Day 5

Constructor

- If we want to initialize instance of a class then we should constructor.
- It looks like method but it is not considered as a method of a class.
- Constructor do not create instance of a class rather it is used to initialize instance of the class.
- We can use any access modifier on constructor. If constructor is public then we create instance of a class inside method of same class as well as different class. If constructor is private then we can create instance of a class inside method of same class only.
- Due to following reasons, constructor is considered as a special syntax of Java:
    1. Its name is always same as class name.
    2. It doesn't have any return type.
    3. It is designed to call implicitly.
    4. If is deisgned to call once per instance.
- JVM do not call constructor on reference. Constructor gets called on instance only.
- Constructor calling seqeuence depends on order instance declaration.
- Types of constructor:
    1. Parameterless constructor / Zero argument constructor / User Defined Default constructor
    2. Parameterized constructor
    3. Default constructor

Parameterless constructor

- Consider following syntax:

```
class Complex{
    private int real;
    private int imag;
    public Complex( ){  //Parameterless constructor
        this.real = 10;
        this.imag = 20;
    }
}
```

- If we define a constructor W/O parameter then it is called parameterless constructor.
- If we create instance W/O passing aargument then parameterless constructor gets called.

```
Complex c1 = new Complex( ); //Here on instance, parameterless constructor
gets called.
```

- parameterless constructor is also called as zero argument constructor/user defined constructor.

Parameterized constructor

- Consider following syntax:

```
class Complex{
    private int real;
    private int imag;
    //Parameterized constructor
    public Complex( int value ){    //ctor with single parameter
        this.real = value;
        this.imag = value;
    }
    //Parameterized constructor
    public Complex( int real, int imag ){//ctor with two parameter
        this.real = real;
        this.imag = imag;
    }
}
```

- A constructor, which can take parameter is called Parameterized constructor.

```
Complex c1 = new Complex( 10, 20 ); //Here on instance, parameterized
constructor will call.
Complex c2 = new Complex( 30 ); //Here on instance, parameterized
constructor will call.
```

- If we create instance, by passing arguments then parameterized constructor gets called.
- Process of defining more than one constructor inside class is called constructor overloading. In other words, we can overload constructor in Java.
- Code reusability:
    1. We can reduce development time.
    2. We can reduce development cost.
    3. We can reduce developer efforts.
- In Java, we can not call constructor on instance explicitly.

```
Complex c1 = new Complex( );
c1.Complex( );  //Not OK
```

- But we can call constructor from another constructor. It is called as constructor chaining.
- For constructor chaining,we should use "this statement".
- "this statement" must be first statement inside constructor body.

```
class Complex{
    private int real;
    private int imag;
    public Complex() {  //Parameterless constructor
        this( 10, 20 ); //Constructor Chaining
```

```
    }
    public Complex(int value) { //parameterized constructor
        this( value, value );//Constructor Chaining
    }
    public Complex(int real, int imag) {    ////parameterized constructor
        this.real = real;
        this.imag = imag;
    }
}
```

Default constructor

- Compiler generated parameterless constructor is called default constructor.
- If we do not define constructor inside class, then compiler generates one constructor, for the class by default. It is called default constructor.
- Compiler do not generate default parameterized constructor. If we want to create instance by passing argument then we should define parameterized constructor inside class.

```
public class Program {
    public static void main(String[] args) {
        //public Integer(int value)

        //Integer i1 = new Integer( );  //Not OK

        //Integer i2 = new Integer( 123 );  //OK

        int value = 123;
        Integer i2 = new Integer( value );  //OK
    }
}
```

```
public static void main(String[] args) {
    //public Integer(String s) throws NumberFormatException

    //Integer i1 = new Integer( "123" );    //OK

    //String value = "123";
    //Integer i2 = new Integer( value );    //OK

    Integer i3 = new Integer( "1A2B3" );    //NumberFormatException
}
```

final

- If we dont want to modify, value of the variable then we shoild declare it final.
- final is keyword in Java.

```
public static void main1(String[] args) {
    int number = 10;
    number = number + 5;
    System.out.println( number );   //15
}
```

```
public static void main2(String[] args) {
    final int number = 10;  //Intialization
    //number = number + 5;  //Not Ok
    System.out.println( number );   //10
}
```

```
public static void main3(String[] args) {
    final int number;
    number = 10;    //Assignment
    //number = number + 5;  //Not Ok
    System.out.println( number );   //10
}
```

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Number    :   ");
    final int number = sc.nextInt();
    //number = number + 5;  //Not Ok
    System.out.println( number );   //10
}
```

final field

- If we dont want to modify value of any field inside any method of the class including constructor body then we should declare field final.

```
class Test{
    private final int number = 10;  //OK
    public Test() {
        //this.number = 10; //Not OK
    }
    public void showRecord( ) {
        //++ this.number;   //Not OK
        System.out.println("Number  :   "+this.number);
    }
    public void displayRecord( ) {
        //++ this.number;   //Not OK
        System.out.println("Number  :   "+this.number);
```

```
    }
}
public class Program {
    public static void main(String[] args) {
        Test t = new Test( );

        t.showRecord();

        t.displayRecord();
    }
}
```

- We can declare reference final. But we can not declare instance final.

