Genetics

16th dec Generics (Jdk 1.5)

17 December 2022 21:55

Agenda:

1. Introduction
2. Type-Safety
3. Type-Casting
4. Generic Classes
5. Bounded Types
6. Generic methods and wild card character(?)
7. Communication with non generic code
8. Conclusions

Deff : The main objective of Generics is to provide Type-Safety and to resolve Type-Casting problems.

Case 1: Type-Safety

Arrays are always type safe that is we can give the guarantee for the type of elements present inside array.

For example if our programming requirement is to hold String type of objects it is recommended to use String array.

In the case of string 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.

eg:

```
String name[] =new String[500];
name[0] = "Navin Reddy";
name[1] = "Haider";
name[2] = new Integer(100); //CE: incompatbile types found: java.lang.Integer
 required: java.lang.String
```

That is we can always provide guarantee for the type of elements present inside array and hence arrays are safe to use with respect to type that is arrays are type safe.

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

For example if our programming requirement is to hold only string type of objects

it is never recommended to go for ArrayList.

By mistake if we are trying to add any other type we won't get any compile time error but the program may fail at runtime.

```
eg:
 ArrayList al =new ArrayList();
al.add("NavinReddy");
al.add("Haider");
al.add(new Integer(10));
;;;
String name1 = (String)al.get(0);
String name2 = (String)al.get(1);
String name3 = (String)al.get(2);//Exception in thread "main" ::
java.lang.ClassCastException
java.lang.Integer cannot
be cast to java.lang.String
```

Hence we can't provide guarantee for the type of elements present inside collections that is collections are not safe to use with respect to type.

Case 2: Type-Casting

In the case of array at the time of retrieval it is not required to perform any type casting.

```
eg::
String name[] =new String[500];
 name[0] = "Navin Reddy";
 name[1] = "Haider";
;;;;
 String data =name[0];//here type casting is not required.
```

But in the case of collection at the time of retrieval compulsory we should perform type casting otherwise we will get compile time error.

```
eg::
ArrayList al =new ArrayList();
al.add("NavinReddy");
al.add("Haider");
String name1= al.get(0);//CE: incompatible types: found : java.lang.Object required:
java.lang.String
String name1=(String) al.get(0);//At the time of retrieval type casting is madantory
That is in collections type casting is bigger headache.
```

To overcome the above problems of collections(type-safety, type casting)sun people introduced generics concept in 1.5v

hence the main objectives of generics are:

1. To provide type safety to the collections.

2. To resolve type casting problems.

To hold only string type of objects we can create a generic version of ArrayList as follows.

```
ArrayList<String> al =new ArrayList<String>();
al.add("NavinReddy");
al.add(10);//CE: can't find symbol
symbol: method add(int)
location : class
java.util.ArrayList<java.lang.String>
al.add(10)
```

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

that is through generics we are getting type safety.

At the time of retrieval it is not required to perform any type casting we can assign elements directly to string type variables.

```
eg:ArrayList<String> al =new ArrayList<String>();
al.add("NavinReddy");
;;;;
;;;;
String name =al.get(0);//type casting is not required as it is an TypeSafe
```

That is through generic syntax we can resolve type casting problems.

Conclusions

==========

1. Polymorphism concept is applicable only for the base type but not for parameter Type [usage of parent reference to hold child object is called polymorphism]. → to achieve this we use wildcard -]

```
eg: ArrayList<String> al =new ArrayList<String>();
 List<String> al =new ArrayList<String>();
 Collection<String> al =new ArrayList<String>();
 Collection<Object> al =new ArrayList<String>();//CE: incompatible types
```

Base Type — Parameter Type

But we don't say it, it is indirect polymorphism of Type parameter.

2.Collections concept applicable only for objects , Hence for the parameter type we can use any class or interface name but not primitive value(type).Otherwise we will get compile time error.

```
eg: ArrayList<int> al =new ArrayList<int>();//CE: unexcpected type
found:primitive
required:
reference
```

19th dec Generics

20 December 2022 16:53

Generic classes:

Until 1.4v a non-generic version of ArrayList class is declared as follows. e.g.

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

add() method can take object as the argument and hence we can add any type of object to the ArrayList.

