Generics(JDK1.5V)
==========
Agenda:
1. Introduction
2. Type-Safety
3. Type-Casting
4. Generic Classes
5. Bounded Types
6. Generic methods and wild card character(?)
7. Conclusions
8. Collection vs Collections

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
                              incompatible type: required java.lang.String   found:
int
```

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].

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

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
```

Generic classes:
Until 1.4v a non-generic version of ArrayList class is declared as follows.

Example:

```
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


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>();//valid
Test <String> t2=new Test < String>();//valid

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
----------------------------------------
 => We can't define bounded types by using implements and super keyword
 => But implements keyword purpose we can replace with extends keyword.
            eg: class Test<T implements Runnable>{}//invalid
```

```
                class Test<T super String>{}//invalid

=> 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(first class and then interface)
class Test<T extends Number&Runnable&Comparable> {} //valid(first class and then
multiple interfaces)
class Test<T extends Number&String> {} //invalid(both are classes becoz multiple
inheritance through class is not supported)
class Test<T extends Runnable&Comparable> {}//valid (both are interfaces)
class Test<T extends Runnable&Number> {}//invalid(first class then interface)


Generic methods with wildcard pattern
=================================
  ? => it is a wild chard symbol to indicate any type i can collect.

methodOne(ArrayList<String> l):
This method is applicable for ArrayList of only String type.
Example:
      l.add("A");
      l.add(null);
      l.add(10);//(invalid)

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

methodOne(ArrayList<? Extends x> l):
If x is a class then this method is applicable for ArrayList of either x type or
its child classes.
If x is an interface then this method is applicable for ArrayList of either x type
or its implementation classes.
In this case also within the method we can't add anything to the List except null.

methodOne(ArrayList<? super x> l):
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.
But within the method we can add x type objects and null to the List.

eg:    Runnable
            |
           Thread<==super class====== Object
```

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(String and Number no relationship)
6. ArrayList<?> l6=new ArrayList<? extends Number>(); //invalid becoz of <? extends Number is right hand side>
7. ArrayList<?> l7=new ArrayList<?>(); //invalid

Declaring type parameter at class level
==========
class Test<T>{
	We can use anywhere this 'T'.
}

Declaring type parameter at method level
==========
We have to declare just before return type.

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(2 classes extends not possible)
public<T extends Runnable&Number> void methodOne(T t){}//invalid(first interface not possible)
public<T extends Number&Runnable> void methodOne(T t){}//valid

Collection vs Collections

Collection(I) => It is a root interface in Collection hierarchy
Collections(C) => It is a utility class(static methods/helper methods would be available)

```
import java.util.*;
public class Test {
	public static void main(String[] args) {
		ArrayList al =new ArrayList();
		al.add(10);
		al.add(5);
		al.add(0);
		al.add(15);
		System.out.println(al);//[10,5,0,15]

		Collections.sort(al);//sorting is done in Ascending order
		System.out.println(al);
	}
}
```

Usage of Compartor will be discussed to work with "Descending order".