Day 9

Iterable

```
* It is a interface declared in java.lang package
* Methods of java.lang.Iterable interface:
        * Abstract Method
                1. java.util.Iterator<T> iterator()
        * Default Method
                1. default Spliterator<T> spliterator()
                2. default void forEach(Consumer<? super T> action);
* Implementing this interface allows an object to be the target of the
"for-each loop" statement. In other words, Using foreach loop we can
traverse elements of array and instance of class that implements Iterable
iterface.
* Return type of "iterator()" method is java.util.Iterator.
```

Iterator

```
* It is a interface declared in java.util pacakage.
* Methods of java.util.Iterator interface:
        * Abstract Methods
                1. boolean hasNext()
                2. E next()
        * Default Methods
                1. default void remove()
                2. default void forEachRemaining(Consumer<? super E>
action)
```

Nested Class

```
* We can define class inside scope of another class. It is called nested
class.
- Example
```

```
class Outer     //Top Level class       --> Outer.class
{
        class Inner     //Nested class  --> Outer$Inner.class
        {
        }
}
```

```
* Access modifier of Top level class can be either package level private
or public only. But we can use any access modifier on nested class.
* By defining nested class, we can achive encapsulation.
* Types of nested class
        1. Non static Nested class / Inner class
        2. Static nested class
```

Non Static Nested class

```
* It is also called inner class.
* If implementation of nested class is depends on implementation of top
level class then we should declare nested class non static.
* ** Note ** : For simplicity, consider non static nested class as a non
static method of a class.
* Instantiation
```

```
class Outer     //Top-Level class
{
        public class Inner      //Non static Nested class or Inner class
        {

        }
}
public class Program
{
        public static void main(String[] args)
        {
                Outer.Inner in = new Outer().new Inner();
        }
        public static void main2(String[] args)
        {
                Outer out = new Outer();
                Outer.Inner in = out.new Inner();
        }
        public static void main1(String[] args)
        {
                Outer out = new Outer();
        }
}
```

```
* Inner class do not contain static members. But if field is final then we
can make it static.
* Using instance, we can access member's of non static nested class(inner
class) inside method of top-level class.
```

```
class Outer
{
        private int num1 = 10;  //Ok
        private static  int num2 = 20;  //OK
        public class Inner
        {
                private int num3 = 30;  //OK
                private static final int num4 = 40;
        }//end of class
        public void print( )
        {
                System.out.println("Num1        :       "+this.num1);
                System.out.println("Num2        :       "+Outer.num2);

                Inner in = new Inner();
                System.out.println("Num3        :       "+in.num3);
                System.out.println("Num4        :       "+Inner.num4);
        }
}
public class Program
{
        public static void main(String[] args)
        {
                Outer out = new Outer();
                out.print();
        }
}
```

* Without instance, we can access all members of top level class inside method of inner class.

```
class Outer
{
        private int num1 = 10;  //Ok
        private static  int num2 = 20;  //OK
        public class Inner
        {
                private int num3 = 30;  //OK
                private static final int num4 = 40;
                public void print( )
                {
                        System.out.println("Num1        :       "+num1);
//OK
                        System.out.println("Num2        :       "+num2);
//OK
                        System.out.println("Num3        :
"+this.num3);
```

```
                        System.out.println("Num4        :
"+Inner.num4);
                }
        }//end of class

}
public class Program
{
        public static void main(String[] args)
        {
                Outer.Inner in = new Outer().new Inner();
                in.print();
        }
}
```

* Inside method of inner class, if we want to access non static members of top level class then we should use "TopLevel Type Name.member name".

```
class Outer
{
        private int num1 = 10;  //Ok
        public class Inner
        {
                private int num1 = 20;  //Ok
                public void print( )
                {
                        int num1 = 30;  //Ok
                        System.out.println("Num1        :
"+Outer.this.num1);     //10
                        System.out.println("Num1        :
"+this.num1);   //20
                        System.out.println("Num1        :       "+num1);
//30

                }
        }
}
public class Program
{
        public static void main(String[] args)
        {
                Outer.Inner in = new Outer().new Inner();
                in.print();
        }
}
```

Static Nested class

* If we declare nested class static, then it is called static nested class.
* We can declate nested class static but we can not declate top level class static.
* If implementation of nested class do not depend on implementation of top level class then we should declare nested class static.
* **Note** : For simplicity, consider static nested class as a static method of class.
* Instantiation:

```
class Outer     // Outer.class
{
        public static class Inner       //Outer$Inner.class
        {

        }
}
public class Program
{
        public static void main(String[] args)
        {
                Outer out = new Outer();

                Outer.Inner in = new Outer.Inner();
        }
}
```

