Day 14

Boxing & AutoBoxing

- Process of converting state of instance of value type into reference type is called boxing.

```
int number = 10;
String strNumber = String.valueOf( number );
```

```
int number = 10;
String strNumber = Integer.toString(number);
```

```
int number = 10;
Integer i = Integer.valueOf(number);
```

- If boxing is done implicitly then it is called auto-boxing.

```
int number = 10;
Object obj = number; //AutoBoxing
```

UnBoxing & AutoUnBoxing

- Process of converting state of instance of reference type into value type is called unboxing.

```
String str = "125";
int number = Integer.parseInt(str);
```

```
Integer n1 = new Integer(125);
int n2 = n1.intValue();
```

- If unboxing is done implicitly then it is called auto unboxing.

```
Integer n1 = new Integer(125);
int n2 = n1;
```

Generics

- In java, if we want to write generic code then we should use generics.
- Generic Code without generics

```
class Box
{
    private Object object;
    public Object getObject()
    {
        return object;
    }
    public void setObject(Object object)
    {
        this.object = object;
    }
}
```

```
Object obj = new String();//Upcasting : OK
String str = (String)obj;//Downcasting : OK
```

```
Object obj = new Date();//Upcasting : OK
Date dt = (Date)obj;//Downcasting : OK
```

```
Object obj = new Date();//Upcasting : OK
String str = (String)obj;//Downcasting
//ClassCastException
```

```
Box b1 = new Box();
b1.setObject( new Date( 119, 10, 6 ));
String str = (String) b1.getObject();
//Output : ClassCastException
```

- Using java.lang.Object class we can not write type safe generic code. If we want to write typesafe generic code then we should use generics.
- By passing, datatype / type as argument, we can write generic code in java. Hence parameterized type is called generics.
- Generic code using generics:

```
class Box<T> //T -> Type Parameter Name
{
    private T object;
    public T getObject()
    {
        return object;
    }
    public void setObject(T object)
    {
        this.object = object;
    }
}
public class Program
{
    public static void main1(String[] args)
    {
        Box<Date> b1 = new Box<Date>(); //Date -> Type Argument
        b1.setObject(new Date());
        Date date = b1.getObject();
    }
}
```

Commonly use type parameter names:

1. T : Type
2. N : Number
3. E : Element
4. K : Key
5. V : Value
6. U,S : Second Type Parameters

Type Inference:

- An ability of compiler to detect type of argument at compile time and use it as a type argument is called type inference.

```
Box<Date> b1 = new Box<Date>(); //OK
Box<Date> b2 = new Box<>(); //OK
```

Raw Type:

- If we instantiate generic/paramerized type without type argument then parameterized type is called Raw type.

```
Box b1 = new Box(); //Box -> Raw type
//Box<Object> b1 = new Box<>();
```

- If we want to instantiate parameterized type then type argument must be reference type.

```
Box<int> b1 = new Box();    //Not OK
Box<Integer> b1 = new Box();    // OK
```

Need of Wrapper class

1. If we want to convert String into numeric type.
2. If we want to store numeric values inside instance of parameterized type then type argument must be wrapper class

- It is possible to specify multiple type parameters for the class/interface.

```
class HashTable<K,V>
{
    private K key;
    private V value;
    public void put( K key, V value )
    {
        this.key = key;
        this.value = value;
    }
    public K getKey()
    {
        return key;
    }
    public V getValue()
    {
        return value;
    }
}
public class Program
{
    public static void main(String[] args)
    {
        HashTable<Integer,String> ht = new HashTable<>( );
        ht.put(1, "DAC");
        System.out.println("Key :   "+ht.getKey());
        System.out.println("Value   :   "+ht.getValue());

    }
}
```

Why Generics?

- It gives us stronger type checking at compile time. In other words, it helps us to write type safe code.
- It completly eliminates explict type casting
- It helps us to implement generic algorithm and data structure.

Bounded Type Parameter

- If we want to put restriction on type / datatype that can be used as type argument then we must specify bounded type parameter

```
class Box<T extends Number >
{   }

//T extends Number : Bounded type parameter

public class Program
{
    public static void main(String[] args)
    {
        Box<Number> b1 = new Box<>();//OK
        Box<Integer> b2 = new Box<>();//Ok
        Box<Double> b3 = new Box<>();//Ok
        Box<String> b4=new Box<>(); //Not OK
        Box<Date> b5 = new Box<>(); //Not Ok
    }
}
```

- Specifying bounded type parameter is a job of class implementor.

ArrayList

- It is resizable array.
- It is a part of collection framework

```
ArrayList<Integer> list = null;
list = new ArrayList<>();
list.add(10);
list.add(20);
list.add(30);

for( Integer element : list )
{
    System.out.println(element);
}
```

- On the basis of diffrent type argument we can not overload method.

Wild card

- In java, '?' is called wild card, which represent unknown type.
- Types of wild card
    1. Unbounded wild card
    2. Upper bounded wild card
    3. Lower bounded wild card

Unbounded wild card

```
private static void print(ArrayList<?> list)
{
    for( Object element : list )
        System.out.println(element);
}
```

- In above code, list can contain refernce of ArrayList which can contain unknown type of element.

Upper bounded wild card

```
private static void print(
    ArrayList<? extends Number> list)
{
    for( Number element : list )
        System.out.println(element);
}
```

- In above code, list can contain reference of ArrayList, which can contain elements of Number or its sub type.

Lower bounded wild card

```
private static void print(
    ArrayList<? super Integer> list)
{
    for( Object element : list )
        System.out.println(element);
}
```

- In above code, list can contain reference of ArrayList which can contain elements of Integer and its super type.
- In type argument, we can not use inheritance.

```
private static void print(
    ArrayList<Integer> list)
{
    //TODO
}
ArrayList<Integer> intList = Program.getIntegerList( );

Program.print( intList ); //OK
```

```
private static void print(
    ArrayList<Number> list)
{
    //TODO
}
ArrayList<Integer> intList = Program.getIntegerList( );

Program.print( intList ); //Not OK
```

Generic Method

- generic method without generics:

```
public static void print( Object obj )
{
    System.out.println(obj.toString());
}
```

- generic method using generics:

```
public static <T> void print( T obj )
{
    System.out.println(obj.toString());
}
```

- Generic method with bounded type parameter

```
public static <T extends Number>
void print( T obj )
{
    System.out.println(obj.toString());
}
```

Restrictions on generics

- During instantation of parameterized type, type argument must be reference type.
- On the basis of only different type argument, we can not overload method.
- We can not instantiate type parameter

```
public static <T > void print( T obj )
{
    T t = new T(); //Not Ok
    //TODO
}
```

- we can not declare, parameterized type fields static.

```
class Box<T>
{
    private static T object;
}
```

- We can not use instanceof operator with parameterized type.

```
List<Integer> list = new ArrayList<>();
if( list instanceof ArrayList<Integer>)
//Not OK
{   }
```