Generics
========
 The purpose of Generics is
   1. To provide TypeSafety.
   2. To resolve TypeCasting problems.

Case1:
  TypeSafety
      A guranatee can be provided based on the type of elements.
        If our programming requriement is to hold only String type of Objects,we can choose
      String Array.
      By mistake if we are trying to add any another type of Objects, we will get "CompileTimeError".

eg#1.
     String[] s=new String[10000];
     s[0] = "dhoni";
     s[1] = "sachin";
     s[2] = new Integer(10); //CE: incompatible types found : java.lang.Integer
                                                          required: java.lang.String

  W.r.t Arrays we can give guratante that what type of elements is present inside Array, hence
  Arrays are safe to used w.r.t type,so Arrays as TypeSafety.

eg#2.
    ArrayList l=new ArrayList();
     l.add("dhoni");
     l.add("sachin");
     l.add(new Integer(10));
        ...
        ....
     String s1=(String)l.get(0);
     String s2=(String)l.get(1);
     String s3=(String)l.get(2); //RE: ClassCastException

For Collections, we can't give gurantee for the type of elements present inside Collection.
If our pgm requirement is to hold only String type of Objects then if we choose ArrayList by
mistake if we are trying to add any other type of Object,we won't get any CompileTimeError,but
the program may fail at runtime.

Note: Arrays are TypeSafe,Where as Collections are not TypeSafe.
      Arrays provide guarantee for the type of elements we hold, whereas Collections won't provide
      gurantee for the type of elements.

Need of Generics
================
 1. Arrays to use we need to know the size from the begining,but if we don't the size and still
    if we want to provide type casting we need to use "Collections along with Generics".

Case2:

Type casting
============

```
eg#1.
String s[]=new String[3];
   s[0] = "sachin";
 String name=s[0];//Typecasting not required as we it holds only String elements
only.


eg#2.
ArrayList l=new ArrayList();
   l.add("sachin");
 String name=l.get(0);//CE: found : java.lang.Object
                      required: java.lang.String

 String name=(String)l.get(0);
                 |=>TypeCasting is compulsory when we work with Collections.


To Overcome the above mentioned problems of Collections we need to go for Generics
in 1.5V, which
provides TypeSafety and to Resolve TypeCasting problems.

How TypeSafety is provided in Generics and how it resolves the problems of
TypeCasting?

eg#1.
      ArrayList al=new ArrayList();//Non-Generic ArrayList which holds any type of
elements.
      al.add("sachin");
      al.add("dhoni");
      al.add(new Integer(10));
        al.add("yuvi");

eg#2.
      ArrayList<String> al=new ArrayList<String>();//Generic ArrayList which holds
only String.
      al.add("sachin");
      al.add("dhoni");
      al.add(new Integer(10));//CE
        al.add("yuvi");

Note: Through Generics TypeSafety is provided.

     ArrayList<String> al=new ArrayList<String>();//Generic ArrayList which holds
only String.
      al.add("sachin");
      al.add("dhoni");
        al.add("yuvi");
      String name=al.get(0); //TypeCasting is not required

At the time of retrieval, we ar not requried to perform TypeCasting.

Note: Through Generics TypeCasting problem is solved.


Difference b/w
  ArrayList l=new ArrayList();
```

1. It is a nongeneric version of ArrayList Object.
2. It wont provide TypeSafety as we can add any elements into the ArrayList.
3. TypeCasting is required when we retrieve elements.

ArrayList<String> l=new ArrayList<String>();
1. It is a generic version of ArrayList Object.
2. It provides TypeSafety as when we can add only String type of Objects.
3. TypeCasting is not required when we retrieve elements.

Conclusion1
==========
|=> parameter type
ArrayList<String> al =new ArrayList<String>();
|=> BaseType

List<String> al =new ArrayList<String>();
Collection<String> al =new ArrayList<String>();

ArrayList<Object> al=new ArrayList<String>();//CE:incompatible type:
found ArrayList<String>
required ArrayList<Object>

Polymorphism is applicable only for the BaseType,but not for Parameter type.
Polymorphism=> usage of parent reference to hold Child object is the concept of "Polymporphism".

Conclusion2
==========
Collection concept is applicable only for Object,it is not applicable for primitive types.
so parameter type should be always be class/interface/enum,if we take primitive it would
raise in "CompileTimeError".

eg#1.
ArrayList<int> al=new ArrayList<int>();//CE: unexpected type: found int
required reference

Generic classes
==========
untill 1.4 version, nongeneric version of ArrayList class is declared as follows

```
class ArrayList{
      boolean add(Object o);//Argument is Object so no typesafety.
      Object get(int index);//return type is Object so type casting is requried.
}
```

