Generics :

What is the need of Generics ?

As we know our compiler is known for Strict type checking because java is a statically typed checked language.

Before Generic, The basic problem with collection is, It can hold any kind of Object.

```
ArrayList al = new ArrayList(); //raw type
al.add("Ravi");
al.add("Aswin");
al.add("Rahul");
al.add("Raj");
al.add("Samir");

for(int i =0; i<al.size(); i++)
{
  String s = (String) al.get(i);
  System.out.println(s);
}
```

By looking the above code it is clear that Collection stores everything in the form of Object so here even after adding String type only we need to provide casting as shown below.

```
import java.util.*;
class Test
{
     public static void main(String[] args)
     {
       ArrayList al = new ArrayList();
          al.add(12);
          al.add(15);
          al.add(18);
          al.add(22);
          al.add(24);

          for (int i=0; i<al.size(); i++)
          {
            Integer x = (Integer) al.get(i);
                System.out.println(x);
          }

    }

}
```

Note : Even we are accepting only Integer type of Object but still type casting is required.

```
import java.util.*;
class Test2
{
     public static void main(String[] args)
     {
          ArrayList t = new ArrayList(); //raw type
          t.add("alpha");
          t.add("beta");
          for (int i = 0; i < t.size(); i++)
          {
            String str =(String) t.get(i);
            System.out.println(str);
          }

           t.add(1234);
           t.add(1256);

            for (int i = 0; i < t.size(); ++i)
            {
                String obj= (String)t.get(i); //we can't perform type casting here
                System.out.println(obj);
            }
     }
}
```

Even after type casting there is no guarantee that the things which are coming from ArrayList Object is String only because we can add anything in the Collection as a result java.lang.ClassCastException
----
To avoid all the above said problem Generics came into picture from JDK 1.5 onwards

-> It deals with type safe Object so there is a guarantee of both the end i.e putting inside and getting outside.

Example:-
```
ArrayList<String > al = new ArrayList<>();
```

Now here we have a guarantee that only String Object can be inserted as well as only String will come out from the Collection so we can perform String related operation.

Advantages of Generics :
----
1) Type Safe Object (No Compilation warning)
2) No need of type casting
3) Strict compile time checking. (*Type Erasure)

```
import java.util.*;
public class Test3
{
public static void main(String[] args)
{
          ArrayList<String> al = new ArrayList<>();  //Generic type
          al.add("Ravi");
          al.add("Ajay");
          al.add("Vijay");

       for(int i=0; i<al.size(); i++)
          {
          String name = al.get(i); //no type casting is required
          System.out.println(name.toUpperCase());
          }
   }
}

//Program that describes the return type of any method
//can be type safe //[We can apply generics on method return type]

import java.util.*;
public class Test4
{
     public static void main(String [] args)
     {
          Dog d1 = new Dog();
          Dog d2 = d1.getDogList().get(0);
          System.out.println(d2);
     }
}
class Dog
{
     public List<Dog> getDogList()
     {
          ArrayList<Dog> d = new ArrayList<>();
       d.add(new Dog());
          d.add(new Dog());
          d.add(new Dog());
          return d;
     }
}
```

```
//Mixing generic with non-generic
import java.util.*;
class Car
{
}
public class Test5
{
     public static void main(String [] args)
     {
     ArrayList<Car> a = new ArrayList<>();
     a.add(new Car());
     a.add(new Car());
    a.add(new Car());

     ArrayList b = a;  //assigning Generic to raw type

    System.out.println(b);
     }
}
```

```
//Mixing generic to non-generic
import java.util.*;
public class Test6
{
     public static void main(String[] args)
     {
          List<Integer> myList = new ArrayList<>();
          myList.add(4);
          myList.add(6);
          myList.add(5);

          UnknownClass u = new UnknownClass();
          int total = u.addValues(myList);
          System.out.println("The sum of Integer Object is :"+total);
     }
}
class UnknownClass
{
     public int addValues(List list)  //generic to raw type OR
     {

     Iterator it = list.iterator();
     int total = 0;
     while (it.hasNext())
     {
          int i = ((Integer)it.next());
          total += i;                              //total =  15
     }
     return total;
     }
}
```

Note :-
In the above program the compiler will not generate any warning message because even though we are assigning type safe Integer Object to unsafe or raw type List Object but this List Object is not inserting anything new in the collection so there is no risk to the caller.

```
//Mixing generic to non-generic
import java.util.*;
public class Test7
{
     public static void main(String[] args)
     {
          List<Integer> myList = new ArrayList<>();
          myList.add(4);
          myList.add(6);
          UnknownClass u = new UnknownClass();
          int total = u.addValues(myList);
          System.out.println(total);
     }
}
class UnknownClass
{
    public int addValues(List list)  //Generic to raw
     {
          list.add(5); //adding object to raw type
          Iterator it = list.iterator();
          int total = 0;
          while (it.hasNext())
          {
          int i = ((Integer)it.next());
          total += i;
          }
          return total;
     }
}
```
----
*Type Erasure

In the above program the compiler will generate warning message because the unsafe List Object is inserting the Integer object 5 so, the type safe Integer object is getting value 5 from unsafe type so there is a problem to the caller method.

By writing ArrayList<Integer> actually JVM does not have any idea that our ArrayList was suppose to hold only Integers.

All the type safe generics information does not exist at runtime. All our generic code is Strictly for compiler.

There is a process done by java compiler called "Type erasure" in which the java compiler converts generic version to non-generic type.

```
List<Integer> myList = new ArrayList<Integer>();
```

At the compilation time it is fine but at runtime for JVM the code becomes

```
List myList = new ArrayList();
```

Note :- GENERIC IS STRICTLY COMPILE TIME PROTECTION.

```
import java.util.*;
public class TypeErasure
{
     public static void main(String[] args)
     {

     }
     public static void m1(List<Integer> list)
     {

     }
     public static void m1(List<String> list)
     {

     }

}
```

Note : The above code will not compile due to type erasure, Java compiler will remove all the type parameter so at runtime JVM does not have any information.