Generics:
—————————————
In Java applications, if we represent the same type of elements as a single entity there we are able to get Type Safety and we are able to perform Type safe operations.

In Java applications, if we represent different types of elements as a single entity then we are unable to get Type Safety and we are unable to perform Type Safe operations.

As per the above rule, in Java Arrays are able to provide Type Safety , because Arrays are able to allow only one type of elements and it is giving guarantee for the elements type but Collections are unable to provide Type Safety , because Collections are able to allow different types of elements and it is not giving guarantee for the elements type.

In Collections, the main purpose of Generics is

1. To provide Type Safety in Collections
2. To avoid Type Casting problems while reading elements from Collections.

In Collections, if we want to read an element from Collection object then we have to use get() method, it will return the element in Object type, where we have to perform Typecasting to get elements in their original type.
EX:
```
ArrayList al = new ArrayList();
al.add("AAA");
al.add("BBB");
al.add("CCC");
al.add(10);

String str1 = (String)al.get(0);
String str2 = (String)al.get(1);
String str3 = (String)al.get(2);
String str4 = (String)al.get(3); —> java.lang.ClassCastException
```

To overcome the above Type Safety problem and Type Casting problem we have to use Generics which are introduced by JAVA in its JDK1.5 version.

To provide Generics in Collections we have to use the following Syntax.

```
CollectionType<Type> refVar = new CollectionType<Type>();
```

EX:
ArrayList<String> al = new ArrayList<String>();

Here ArrayList is able to allow only String type elements, if we add any other element then the compiler will raise an error.

```
al.add("AAA");
al.add("BBB");
al.add("CCC");
al.add(new Integer(10)); ----> Error
```

Now we can get elements from the above ArrayList object directly in the form of String without using Type casting.

```
String str1 = al.get(0);
String str2 = al.get(1);
String str3 = al.get(2);
```

EX:
```
ArrayList<Integer> al = new ArrayList<Integer>();
al.add(new Integer(10));
al.add(new Integer(20));
al.add(new Integer(30));
al.add(new String("AAA")); --> Error

Integer in1 = al.get(0);
Integer in2 = al.get(1);
Integer in3 = al.get(2);
```

EX:
```
import java.util.ArrayList;

public class Main {
public static void main(String[] args) {
ArrayList<String> al = new ArrayList<String>();
al.add("AAA");
al.add("BBB");
al.add("CCC");
al.add("DDD");
//al.add(new Integer(10)); ---> Error
System.out.println(al);
String str1 = al.get(0);
String str2 = al.get(1);
String str3 = al.get(2);
String str4 = al.get(3);
System.out.println(str1);
System.out.println(str2);
System.out.println(str3);
System.out.println(str4);
}
}
```

In the case of Collections we are unable to provide primitive data types as Generic Types.

EX:

```
ArrayList<int> al = new ArrayList<int>(); --> Invalid
ArrayList<Integer> al = new ArrayList<Integer>(); —> Valid
```

In the case of Collections, we are unable to provide polymorphism over the generic Types.

EX:

```
ArrayList<Serializable> al = new ArrayList<String>();
```

Compilation Error

```
ArrayList<Object> al = new ArrayList<String>();
```

Compilation Error

Generic Classes:

If we declare a class with Generic parameter Type then that class is called Generic class.

Syntax:

```
class ClassName<Type>{
      ----
}
```

If we want to create an Object for the generic class then we have to use the following syntax.

