we can do?

But within the method we can add X type of objects and "null" to the list.

```
m1(ArrayList<? extends X > al) {
    l.add(X);
    l.add(null); -- // Valid
}
```

4.5 Declarations

4.5.1 Valid declarations

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

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

ArrayList <?> al = new ArrayList < Integer >();

ArrayList <? extends Number> al = new ArrayList < Integer >();

ArrayList <? extends String> al = new ArrayList < Object >();

4.5.2 Invalid declarations

ArrayList <? extends **String**> al = new ArrayList < **Integer** >();

CE: Incompatible types foundAl <String> required <Al<? externds Number>

ArrayList <**?** > al = new ArrayList <**?**>();

CE: Unexpected typefound ? required : Class or Interface without Bounds

ArrayList <? > al = new ArrayList **<? extends Number**>();

CE: Unexpected type found ? extends Number required : Class or Interface without Bounds

4.6 Generic Type Methods

We can declare type parameters either at Class level, or at Method level

4.6.1 Declaring Type parameter at class level

```
Class Test<T> {
    We can use “T” (type) with in the class based on our requirement.
}
```

4.6.2 Declaring Type parameter at Method level

We have to declare type parameter just before return type

```
Class Test  {
    Public <T> void m1(T obj) {
            We can use “T” (type) with in the class based on our requirement.
    }
}
```

4.7 Bounded Types at Methods level

Like Class level bound types, we can also have bounded types at method level

We can define bounded types even at method level also.

Below are Valid declarations -

Public <T> void m1(T obj) { }
Public <T extends Number>
Public <T extends Number & Runnable>
Public <T extends Comparable & Runnable>
Public <T extends Number & Comparable & Runnable>

Below are InValid declarations -

Public <T extends Runnable & Number > // Class should be declared first followed by Interface

Public <T extends Thread & Number > // We can't extend more than one class.

Method level and Class level parameters can be used in combinations.

Example Class-

```
import java.util.Collection;
public class MyClass {
        public static  <T>   T addAndReturn(T element, Collection<T> collection){
           collection.add(element);
           return element;
        }
        public static <T> T addAndReturnValue(T element, Collection<T> collection){
           collection.add(element);
           return element;
        }
        public  <T>  T getInstance(Class<T> theClassInstance) throws IllegalAccessException,
        InstantiationException {
              return  theClassInstance.newInstance();
           }
        public static <T> T read(Class<T> theClass, String sql)
           throws IllegalAccessException, InstantiationException {
           //execute SQL.
           T o = theClass.newInstance();
           //set properties via reflection.
           return o;
        }
}
```

5 Communication with non-generic Code

- If we send generic object to non-generic area, then its starts behaving like non-generic object.
- Similarly, if we send non-generic object to generic area, then its starts behaving like generic object.
- That is based on the location object behavior will be derived.

```
Classs Test {
        P s v m (Srting[] args) {
                ArrayList<String> al = new ArrayList<String>();
                al.add(“Naren”);
                al.add(“Kumar”);
                al.add.(10.5) // compile time error
                m1(al);
                al.add.(10.5) // compile time error
                al.add(“Naren Kumar”);
        }
        P s v m1(){
                al.add(“Naren”); // valid
                al.add(10); // valid
                al.add(10.5); // valid
                al.add(true); // valid

        }
}
```

6 Conclusion

- To provide “type safety” and to resolve “type casting” problems
- Type safety and type casting, both are applicable at compile time**, hence generics concepts applicable at compile time but not at runtime.**

  Al al = new Al <Sting> ();
- Compiler only checks “reference type” (“al”), But JVM will always check “runtime object” (actual object ArrayList.
- 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.

  Al al = new Al <Sting> ();
  al.add(“Naren”); // valid
  al.add(10); // valid
  al.add(10.5); // valid
  al.add(true); // valid
  Sout(al) =➔ [“Naren”, 10, 10.5, true]

  We won’t get any compile time and runtime errors and the output will be - [“Naren”, 10, 10.5, true]
- The following **declaration are equal.**

  Al al = new Al <Sting> ();
  Al al = new Al <Integer> ();
  Al al = new Al <Double> ();
  Al al = new Al ();
- The following **declaration are equal.** For these AL objects we can add only string type of objects.

  Al<String> al = new Al <Sting> ();
  Al<String> al = new Al ();
- The following **code fails**.

