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".
```