Day 8

Generics

```
class Pair<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)
        {
                Pair<Integer,String> p = new Pair<>( );
                p.put(1, "Pune");       //p.put(Integer.valueOf(1),
"Pune");
                System.out.println("Key :       "+p.getKey());  //1
                System.out.println("Value       :       "+p.getValue());
        }
}
```

```
- As shown in above code, we can pass multiple type arguments to the
parameterized type.
- By passing, datatype(type) as a argument, we can define generiv type.
Hence parameterized type is called generics.
```

Bounded Type Parameter

```
- If we want to put restriction on data type that can be used as type
argument then we should use bounded type parameter.
- Specifying bounded type parameter is a job of class implementor.
```

```
//N extends Number -> Bounded Type Parameter
class Box<N extends Number >
{
        private N object;
        public N getObject()
        {
                return object;
        }
        public void setObject(N object)
        {
                this.object = object;
        }
}
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
        }
}
```

```
- On the basis of, different type argument, we can not overload method in
java.
- If implementation of method is logically same/equivalent then we should
give same name function. In other words, we should overload method.
```

Wild Card

```
- In generic "?" is called wild card which represents unknown type.
- There are 3 types of wild card
        1. Unbounded wild card
        2. Upper bounded wild card
        3. Lower boubded wild card
```

Unbounded wild card

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

```
}
public static void main(String[] args)
{
        ArrayList<Integer> intList = Program.getIntegerList( );
        Program.printList( intList );   //Ok
        ArrayList<Double> doubleList = Program.getDoubleList( );
        Program.printList( doubleList );        //OK
        ArrayList<String> stringList = Program.getStringList( );
        Program.printList(stringList);  //OK
}
```

- In above code, list can contain reference of ArrayList which can contain any type of element.

Upper bounded wild card

```
private static void printList( ArrayList<? extends Number > list )
{
        for( Number element : list )
                System.out.println(element );
}
public static void main(String[] args)
{
        ArrayList<Integer> intList = Program.getIntegerList( );
        Program.printList( intList );   //OK
        ArrayList<Double> doubleList = Program.getDoubleList( );
        Program.printList( doubleList );        //OK
        ArrayList<String> stringList = Program.getStringList( );
        Program.printList(stringList);  //Not OK
}
```

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

Lower boubded wild card

```
private static void printList( ArrayList<? super Integer > list )
{
        for( Object element : list )
                System.out.println(element );
}
public static void main(String[] args)
{
```

```
        ArrayList<Integer> intList = Program.getIntegerList( );
        Program.printList( intList );   //OK
        ArrayList<Double> doubleList = Program.getDoubleList( );
        Program.printList( doubleList );        //Not OK
        ArrayList<String> stringList = Program.getStringList( );
        Program.printList(stringList);  //Not OK
}
```

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

```
private static void printList( ArrayList<Number > list )
{       }
public static void main(String[] args)
{
        ArrayList<Integer> intList = Program.getIntegerList( );
        Program.printList( intList );//Error : Type mismatch in type argument
}
```

- Consider following code

```
ArrayList<Integer> l1 = new ArrayList<Integer>( );      //OK
List<Integer> l2 = new ArrayList<Integer>( );   //OK

ArrayList<Number> l1 = new ArrayList<Integer>( );       //Not OK
List<Number> l1 = new ArrayList<Integer>( );    //Not OK
```

Generic Method

```
/*private static void printRecord( Object obj )
{
        System.out.println(obj.toString());
}*/

/*private static <T> void printRecord( T obj )
{
        System.out.println(obj.toString());
}*/
private static <N extends Number> void printRecord( N obj )
{
```

```
        System.out.println(obj.toString());
}
public static void main(String[] args)
{
        //Program.printRecord( 'A' );   //Not OK
        Program.printRecord( 10 );      //Ok
        Program.printRecord( 10.5 );//OK
        //Program.printRecord( "SunBeam" );     //Not tOK
        //Program.printRecord( new Date() );    //Not OK
}
```

Limitations of Generics

```
1. During instantiation of parameterized type, primitive/value types are
not allowed.
        - Box<int> b1 = new Box<>();            //int : not allowed
        - Box<Integer> b1 = new Box<>();        //OK
2. On the basis of different type argument, we can not overload method.
3. We can not instantiate type parameter.
        private static <T> void printRecord( T obj )
        {
                T t = new T();  //Error
        }
4. We can not declare Type parameter field static
        class Box<T>
        {
                private static T object;        //Not OK
        }
5. We can not create array of parameterized type.
        ArrayList<Integer>[] arr = new ArrayList<Integer>[ 3 ];
6. We can not use instanceof operator with parameterized type
        List<Integer> list = new ArrayList<Integer>( ); //OK
        if( list instanceof ArrayList<Integer>( ) )//Not OK
        {       }
        else
        {       }
7. Parameterized type can not be used to define, throw or catch exception.
        class MyException<T> extends Throwable //Error
        {       }
```

Fragile Base Class Problem

- If we make chanages in method of super class then it is necessary to recomplie super class as well as all its sub classes. This problem is called fragile base class problem.
- To avoid this problem, we should use interface.