Due to this we are not getting type safety.

The return type of get() method is object hence at the time of retrieval compulsory

we should perform type casting.

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

|=> Type parameter

```
class ArrayList<T>{
    add(T t);
    T get(int index)
    }
```

Based on our requirement T will be replaced with our provided type.

For Example to hold only string type of objects we can create ArrayList object as

follows.

Example:

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

For this requirement compiler considered ArrayList class is

Example:

```
class ArrayList<String>{
    add(String s);
    String get(int index);
    }
```

add() method can take only string type as argument hence we can add only string type of objects to the List.

By mistake if we are trying to add any other type we will get compile time error.

eg#1.

```
ArrayList<String> al =new ArrayList<String>();
al.add("NavinReddy");
al.add(10);//CE: can't find symbol
```

symbol: method add(int)

location : class

java.util.ArrayList<java.lang.String>

al.add(10)

eg#2.

```
ArrayList<String> al =new ArrayList<String>();
al.add("NavinReddy");
String name = al.get(0);//type casting is not requried
```

Hence through generics we are getting type safety.

At the time of retrieval it is not required to perform any type casting we can assign its values directly to string variables.

In Generics we are associating a type-parameter to the class, such type of parameterised classes are nothing but Generic classes.

Generic class : class with type-parameter.

Based on our requirement we can create our own generic classes also.

Example:

```
class Account<T>
{}
Account<Gold> g1=new Account<Gold>();
Account<Silver> g2=new Account<Silver>();
```

Example:

```
class Gen<T>{
    T obj;
    Gen(T obj){
    this.obj=obj;
    }
    public void show(){
    System.out.println("The type of object is :"+obj.getClass().getName());
    }
    public T getObject(){
    return obj;
    }
    }
    class GenericsDemo{
    public static void main(String[] args){
    Gen<Integer> g1=new Gen<Integer>(10);
    g1.show();
    System.out.println(g1.getObject());
    Gen<String> g2=new Gen<String>("iNeuron");
    g2.show();
    System.out.println(g2.getObject());
    Gen<Double> g3=new Gen<Double>(10.5);
    g3.show();
    System.out.println(g3.getObject());
    }
    }
    Output:
    The type of object is: java.lang.Integer
    10
    The type of object is: java.lang. String
    iNeuron
    The type of object is: java.lang. Double
    10.5
```

Note: To get the underlying object of any reference type we use a method called

ref.getClass().getName()

eg!

```
interface Calculator{}
class Casio implements Calculator{}
class Quartz implements Calculator{}
class Kadio implements Calculator{}
Calculator c1 =new Casio();
System.out.println(c1.getClass().getName());//Casio
Calculator c2 =new Quartz();
System.out.println(c2.getClass().getName());//Quartz
Calculator c3 =new Kadio();
System.out.println(c3.getClass().getName());//Kadio
```

Bounded types

We can bound the type parameter for a particular range by using extends keyword such types are called bounded types.

Example 1:

```
class Test<T>
{}
Test <Integer> t1=new Test< Integer>();
Test <String> t2=new Test < String>();
```

Here as the type parameter we can pass any type and there are no restrictions hence it is unbounded type.

Example 2:

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

If x is a class then as the type parameter we can pass either x or its child classes.

If x is an interface then as the type parameter we can pass either x or its implementation classes.

eg#1.

```
class Test <T extends Number>{}
class Demo{
public static void main(String[] args){
Test<Integer> t1 = new Test<Integer>();
Test<String> t2 = new Test<String>(); //CE
}
}
```

```
eg#2.
class Test <T extends Runnable>{}
class Demo{
public static void main(String[] args){
Test<Thread> t1 = new Test<Thread>();
Test<String> t2 = new Test<String>(); //CE
}
}
```

Keypoints about bounded types

```
eg: class Test<T implements Runnable>{}//invalid
class Test<T super String>{}//invalid
```

=> We can't define bounded types by using implements and super keyword

=> But implements keyword purpose we can replace with extends keyword.

keyPoints abount BoundedTypes

==========

=> As the type parameter we can use any valid java identifier but it convention to use T always.

```
eg:class Test<T>{}
  class Test<iNeuron>{}
```