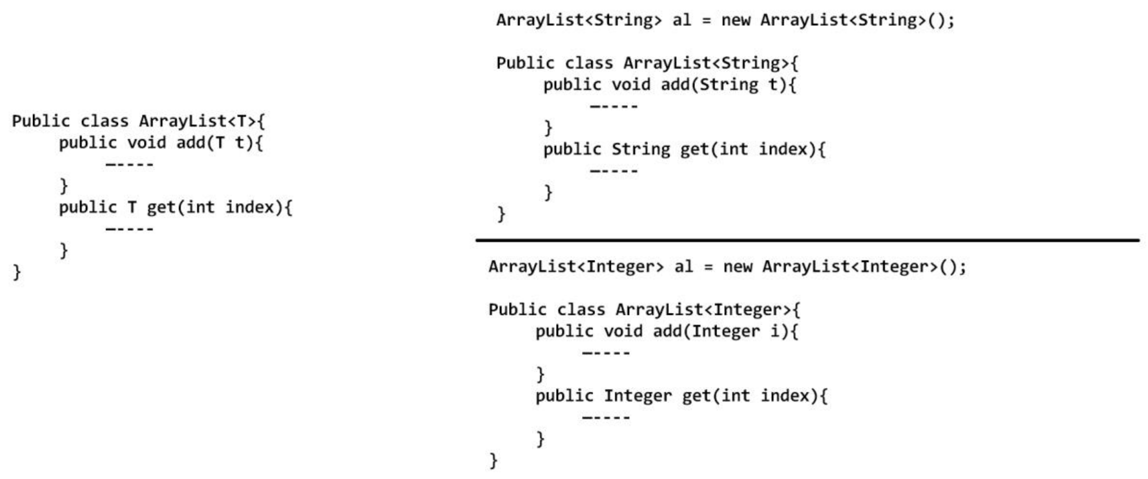
```
ClassName<Type> refVar = new ClassName<Type>();
```

EX:
Before the JDK1.5 version, the ArrayList class was as below.

```
Public class ArrayList{
      public void add(Object obj){
            —----
      }
      public Object get(int index){
            —----
      }
}
```

After JDK1.5 version the ArrayList was as below.

```
Public class ArrayList<T>{
      public void add(T t){
            —----
      }
      public T get(int index){
            —----
      }
}
```



```
Public class ArrayList<T>{
      public void add(T t){
            —----
      }
      public T get(int index){
            —----
      }
}
```

```
ArrayList<String> al = new ArrayList<String>();

Public class ArrayList<String>{
      public void add(String t){
            —----
      }
      public String get(int index){
            —----
      }
}
```

```
ArrayList<Integer> al = new ArrayList<Integer>();

Public class ArrayList<Integer>{
      public void add(Integer i){
            —----
      }
      public Integer get(int index){
            —----
      }
}
```

EX:

```
class Account<T>{
      —---
}
```

```
EX:
Account<Gold> acc = new Account<Gold>();
Account<Silver> acc = new Account<Silver>();

EX:
Medal<Gold> medal = new Medal<Gold>();
Medal<Silver> medal = new Medal<Silver>();
```

EX:

```
class Test<T>{
T t;
Test(T t){
this.t = t;
}
public void show(){
System.out.println("The Type of Object is :
"+t.getClass().getName());
}
public T getObject(){
return t;
}
}
public class Main {
public static void main(String[] args) {
Test<Integer> test1 = new Test<Integer>(10);
test1.show();
System.out.println(test1.getObject());

Test<String> test2 = new Test<String>("Durgasoft");
test2.show();
System.out.println(test2.getObject());

Test<Double> test3 = new Test<Double>(22.2222);
test3.show();
System.out.println(test3.getObject());

}
}
```

Bounded Types:

If we declare a generic class then we are able to pass any data type as generic type Parameter.

```
class Test<T>{
}

Test<Integer> t1 = new Test<Integer>();
Test<String> t2 = new Test<String>();
```

In the above example, if we want to restrict generic Type parameters then we have to use Bounded Types by using 'extends' keyword.

Syntax:

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

1. Where if X is a class then we are able to pass either X type or its sub types only as type parameters.
2. Where if X is an interface then we are able to pass either X or its implementation classes only.