* Static nested class can contain static as well as non static members but non static nested class contains only non static member.
* Using instance, we can access members of static nested class inside method of top level class.

```
class Outer     // Outer.class
{
        private int num1 = 10;
        private static int num2 = 20;
        public static class Inner
        {
                private int num3 = 30;  //OK
                private static int num4 = 40;   //Ok
        }
        public void print( )
        {
                System.out.println("Num1        :       "+this.num1);
                System.out.println("Num2        :       "+Outer.num2);
```

```
                Inner in = new Inner();

                System.out.println("Num3        :       "+in.num3);
                System.out.println("Num4        :       "+Inner.num4);
        }
}
public class Program
{
        public static void main(String[] args)
        {
                Outer out = new Outer();
                out.print();
        }
}
```

* Inside method of static nested class, we can access static members of top level class directly.
* We can not access non static members of top level class inside method of static nested class directly. In this case, we should use instance of top level class.

```
class Outer     // Outer.class
{
        private int num1 = 10;
        private static int num2 = 20;
        public static class Inner
        {
                private int num3 = 30;  //OK
                private static int num4 = 40;   //Ok
                public void print( )
                {
                        Outer out = new Outer();
                        System.out.println("Num1        :
"+out.num1);
                        System.out.println("Num2        :       "+num2);
                        System.out.println("Num3        :
"+this.num3);
                        System.out.println("Num4        :
"+Inner.num4);
                }
        }
}
public class Program
{
        public static void main(String[] args)
        {
                Outer.Inner in = new Outer.Inner();
                in.print();
```

```
        }
}
```

Example

```
class LinkedList implements Iterable<Integer>
{
        public static class Node        //static
        {
                int data;
                Node next;
        }
        Node head;
        Node tail;
        public class LinkedListIterator //Non static
        {
                private Node trav = head;
        }
        Iterator iterator( )
        {       }
}
```

Local Class

```
* We can write class inside method. It is called local class.
* We can create reference and instance of local class outside method.
* Types of local class
        1. Method Local Inner class
        2. Method Local Annonymous Inner class
```

Method Local Inner class

```
* In java, we can not declare local class static hence local non static
class is also called as method local inner class.
```

```
public class Program    //Program.class
{
        public static void main(String[] args)
        {
                //Method Local Inner class
                class Complex   //Program$1Complex
                {
                        private int real = 10;
```

```
                        private int imag = 20;
                        @Override
                        public String toString()
                        {
                                return this.real+"      "+this.imag;
                        }
                };
                Complex c1 = new Complex();
                System.out.println(c1.toString());
        }
}
```

Method Local Annonymous Inner class

```
* In java, we can create instance without reference, It is called
annonymous instance.
* Example:
        Complex c1 = new Complex();
        new Complex();  //Annonymous instance.
* In java, we can write class without name. It is called annonymous class.
We can write annonymous class inside method only. Hence annonymous class
is also called as method local annonymous inner class.
* In java we can not define annonymous class independantly. To define
anonymous class we should take help of exisiting concrete class, abstract
class or interface.
```

Method Local Anonymous Inner class using Concrete class

```
public static void main(String[] args)
{
        //Person p;     //p is reference / object reference
        //new Person(); //Anonymous instance
        //Person p = new Person();      //Instance with reference
        Person  p = new Person() //Program$1.class
        {
                private int empid;
                private float salary;
                @Override
                public void printRecord()
                {
                        super.printRecord();
                        System.out.println("Empid       :
"+this.empid);
                        System.out.println("Salary      :
"+this.salary);
                }
        };
```

```
        p.printRecord();//DMD
}
```

Method Local Anonymous Inner class using abstract class

```
abstract class Shape
{
        protected float area;
        public abstract void calculateArea();
        public float getArea()
        {
                return area;
        }
}
public class Program
{
        public static void main(String[] args)
        {
                Shape sh = new Shape()
                {
                        private float radius = 10;
                        @Override
                        public void calculateArea()
                        {
                                this.area = (float) (Math.PI *
Math.pow(this.radius, 2));
                        }
                };
                sh.calculateArea();
                System.out.println("Area        :       "+sh.getArea());
        }
}
```

Method Local Anonymous Inner class using interface

```
interface Printable
{
        void print( );
}
public class Program
{
        public static void main(String[] args)
        {
                Printable p = new Printable()
                {
                        @Override
                        public void print()
                        {
```

```
                                System.out.println("Inside print");
                        }
                };
                p.print();
        }
}
```