=> We can pass any no of type parameters need not be one.

```
eg: class HashMap<K,V>{}
```

```
HashMap<Integer,String> h=new HashMap<Integer,String>();

Which of the following are valid?

class Test <T extends Number&Runnable> {}//valid
    Number -> class
    Runnable-> interface

class Test<T extends Number&Runnable&Comparable> {} //valid
    Number -> class
    Runnable-> interface
    Comparable -> interface



class Test<T extends Number&String> {} //invalid
    // we can't extends more than one class at a time.

class Test<T extends Runnable&Comparable> {} //valid
    Runnable-> interface
    Comparable -> interface

class Test<T extends Runnable&Number> {}//invalid
    Runnable-> interface
    Number -> class
rule: first inherit and the implement so invalid
```

GenericClass

==========

class: Type parameter

Can we apply TypeParameter at MethodLevel?

Ans.Yes, it is possible.

Generic methods and wild-card character (?)

==========

1. methodOne(ArrayList<String>al):

This method is applicable for ArrayList of only String type.

```
methodOne(ArrayList<String> al){
    al.add("sachin");
    al.add("navinreddy");
    al.add("iNeuron");
al.add(new Integer(10));//invalid
}
```

Within the method we can add only String type of objects and null to the List.

```
2.        methodOne(new ArrayList<String>());
          methodOne(new ArrayList<Integer>());
          methodOne(new ArrayList<Runnable>());
                        |
                        |ArrayList<?> l =new ArrayList<String>();
                        |
    methodOne(ArrayList<?> l):
```

We can use this method for ArrayList of any type but within the method we can't add anything to the List except null.

Example:

```
l.add(null);//(valid)
l.add("A");//(invalid)
l.add(10);//(invalid)
```
This method is useful whenever we are performing only read operation.

3. `methodOne(ArrayList<? extends X> al)`

X -> class, we can make a call to method by passing ArrayList of X type or its Child type.

X -> interface, we can make a call to method by passing ArrayList of X type or its Implementation class.

```
methodOne(ArrayList<? extends X> al){ al.add(null);}
```
Best suited only for read operation.

4. methodOne(ArrayList<? super X> al)

X -> class, we can make a call to method by passing ArrayList of X type or its super class

X -> interface, we can make a call to method by passing ArrayList of X type or its super class of implementation class of x.

```
methodOne(ArrayList<? super X> al){
    al.add(X);
    al.add(null);
}
```

Which of the following declarations are allowed?

```
1. ArrayList<String> l1=new ArrayList<String>();//valid
2. ArrayList<?> l2=new ArrayList<String>();//valid
3. ArrayList<?> l3=new ArrayList<Integer>();//valid
4. ArrayList<? extends Number> l4=new ArrayList<Integer>();//valid
5. ArrayList<? extends Number> l5=new ArrayList<String>();//invalid
6. ArrayList<?> l6=new ArrayList<? extends Number>(); //invalid
7. ArrayList<?> l7=new ArrayList<?>(); //invalid
```

More Fun with Wildcards (The Java™ Tutorials > Bonus > Generics) (oracle.com)

Now let's turn to a more realistic example. A java.util.TreeSet<E> represents a tree of elements of type E that are ordered. One way to construct a TreeSet is to pass a Comparator object to the constructor. That comparator will be used to sort the elements of the TreeSet according to a desired ordering.

TreeSet(Comparator<E> c)

The Comparator interface is essentially:

```
interface Comparator<T> {
    int compare(T fst, T snd);
}
```

Suppose we want to create a TreeSet<String> and pass in a suitable comparator, We need to pass it a Comparator that can compare Strings. This can be done by a Comparator<String>, but a Comparator<Object> will do just as well. However, we won't be able to invoke the constructor given above on a Comparator<Object>. We can use a lower bounded wildcard to get the flexibility we want:

TreeSet(Comparator<? **super** E> c)

This code allows any applicable comparator to be used.