EX:
```
class Test<T extends Number>{

}
public class Main {
public static void main(String[] args) {
Test<Number> test1 = new Test<Number>();
Test<Integer> test2 = new Test<Integer>();
Test<Float> test3 = new Test<Float>();
//Test<String> test4 = new Test<String>(); ---> Error
}
}
```

EX:
```
import java.io.Serializable;

class Test<T extends Serializable>{

}
```

```
public class Main {
public static void main(String[] args) {

Test<Number> test1 = new Test<Number>();
Test<Integer> test2 = new Test<Integer>();
Test<Float> test3 = new Test<Float>();
Test<String> test4 = new Test<String>();
//Test<Thread> test5 = new Test<Thread>(); --> Error
}
}
```

Note: To declare bounded types in the Generic classes we have to use only extends keyword, it is not possible to use super keyword and implements keywords.

Wild Card Characters in Generic Methods:

If we declare a method with the parameter contains generic Type then that method is a Generic method.
EX:

```
void m1(ArrayList<String> al){
}
```

The above method is able to take an ArrayList as a parameter, it must have only String type elements, it must not have any other type elements.

EX:

```
import java.io.Serializable;
import java.util.ArrayList;

class Bank{
public void getCustomersDetails(ArrayList<String> list){

System.out.println(list);
}
}

public class Main {
public static void main(String[] args) {
Bank bank = new Bank();
ArrayList<String> al = new ArrayList<String>();
al.add("AAA");
al.add("BBB");
al.add("CCC");
bank.getCustomersDetails(al);

ArrayList<Integer> al1 = new ArrayList<Integer>();
al1.add(10);
al1.add(20);
al1.add(30);
//bank.getCustomersDetails(al1); --> Error


}
}
```

EX:

```
void m1(ArrayList<Integer> al){
}
```

The above method is able to take an ArrayList as a parameter, it must have only Integer type elements, it must not have any other type elements.

EX:

```
import java.io.Serializable;
import java.util.ArrayList;

class Bank{
public void getCustomersDetails(ArrayList<Integer> list){

System.out.println(list);
}
}

public class Main {
public static void main(String[] args) {
Bank bank = new Bank();
ArrayList<String> al = new ArrayList<String>();
al.add("AAA");
al.add("BBB");
al.add("CCC");
//bank.getCustomersDetails(al); --> error

ArrayList<Integer> al1 = new ArrayList<Integer>();
al1.add(10);
all.add(20);
all.add(30);
bank.getCustomersDetails(all);


}
}
```

In the Generic Methods,if we want to pass a parameter having Generic Type and it must have any type of elements then we have to use Wildcard character.

Syntax:
```
void m1(ArrayList<?> al){
}
```

Inside the method we must not add any other element except null.

EX:
```
import java.io.Serializable;
import java.util.ArrayList;

class Bank{
public void getCustomersDetails(ArrayList<?> list){

System.out.println(list);
}
}

public class Main {
public static void main(String[] args) {
Bank bank = new Bank();
ArrayList<String> al = new ArrayList<String>();
al.add("AAA");
al.add("BBB");
al.add("CCC");
bank.getCustomersDetails(al);

ArrayList<Integer> all = new ArrayList<Integer>();
all.add(10);
all.add(20);
all.add(30);
bank.getCustomersDetails(all);
}
}
```

```
In the generic methods we are able to provide boundaries to the wildcard
characters by using extends and super keywords.

void m1(ArrayList<? extends X> al){
}

If X is a class then the ArrayList is able to add either X type elements or
its subtype elements.

If X is an interface then this method is applicable for ArrayList of either X
type element or its implementation class types.

EX:
```

```java
import java.io.Serializable;
import java.util.ArrayList;

class Bank{
public void getDetails(ArrayList<? extends Number> list){

System.out.println(list);
}
}

public class Main {
public static void main(String[] args) {
Bank bank = new Bank();

ArrayList<Integer> al1 = new ArrayList<Integer>();
al1.add(10);
al1.add(20);
al1.add(30);
bank.getDetails(al1);

ArrayList<Float> al2 = new ArrayList<Float>();
al2.add(22.22f);
al2.add(33.33f);
al2.add(44.44f);
bank.getDetails(al2);
ArrayList<String> al3 = new ArrayList<String>();
al3.add("AAA");
al3.add("BBB");
al3.add("CCC");
//bank.getDetails(al3); --> Error


}
}
```