Interface

- Set of rules of rules is called Specification/Standard.
- If we want to define specification for the sub classes then we should use interafce.
- Main purpose of interface is:
	1. To develop/build trust between service provide and service consumer
	2. To minimize vendor dependancy.
- Interface is keyword in java.
- Interface is reference type in java.
- We can not instantiate interface but we can create reference of interface.

```
interface A
{	}
public class Program
{
	public static void main(String[] args)
	{
		A a  = null;
		a = new A( );
	}
}
```

- Interface can contain
	1. Netsted interface
	2. Contstant / Final Field
	3. Abstract Method
	4. Default Method
	5. Static Method.
- We can not define constructor inside interface.
- We can declare field inside interface. Interface fields are implicitly considered as public, static and final.
- We can declare methods inside interface. Interface methods are by default considered as public and abstract.

```
interface A
{
	int number = 10;
	//public static final int number = 10;
	void print( );
	//public abstract void print( );
}
```

Interface and class Syntax:

- Classes : C1, C2, C3
- Interfaces : I1, I2, I3

1. I2 implements I1; //Not OK
2. I2 extends I1; // OK : Interface Inheritance
3. I3 extends I1, I2; // OK : Multiple Interface Inheritance
4. I1 extends C1; //Not OK
5. C1 extends I1; //Not OK
6. C1 implements I1; //OK : Interface Implementation Inheritance
7. C1 implements I1,I2; //OK : Multiple I/F Impl. Inheritance
8. C2 implements C1; //Not OK
9. C2 extends C1; //OK : Implementation Inheritance
10. C3 extends C1, C2; //Not OK : Multiple Impl. Inheritance
11. C2 implements I1 extends C1; //Not OK
12. C2 extends C1 implements I1; // OK

Conclusion:

```
1. Interface can extend one or more than one interfaces. In other words,
java supports "Multiple Interface Inheritance".
2. Class can implement one or more than one interfaces. In other words,
java supports "Multiple Interface Implementation Inheritance".
3. Class can extend only class. In other words, java do not support
"Multiple Implementation Inheritance".
```

Interface Implementation:

```
//Interface Definition
interface A
{
        int number = 10;
        void print( );
}
//Interface implementation
class B implements A    //Error
{       }
```

```
- It us mandatory to override all abstract methods os I/F in sub class
otherwise sub class will be considered as abstract.
```

```
//Interface Definition
interface A
{
        int number = 10;
        void print( );
}

//Interface implementation
class B implements A
{
        @Override
        public void print()
        {
                System.out.println("Number      :       "+A.number);
        }
}
public class Program
{
        public static void main(String[] args)
        {
                //Interface Use
                B b = new B();  //OK. Not Recommended
                b.print();

                A a = new B();  //OK : Recommended --> Upcasting
                a.print();      //DMD

        }
}
```

> When we should use abstract class and interface? - If "is-a" relationship exist between super type & sub type and if we want to provide common/same method design in all the sub classes then super type should be abstract.

```
- Using abstract class, we can group, instances of related type together.
        Shape[] arr = new Shape[ 3 ];
        arr[ 0 ] = new Rectangle();
        arr[ 1 ] = new Circle( );
        arr[ 2 ]  = nre Triangle( );
- Abstract class can exted ony one class(abstract/concrete )
        Abstract classes : A, B, C
        abstract class B extends A;     //OK
        abstract class C extends A,B;   //Not OK
- We can write constructor inside abstract class
- Abstract class may/may not contain abstract method

- If "is-a" relationship doesn't exist between super type & sub type and
if we want to provide common/same method design in all the sub classes
then super type should be interface.
- Using interface, we can group, instances of unrelated type together.
```

```
        Printable[] arr = new Printable[ 3 ];
        arr[ 0 ] = new Complex();
        arr[ 1 ] = new Employee();
        arr[ 2 ] = new Book();
- Interface can extend more than one interfaces
        Interface I1, I2, I3
        interafce I2 extends I1;        //Ok
        interafce I3 extends I1,I2;     //Ok
- We can not write constructor inside interface.
- Interface methods are by default abstract.
```

Commonly Used Interfaces in java:

```
1. java.lang.Cloneable
2. java.lang.Iterable<E>
3. java.util.Iterator<E>
4. java.lang.Comparable<T>
5. java.util.Comparator<T>
6. java.lang.AutoCloseable
7. java.io.Closeable
8. java.io.Serializable
```

Cloneable Implementation

```
- Consider following code:
        Date dt1 = new Date(5,10,2019);
        Date dt2 = dt1; //Shallow Copy of references.
- If we want to create new instance from existing instance of same class
then we should use "clone()" method.
- Shallow Copy in java : Creating instance and then copying state of
instance into another instance is called shallow copy.
- Syntax:
        "protected native Object clone( )throws CNSE"
        CNSE -> CloneNotSupportedException
- Inside clone method, if we want to create shallow copy of current
instance then we should use "super.clone()" method.
- java.lang.Cloneable is marker/tagging interface.
- Without implementing, Cloneable interface, if we try to create new
instance from existing instance then clone( ) method throws
CloneNotSupportedException.
```

Marker Interface

```
- It is also called as tagging interface.
- Empty interface is also called as marker/tagging interface.
- It is used to generate metadata for JVM.
- Example:
        1. java.lang.Cloneable
        2. java.util.EventListener
        3. java.util.RandomAccess
        4. java.io.Serializable
        5. java.rmi.Remote
- Consider following code
        class Date
        {       }
        class Employee
        {
                String name;
                int empid;
                float salary;
                Date joinDate;
        }
        Date jd = new Date(26,12,2006);
        Employee emp1 = new Employee( "Sandeep", 33, 45000, jd );
        Employee emp2 = emp1;
```