Functional Interface

```
* An interface, which contains only one abstract method is called
functional interface.
* It can contain multiple static and default methods.
* @Functional Innterface is annotation that is designed to ensure that
wheather interface is functional or not.
```

```
@FunctionalInterface
interface A     //Ok
{
        void f1( );
}
```

```
* Abstract method defined in functional interface is called method
descriptor.
* java.util.function package contains all standard functional interfaces.
        1. Predicate<T>
                - boolean test(T t)
        2. Consumer<T>
                - void accept(T t)
        3. Supplier<T>
                - T get()
        4. Function<T,R>
                - R apply(T t)
        5. UnaryOperator<T>
* To implement functional interface we should use either lambda expression
or method reference.
```

lambda Expression

```
* "->" operator is called lambda operator in java.
* If expression contains lambda operator then it is called lambda
expression.
* lambda operator divides expression into two parts:
        1. Input parameters
```

```
        2. Lambda body
* Syntax:
        (I/P Params. ) -> Lambda Body;
* Lambda body may or may not contain multiple statements. If lambda body
contains multiple statements then it is mandatory to provide curly braces.
* To define lambda expression, we need to take help of method descriptor.
* Lambda expression is also called as anonymous method.
```

```
@FunctionalInterface
interface Printable
{
        void print( );
}
class Program
{
        public static void main(String[] args)
        {
                Printable p = ( )-> System.out.println("Hello LE");
                p.print();
        }
}
```

```
@FunctionalInterface
interface Math
{
        void sum( int num1, int num2 );
}

public class Program
{
        public static void main1(String[] args)
        {
                Math m = ( int num1, int num2 )->System.out.println("Sum
:       "+( num1 + num2));
                m.sum(10, 20);
        }
        public static void main2(String[] args)
        {
                Math m = ( int a, int b )->System.out.println("Sum      :
"+( a + b));
                m.sum(10, 20);
        }
        public static void main(String[] args)
        {
                Math m = (  num1,  num2 )->System.out.println("Sum      :
"+( num1 + num2));
                m.sum(10, 20);
        }
}
```

```
@FunctionalInterface
interface Math
{
        int square( int number );
}

public class Program
{
        public static void main(String[] args)
        {
                Math m = number -> number * number;
                int result = m.square(5);
                System.out.println("Result      :       "+result);
        }
        public static void main2(String[] args)
        {
                Math m = ( number )-> number * number;
                int result = m.square(5);
                System.out.println("Result      :       "+result);
        }
        public static void main1(String[] args)
        {
                Math m = ( int number )-> number * number;
                int result = m.square(5);
                System.out.println("Result      :       "+result);
        }
}
```

```
@FunctionalInterface
interface Math
{
        int factorial( int number );
}
public class Program
{
        public static void main(String[] args)
        {
                Math m =  number ->
                {
                        int result = 1;
                        for( int count = 1; count <= number; ++ count )
                                result = result * count;
                        return result;
                };
                int result = m.factorial(5);
                System.out.println("Result      :       "+result);
        }
}
```

Method reference

```
- It is a java language feature that is used to override method
descriptor.
- We can use it as a alternative to the lambda expression.
```

```
@FunctionalInterface
interface Printable
{
        void print( );
}
public class Program
{
        public static void showRecord( )
        {
                System.out.println("Inside Program.showRecord");
        }
        public  void displayRecord( )
        {
                System.out.println("Inside Program.displayRecord");
        }
        public static void main(String[] args)
        {
                Printable p = Program::showRecord;
                p.print();

                Program prog = new Program();
                Printable p1 = prog::displayRecord;
                p1.print();
        }
        public static void main1(String[] args)
        {
                Printable p = ( )->System.out.println("Hello");
                p.print();
        }
}
```

```
- Compiler do not generate .class file for lambda expression and method
reference.
```

String Handling

```
- String is not a built in type in java. It is a final class declared in
java.lang package. Hence it is considered as refernce type.
```

- It is sub class of Object which extends 3 interfaces:
    1. CharSequence
    2. Comparable
    3. Serializable
- Serializable is marker interace
- "int compareTo(T other)" is a method of java.lang.Comparable interface
- Following are methods of java.lang.CharSequence interface
    1. char charAt(int index)
    2. int length()
    3. CharSequence subSequence(int start, int end)
    4. default IntStream chars()
    5. default IntStream codePoints()
- In java, String is collection of character object that do not ends with '/0' character.
- String str = "SunBeam";
char ch = str.charAt(7); //StringIndexOutOfBoundsException
- Even though, String is reference type, we can create its instance with and without new operator.

```
public static void main(String[] args)
{
	String s1 = new String("Pune");		//OK
	//new String("Pune"); --> String instance

	String s2 = "Pune";	//OK
	//"Pune" --> String literal
}
```

- String instance get space on heap section whereas String literal get space on String literal pool / constant pool.
- String objects are constant/immutable. i.e if we try to modify state of string then jvm create new instance of string.
- if we want to append string to the another string then we should use "String concat(String str)" method but if we want to append state of any value/reference type then we should use + operator.

```
String str = "Sunbeam";
str = str.concat("Pune");	//Ok

String msg = "Hello";
msg = msg + 123;	//OK

String date = "Todays Date	:	";
date = date + new Date();
```