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
```

Note: To get the underlying object of any reference type we use a method called
ref.getClass().getName()

eg1

```
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

=> 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
```