EX:

```
import java.io.Serializable;
import java.util.ArrayList;

class Bank{
public void getDetails(ArrayList<? extends Serializable> list){

System.out.println(list);
}
}

public class Main {
public static void main(String[] args) {
Bank bank = new Bank();

ArrayList<Integer> al1 = new ArrayList<Integer>();
al1.add(10);
al1.add(20);
al1.add(30);
bank.getDetails(al1);

ArrayList<Float> al2 = new ArrayList<Float>();
al2.add(22.22f);
al2.add(33.33f);
al2.add(44.44f);
bank.getDetails(al2);

ArrayList<String> al3 = new ArrayList<String>();
al3.add("AAA");
al3.add("BBB");
al3.add("CCC");
bank.getDetails(al3);
}
}
```

```
void m1(ArrayList<? super X>){
}
```

If X is a class then it is applicable for the ArrayList of either X type or its Super type elements.

If X is an interface then this method is applicable for ArrayList of either X type or super classes of the X implementation class.

EX:

```
import java.io.Serializable;
import java.util.ArrayList;

class Bank{
public void getDetails(ArrayList<? super Integer> list){

System.out.println(list);
}
}

public class Main {
public static void main(String[] args) {
Bank bank = new Bank();

ArrayList<Integer> al1 = new ArrayList<Integer>();
al1.add(10);
al1.add(20);
al1.add(30);
bank.getDetails(al1);

ArrayList<Number> al2 = new ArrayList<Number>();
al2.add(22.22f);
al2.add(33.33f);
al2.add(44.44f);
bank.getDetails(al2);

ArrayList<Float> al3 = new ArrayList<Float>();
al3.add(22.22f);
al3.add(33.33f);
al3.add(44.44f);
//bank.getDetails(al3); --> Error


}
}
```

EX:

```
import java.io.Serializable;
import java.util.ArrayList;

class Bank{
public void getDetails(ArrayList<? super Runnable> list){

System.out.println(list);
}
}

public class Main {
public static void main(String[] args) {
Bank bank = new Bank();

ArrayList<Runnable> al1 = new ArrayList<Runnable>();
bank.getDetails(al1);

ArrayList<Object> al2 = new ArrayList<Object>();
bank.getDetails(al2);

}
}
```

Q)Find the valid Syntaxes from the following generic Declarations?
----

1. ArrayList<String> al = new ArrayList<String>();// Valid
2. ArrayList<?> al = new ArrayList<String>();//Valid
3. ArrayList<?> al = new ArrayList<Integer>();//Valid
4. ArrayList<? extends Number> al = new ArrayList<Integer>();// Valid
5. ArrayList<? extends Number> al = new ArrayList<String>();// Invalid
6. ArrayList<?> al = new ArrayList<? extends Number>();// Invalid
7. ArrayList<?> al = new ArrayList<?>();// Invalid
8. Map<String, String> map = new HashMap<String, String>();//Valid
9. Map<String, Number> map = new HashMap<String, Integer>();// Invalid
10. Map<?,?> map = new HashMap<String, String>();// Valid

```
11.     Map<?,?> map = new HashMap<Integer, Integer>();// Valid
12.     class Test<T extends Number>{  } // Valid
13.     class Test<T extends Number, String>{} // Valid
14.     class Test<T extends Number, Runnable>{} // Valid
15.     Class Test{ void m1(ArrayList<String> al){}   }// Valid
16.     Class Test{ void m1(ArrayList<?> al){}   }// Valid
17.     Class Test{ void m1(ArrayList<? extends Number> al){}   }// Valid
18.     Class Test{ void m1(ArrayList<? extends Number, String>
    al){}}//Invalid
19.     Class Test{ void m1(ArrayList<? extends Number, Runnable>
    al){}}//Invalid
20.     Class Test{ void m1(ArrayList<? super Integer> al){}}//valid
21.     Class Test{ void m1(ArrayList<? super Integer,String>
   al){}}//invalid
22.     Class Test{ void m1(ArrayList<? super Integer,Runnable>
   al){}}//Invalid
```