  Class Test {
  Public void m1(Al<String> al) {
  }
  Public void m1(Al<Integer> al) {
  }
  }
  At compile time -

  a. Compile code normally by considering generics syntax.
  b. Remove generics syntax.
  c. Compile once again resultant code from step 2, here its fail to compile by throwing exception – **name clash, both methods having same erasure.**

7 Examples

Example class –

```
package testing.com;

public class GenericFactory<T> {

   Class theClass = null;

   public GenericFactory(Class theClass) {
      this.theClass = theClass;
   }

   public  T createInstance() throws IllegalAccessException, InstantiationException {
      return (T) this.theClass.newInstance();
   }

   public <K>  K getInstance(Class<K> theClassInstance) throws IllegalAccessException,
InstantiationException {
      return  theClassInstance.newInstance();
   }
}
```

Example class –

```
package testing.com;
import java.util.ArrayList;
import java.util.List;
public class Test<T extends Comparable<T>> {

        public static void main(String[] args){
        try {
                GenericFactory<MyClass> factory =  new GenericFactory<>(MyClass.class);
                MyClass myClassInstance = factory.createInstance();

                String ele = "testing";
                List<String> stringList = new ArrayList<>();
                myClassInstance.addAndReturn(ele, stringList);
                Integer integerElement = new Integer(123);

                List<Integer> integerList = new ArrayList<>();
                Integer theElement = myClassInstance.addAndReturn(integerElement, integerList);
                GenericFactory<SomeClass> factory1 =  new GenericFactory<>(SomeClass.class);
                SomeClass someObjectInstance = factory1.createInstance();
                MyClass employee   = myClassInstance.read(MyClass.class, "select * from drivers
                where id=1");
                SomeClass vehicle   = myClassInstance.read(SomeClass.class, "select * from
                vehicles where id=1");

                // Super and subclasses
```

```
                List<A> listA = new ArrayList<A>();
                List<B> listB = new ArrayList<B>();
                List<C> listC = new ArrayList<C>();
                // not possible
                /*
                listA = listB;
                listB = listA;
                */
                processElements(listA);
                //processElements(listB);
                /*
                Generic Wildcards
                The generic wildcard operator is a solution to the problem explained above.
                The generic wildcards target two primary needs:
                Reading from a generic collection
                Inserting into a generic collection
                  */
                List<String> listString = new ArrayList<String>();

                // method still cannot insert elements into the list,
                // because you don't know if the list passed as parameter is typed to the class A, B or
                C.
                processElementsUnknown(listA);
                processElementsUnknown(listB);
                processElementsUnknown(listC);
                processElementsUnknown(listString);
                // method still cannot insert elements into the list,
                // because you don't know if the list passed as parameter is typed to the class A, B or
                C.
                processElementsWildcard(listA);
                processElementsWildcard(listB);
                processElementsWildcard(listC);
                //processElementsWildcard(listString);

                List<Object> listObject = new ArrayList<Object>();
                insertElements(listA);
                //insertElements(listB); // can't do this
                //insertElements(listC); // can't do this
                insertElements(listObject);
                } catch (IllegalAccessException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        } catch (InstantiationException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }
}
```

```
    public static void processElements(List<A> elements){
      for(A o : elements){
        System.out.println(o.getValue());
      }
    }
    public static void processElementsUnknown(List<?> elements){
      for(Object o : elements){
        System.out.println(o);
      }
    }
    public static void processElementsWildcard(List<? extends A> elements){
      for(A a : elements){
        System.out.println(a.getValue());
      }
    }
    public static void insertElements(List<? super A> list){
       list.add(new A());
       list.add(new B());
       list.add(new C());

       Object object = list.get(0);

       // But this is not valid:
       // A object1 = list.get(0);
    }
    private <K extends Comparable<K>> K maximum(K x, K y, K z) {
         K max = x;   // assume x is initially the largest

         if(y.compareTo(max) > 0) {
            max = y;   // y is the largest so fr
         }

         if(z.compareTo(max) > 0) {
            max = z;   // z is the largest now
         }
         return max;   // returns the largest object
    }

    public <K>  K getInstance(Class<K> theClassInstance) throws IllegalAccessException,
    InstantiationException {
           return  theClassInstance.newInstance();
     }
}
```