```
public class Program {
    public static void main(String[] args) {
        final Complex c1 = new Complex( 10, 20 );
        c1.setReal(50); //OK
        c1.setImag(60); //OK

        //c1 = new Complex( 11, 22 );   //Error

        System.out.println("Real Number :   "+c1.getReal());    //50
        System.out.println("Imag Number :   "+c1.getImag());    //60
    }
    public static void main1(String[] args) {
        Complex c1 = null;

        c1 = new Complex( 10, 20 );

        c1 = new Complex( 50, 60 );

        System.out.println("Real Number :   "+c1.getReal());
        System.out.println("Imag Number :   "+c1.getImag());
    }
}
```

Static Field

- Non static field is also called as instance variable.
- Instance variable get space once per instance. Instance get space on heap.
- Non static method is also called as instance method.
- Instance method is designed to call on instance.
- If we call, non static method on instance then method get this reference.
- this reference is considered as a connection between non static field and non static method.

- If we want to share, value of any field inside all the instances of same class then we should declare field static.
- static is modifier in java.
- Static field is also called as class level variable.
- class level variable get space during class loading on method area once per class.

```
class A{    //A.class
    private int a;
    private int b;
    private static int z;
}
A a1 = new A();
A a2 = new A();
A a3 = new A();

class B{
    private int p;
    private int q;
    private static int z;
}
B b1 = new B( );
B b2 = new B( );
B b3 = new B( );

class C{
    private int x;
    private int y;
    private static int z;
}
C c1 = new C();
C c2 = new C();
C c3 = new C();
```

- To access non static fields / instance variable we should use object reference(t1/t2/t3/this)
- We can access static fields / class level variable using instance but it is designed to access using class name and dot operator.
- Static field, do not get space inside instance rather all the instances of same class share single copy of it. Hence size of instance depends on size of non static fields declared inside class.

Static Method

- Non static methods are designed to call on instance hence it gets this reference. Hence we can use static as well as non static members inside non static method.
- To access non static members of the class, method should be non static and to access static members of the class method should be static.

```
class Test{
    private int num1;           //0
    private int num2;           //0
    private static int num3;    //0

    public void setNum1(/*Test this,*/ int num1 ) {
        this.num1 = num1;
    }
    public void setNum2(/*Test this,*/ int num2 ) {
        this.num2 = num2;
    }
    public static void setNum3( int num3 ) {
        Test.num3 = num3;
    }
}
```

Why static method do not get this reference.

```
1. If we call, non static method on instance then method get this
reference.
2. Static methods are designed to call on class name.
3. Since static method is not designed to call on instance, it doesn't get
this reference.
```

- Since static method, do not get this reference, we can not access non static member inside static method directly.

```
public static void setNum3( int num3 ) {
    //this.num1 = num3; //Cannot use this in a static context
    //this.num2 = num3; //Cannot use this in a static context
    Test.num3 = num3;
}
```

- In other words, static method can access only static members of the class directly.
- Static method do not get this reference but we can create instance inside static method.
- Using instance, we can access non static members inside static method.

```
public class Program {
    public int num1 = 10;   //OK
    public static int num2 = 20;    //OK
    public static void main(String[] args) {
        //System.out.println("Num1  :   "+num1);    //Not OK
        Program p = new Program();
        System.out.println("Num1    :   "+p.num1);  //Not OK
        System.out.println("Num2    :   "+num2);    //OK : 20
```

```
    }
}
```

- Inside a method, if there is a need to use this reference then method should be non static otherwise, method should be static.

```
class Calculator{
    public static double power( double base, int index ) {
        double result = 1;
        for( int count = 1; count <= index; ++ count )
            result = result * base;
        return result;
    }
}
```

Instance Counter

```
class InstanceCounter{
    private static int count;
    public InstanceCounter() {
        ++ InstanceCounter.count;
    }
    public static int getCount() {
        return InstanceCounter.count;
    }
}
public class Program {
    public static void main(String[] args) {
        InstanceCounter c1 = new InstanceCounter();
        InstanceCounter c2 = new InstanceCounter();
        InstanceCounter c3 = new InstanceCounter();
        System.out.println("Count   :   "+InstanceCounter.getCount());
    }
}
```

- Instance Members = { Instance variable + Instance method}
- Class Level Members = { Class Level Field + Class Level Method }
- In Java, class level variables must exist inside class scope hence we can not declare local variable static.

```
public class Program {
    public static void print( ) {
        static int number = 0;      //Not OK
        number = number + 1;
        System.out.println("Number  :   "+number);
    }
    public static void main(String[] args) {
```

```
        Program.print();    //1
        Program.print();    //1
        Program.print();    //1
    }
}
```

```
public class Program {
    private static int number = 0;
    public static void print( ) {
        number = number + 1;
        System.out.println("Number  :   "+number);
    }
    public static void main(String[] args) {
        Program.print();    //1
        Program.print();    //2
        Program.print();    //3
    }
}
```