```
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


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:

```
importjava.util.*;
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
```

Comparable vs Comparator
=======================

```
public TreeSet();
            |=> When we use the above constructor,JVM will internally use
Comparable interface method to sort the Objects
                  based on default natural sorting order.
```

What is Comparable interface?
      It is a functiaonal interace present in java.lang package.
      This interface is internally used by TreeSet object during sorting process of the Object.

```
  @FunctionalInterface
  public interface java.lang.Comparable<T> {
      public abstract int compareTo(T);
  }
```

eg#1.

```
import java.util.*;

class Test
{
      public static void main(String[] args)
      {
            //Sorting of objects will happen based on default natural sorting order
            TreeSet ts = new TreeSet();
```

```
                  ts.add("A");
                  ts.add("Z");
                  ts.add("L");
                  ts.add("B");
                  ts.add(null);//NullPointerException
                  ts.add(10);//ClassCastException

                  System.out.println(ts);//[A,B,L,Z]
      }
}
```

Note:
      If we are keeping the data inside TreeSet object, then the data should be
                  a. Homogenous ====> because it uses compareTo() to sort the
Object
                  b. The object should compulsorily implements an interface called
"Comparable".
                       if we fail to do so , it would result in
"ClassCastException".

eg#2.

```
import java.util.*;
class Test
{
      public static void main(String[] args)
      {
            //Sorting of objects will happen based on default natural sorting order
            TreeSet ts = new TreeSet();

            ts.add(new StringBuffer("A"));
            ts.add(new StringBuffer("Z"));
            ts.add(new StringBuffer("L"));
            ts.add(new StringBuffer("B"));

            System.out.println(ts);//ClassCastException

      }

}
```

note: All Wraper classes and String class has implemented "Comparable" interface.
           StringBuffere class has not implemented Comparable interface, so the
above program would
         result in "ClassCastException".