As a final example of using lower bounded wildcards, lets look at the method Collections.max(), which returns the maximal element in a collection passed to it as an argument. Now, in order for max() to work, all elements of the collection being passed in must

implement Comparable. Furthermore, they must all be comparable to each other.
A first attempt at genericfying this method signature yields:

```
public static <T extends Comparable<T>> T max(Collection<T> coll)
```

That is, the method takes a collection of some type T that is comparable to itself, and returns an element of that type. However, this code turns out to be too restrictive. To see why, consider a type that is comparable to arbitrary objects:

```
class Foo implements Comparable<Object> {
    ...
}
Collection<Foo> cf = ... ;
Collections.max(cf); // Should work
```

Every element of cf is comparable to every other element in cf, since every such element is a Foo, which is comparable to any object, and in particular to another Foo. However, using the signature above, we find that the call is rejected. The inferred type must be Foo, but Foo does not implement Comparable<Foo>.
It isn't necessary that T be comparable to **exactly** itself. All that's required is that T be comparable to one of its supertypes. This give us:

```
public static <T extends Comparable<? super T>>
        T max(Collection<T> coll)
```

Note that the actual signature of Collections.max() is more involved. We return to it in the next section, Converting Legacy Code to Use Generics. This reasoning applies to almost any usage of Comparable that is intended to work for arbitrary types: You always want to use Comparable<? **super** T>.
In general, if you have an API that only uses a type parameter T as an argument, its uses should take advantage of lower bounded wildcards (? **super** T). Conversely, if the API only returns T, you'll give your clients more flexibility by using upper bounded wildcards (? **extends** T).

From <https://docs.oracle.com/javase/tutorial/extra/generics/morefun.html>

TypeParameter at Method level

==========

```
                    |=> TypeParameter at the class level
class Demo<T>{

                        |=> Type parameter defined just before the return type
        public <T> void m1(T t){

        }
}
```

Which of the following declarations are allowed?

```
public <T> void methodOne1(T t){}//valid
public<T extends Number> void methodOne2(T t){}//valid
public<T extends Number&Comparable> void methodOne3(T t){}//valid
public<T extends Number&Comparable&Runnable> void methodOne4(T t){}//valid
public<T extends Number&Thread> void methodOne(T t){}//invalid
public<T extends Runnable&Number> void methodOne(T t){}//invalid
```

```
public<T extends Number&Runnable> void methodOne(T t){}//valid
```

Communication with non generic code

==========

To provide compatibility with old version sun people compramized the concept of generics in very few area's the following is one such area.

Example:

```
class Test{
    public static void main(String[] args){
        ArrayList<String> l=new ArrayList<String>();
        l.add("sachin");
        //l.add(10);//C.E:cannot find symbol,method add(int)

        methodOne(l);
        l.add(10.5);//C.E:cannot find symbol,method add(double)

        System.out.println(l);//[sachin, 10, dhoni, true]
}
public static void methodOne(ArrayList l){
        l.add(10);
        l.add("dhoni");
        l.add(true);
}
```

Conclusions :

Generics concept is applicable only at compile time, at runtime there is no such type of concept.

At the time of compilation, as the last step generics concept is removed,hence for jvm generics syntax won't be available.

Hence the following declarations are equal.

```
ArrayList l=new ArrayList<String>();
ArrayList l=new ArrayList<Integer>();
ArrayList l =new Arraylist<Double>();
```

All are equal at runtime,becoz compiler will remove these generics syntax

ArrayList l=new ArrayList();

Example 1:

```
import  java.util.*;
class Test {
    public static void main(String[] args) {
        ArrayList l=new ArrayList<String>();
        l.add(10);
        l.add(10.5);
        l.add(true);
        System.out.println(l);// [10, 10.5, true]
    }
}
```

Example 2:

```
import java.util.*;
class Test {
    public void methodOne(ArrayList<String> l){}
    public void methodOne(ArrayList<Integer> l){}
}
```

CE: duplicate methods found

Behind the scenes by the compiler

==========

1. Compiler will scan the code
2. Check the argument type
3. if Generics found in the argument type remove the Generics syntax
4. Compiler will again check the syntax

Example3:

The following 2 declarations are equal.

```
ArrayList<String> l1=new ArrayList();
ArrayList<String> l2=new ArrayList<String>();
```

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

```
l1.add("A");//valid
l1.add(10); //invalid
```