Module 2

The collections and Framework: Collections Overview, Recent Changes to Collections, The Collection Interfaces, The Collection Classes, Accessing a collection Via an Iterator,Storing User Defined Classes in Collections, The Random Access Interface, Working With Maps, Comparators, The Collection Algorithms, Why Generic Collections?, The legacy Classes and Interfaces, Parting Thoughts on Collections.

Introduction to Generics : Generic class with one type/generic parameter (RAW TYPE)

```
class swap<T> { // T is a TYPE PARAMETER
    private T i,j;
    public void assign(T a, T b)  {
        i=a; j=b;
        System.out.println(i.getClass());
    }

    public void swap()   {
        T t=i;
        System.out.println(t.getClass());
        i=j;    //Assignment or binary bits copy can be done irrespective of the type
        j=t;
    }
    public void display()
    {  System.out.println(i+" "+j);   }
```

Introduction to Generics : Generic class with one type/generic parameter (RAW TYPE)

```
public T acess()
    {  return i; }
   public void modify(T a)
    {i=a;}
}
public class Test
{
        public static void main(String[] args) {
        swap<Integer> a = new swap<Integer>();
// Integer is a TYPE ARGUMENT
        a.assign(1,2);
        a.display();
        Integer t = a.acess();          t=t+10;         a.modify(t);
         a.display();
        a.swap();    a.display();
        }      }
```

Introduction to Generics : Generic class with more than one type/generic parameter (RAW TYPE)

```
import java.util.*;

class generic<u, v> {
        u a;
        v b;
        public void assign(u p, v q)
        {
                a=p; b=q;
        }
        public void display()
        {
                System.out.println(a+ " "+b);
        }
}
```

Introduction to Generics : Generic class with more than one type/generic parameter (RAW TYPE)

```
public class Test  {
    public static void main(String[] args) {
        generic<Integer,Float> a = new generic<Integer,Float>();
        a.assign(1, (float)1.2);
        a.display();

        generic<String,Integer> b = new generic<String,Integer>();
        b.assign("tesla",10);
        b.display();

    } }
```

Introduction to Generics - Generic methods - class level

Like generic classes, generic methods can be coded.

```
public class Test  {
        public static <T> void disp(T a)
        { System.out.println(a); }

        public static void main(String[] args) {
        Integer a = 10;                disp(a);
        String p="aaa";                disp(p);                } }
```

For static methods <T> is always placed before the return type of the method.

It indicates that the T identifier is a type parameter, to distinguish it with concrete types.

If the type parameter of a non-static/instance-level generic method is same as the enclosing class, the indicator <T> is not required.

Introduction to Generics - Generic methods - Instance level

```
class disp<T>
{
        public void disp1(T a)
        {  System.out.println(a);  }
}
public class Test  {
        public static void main(String[] args) {
        disp<Object> r = new disp<Object>();
        Integer a = 10;

        r.disp1(a);

        String p="aaa";
        r.disp1(p);
}  }
```

disp is a generic class and it's parameter T is also generic.

Introduction to Generics - Generic types

Generic type in java can be declared using two ways

1. Raw generic type
2. Bounded generic type

Bounded generic type

The types that are used as **type arguments** can be restricted.
Ex: a method that operates on numbers might only want to accept instances of **Number** or its subclasses.

This is where *bounded type parameters* will be useful.

To declare a bounded type parameter, list the type parameter name, followed by the extends keyword, followed by its *upper bound*, which in this example is **Number**.

In this context, **extends** is used in a general sense to mean either "**extends**" (as in classes) or "**implements**" (as in interfaces).

*****
Generic methods allow type parameters to be used to express dependencies among the types of one or more arguments to a method and/or its return type. If there isn't such a dependency, a generic method should not be used.***** (Applies for Raw and Bounded generic type)