In 1.5 Version Generic version class is defined as follows

```
                |=> TypeParameter
class ArrayList<T>{
      boolean add(T t);
      T get(int index);
}
```
T => Based on our runtime requirement, T will be replaced with our provided type.

```
          |
          |
  class ArrayList<String>{
      boolean add(String t);//We can add only String type of Object it provides
TypeSafety
      String get(int index);//Retrieval Object is always of type String, so
TypeCasting not                                        required.
   }

To hold only String type of Object
    ArrayList<String> al =new ArrayList<String>
      al.add("sachin");
      al.add(new Integer(10));//CE: can't find symbol
                                      method: add(java.lang.Integer)
                                      location: class ArrayList<String>
      String name=al.get(0);
      System.out.println(name);

Note:
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.


Normal Classes also we can apply Generics also
==============================================
Based on our requrimenet, we can define our own generic classes also
eg#1.
class Account<T>{

}

Account<Gold> a1=new Account<Gold>();
Account<Platinum> a2=new Account<Platinum>();

eg:
class Gen<T>{
      T obj;
      Gen(T obj){
            this.obj =obj;
      }

      public void show(){
            System.out.println("The type of class is : "+obj.getClass().getName());
      }

      public T getObj(){
            return obj;
      }

}
public class Test {
      public static void main(String[] args) {

            Gen<String> g1=new Gen<String>("sachin");
            g1.show();
            System.out.println(g1.getObj());
            System.out.println();
```

```
            Gen<Integer> g2=new Gen<Integer>(10);
            g2.show();
            System.out.println(g2.getObj());

            System.out.println();

            Gen<Double> g3=new Gen<Double>(10.5);
            g3.show();
            System.out.println(g3.getObj());

      }
}
```

output
The type of class is : java.lang.String
sachin

The type of class is : java.lang.Integer
10

The type of class 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>();
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.

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

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

```
class Test<T extends Number>{}
     Test<Integer> t=new Test<Integer>();//valid
     Test<String>  t=new Test<String>();//CE: Type parameter java.lang.String is
not in the                                boundary.

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

```
     Test<Runnable> t=new Test<Runnable>();//valid
     Test<Thread> t=new Test<Thread>();//valid
     Test<String> t=new Test<String>(); //CE:Type parameter java.lang.String is not
within its
                                   bound.


Note:
class Test<T extends Number>{}      class Test<T extends Runnable>{}
                        |
                  both conditions should be satisified
         class Test<T extends Number & Runanble>{}

As the type parameter we can pass any type which extends Number class and
implements Runnable interface.

class Test<T extends Number & Runnable>{}
class Test<T extends Runnable & Comparable>{}//valid
class Test<T extends Number & Runnable & Comparable>{}//valid
class Test<T extends Runnable & Number>{}//invalid(first class then interface)
class Test<T extends Number & String>{} //invalid(mulitple inheritance is not
supported)
class Test<T extends Number & Thread>{} //invalid(mulitple inheritance is not
supported)


Conclusions
===========
class Test<T extends Number>{}//valid
class Test<T implements Runnable>{}//invalid
class Test<T extends Runnable>{}//valid
class Test<T super String>{}//invalid
      To specify the bounded types we need to use only extends keyword,we can't use
      implements and super keyword,but we can replace implements keyword with
extends
      keyword.

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

eg#1.
class Sample<T extends Number>
{

}

class Test
{
      public static void main(String[] args)
      {
            Sample<Number>  s1 =new Sample<Number>();
            Sample<Integer> s2 =new Sample<Integer>();
            Sample<String>  s3 =new Sample<String>();//CE: String is not bounded
within T type
      }
}

eg#2.
```

```
class Sample<T extends Runnable>
{

}
class Test
{
	public static void main(String[] args)
	{
		Sample<Runnable> s1 =new Sample<Runnable>();
		Sample<Thread>   s2 =new Sample<Thread>();
		Sample<String>   s3 =new Sample<String>();//String is not within bounds
of type-variable T

	}
}
```

Generic methods and wild-card character (?)

```
==========================================
ArrayList<String> l=new ArrayList<String>();
	m1(l);
	;;;;;
	;;;;;
ArrayList<Integer> l1=new ArrayList<Integer>();
	m1(l1);
	;;;;;;
	;;;;;;
ArrayList<Double> l2=new ArrayList<Double>();
	m1(l2);
	;;;;;;
	;;;;;;
ArrayList<Student> l3=new ArrayList<Student>();
	m1(l3);



public static void m1(ArrayList<String> al){
	;;;;
	;;;;
}
public static void m1(ArrayList<Integer> al){
	;;;;
	;;;;
}
public static void m1(ArrayList<Double> al){
	;;;;
	;;;;
}
public static void m1(ArrayList<Student> al){
	;;;;
	;;;;
}
```

As noticed the length of the code is increased and it increases the burden.
To reduce the burden we can write the code as shown below

```
	public static void m1(ArrayList<?> al){
			;;;;;
			;;;;;
	}
```

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.
            m1(ArrayList<?> l){ System.out.println(l);}

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.
=> This method is useful whenever we are performing only read operation.