3 pillars of oops
==========
1. Encapsulation => It speaks about security
2. Inheritance   => It speaks about reusability
3. Polymorphism => It speaks about flexibility

Datahiding
==========
   Our internal data should not go to the outside world directly that is outside person should not access our interal data directly.
   By using private modifiers we can implement "datahiding".

```
eg:: class Account{
      private double balance;
     }
```
Advantage of Datahiding is security.
Recommended modifier for data members is private.

Abstraction
==========
 Hiding internal implementation and exposing the set of services is called "abstraction".
 In java to promote abstraction we use "abstract class"/"interface".

eg::
  By using ATM,GUI screen bank people are highliting set of services what they are offering without highlighting internal implementation.

Advantages of Abstraction
==========
 a. We can achieve security as we are not highlighting internal implementation.
 b. Enhancement becomes very easy becoz without effecting end user we can able to perform any type
    of changes in our internal system.
 c. It provides more flexibility to the end user to use system easily.
 d. It improves maintainability of the application.
 e. It improves modularity of the application.
 f. It improves easyness to user our system.

By using GUI screens we are implementing "abstraction".

Encapsulation
==========
  Binding of data and corresponding methods into single unit is called "Encapsulation".
  If any java class follows datahiding and abstraction then such class is refered as "Encapsulated class".

Encapsulation = Datahiding + abstraction.

   Every datatmember inside the class should be declared as private, and to access this private
   data we need to have setter and getter methods.

Advantages of Encapsulation
==========
  a. We can achieve security.
  b. Enhancment becomes easy.
  c. Maintainablity and modularisation becomes easy.
  d. It provides flexibility to the user to use system very easily.

Tightly encapsulated class

==========
A class is said to be tightly encapsulated if and only if every variable of that class declared as private,and whether for all variables we have setter and getter method or not and whether these methods are delcared as public or not.

```
eg:: class Account{
          private double balance;
          public void setBalance(double balance){
            this.balance=balance;
          }
        public double getBalance(){
            return balance;
        }
     }
```

Note:: If the parent class is not encapsulated, then all its child classes are not tightly encapsulated.

JavaBean
========
It is a simple java class with private properties and public getter and setter methods.

example
```
public class Student{
     private Integer sid;
     public void setSid(Integer sid){this.sid=sid;}
     public Integer getSid(){return sid;}
}
```

Syntax for setter method
- a. compulsoriy the method name should start with set.
- b. it should be public.
- c. return type should be void.
- d. compulsorily it should have some argument.

Syntax for getter method
- a. compulsoriy the method name should start with get.
- b. it should be public.
- c. return type should not be void.
- d. compulsorily it should not have any argument.

Note:: if the property is of type boolean then for getter method we can prefix with either
"is/get".

```
eg:: public class Student{
              private boolean married;
              public void setMarried(boolean married){
                     this.married=married;
              }

             public boolean isMarried()/getMarried(){
                  return married;
              }
     }
```

Constructors
============
 =>Object creation is not enough, compulsorily we should perform initialization then only the
   object is in a position to provide the response properly.
 => Whenever we are creating an object some piece of the code will be executed automatically to
    perform initialization of an object this piece of code is nothing but constructor.
 => Main objective of constructor is nothing but initialisation of Object.

```
eg:: class Student{
      private String sname;
      private String sage;

      Student(String sname,String sage){
            this.sname=sname;
            this.sage=sage;
        }
      public void display(){
            System.out.println("sname ::"+sname);
            System.out.println("sage  ::"+sage);
      }
     }
    public class DemoApp{
      public static void main(String... args){
            Student std= new Student("sachin",49);
            std.display();
      }
    }
```

Rules for writing a constructor
==============================
1. Name of the constructor and name of the class must be same.
2. Return type concept not applicable for constructor, even if we provide it won't result
   in compile time error, if we do so then java language will treat this as "normal method".

```
     eg:: class Test{
            void Test(){
                  System.out.println("Hello");// It is not a constructor,it is a method.
            }
          }
```

3. It is not a good practise to take the method name same as that of the classname.
4. The modifiers applicable for constructors are private,public,protected,default.
5. The other modifiers if we use, it would result in compile time error.

```
       eg:: class Test{
            static Test(){
                }
           }
```

      output::CE

Default constructor
==================
 1. For every java class constructor concept is applicable.
 2. If we don't write any constructor, then compiler will generate default constructor.
 3. If we write atleast one constructor then compiler won't generate any default constructor, so
    we say every java class will have compiler generated default constructor or

programmer written
    constructor but not both simultaneously.

Prototype of default constructor
================================
  1. It is always no argument constructor.
  2. The access modifier of the default constructor is same as class modifier.
     [applicable for public and default]
  3. Default constructor contains one line, super(), It is a call to super class constructor.
        eg:: constructor.png

super() vs this()
=================
  1. The first line inside the constructor can be super()/ this().
  2. If we are not writing anything then compiler will generate super();

case1: We have to take super()/this() only in the first line of constructor, if we are writing
       any where else it would result in compile time error.
eg#1::
```
   class Test{
      Test(){
            System.out.println("Constructor");//CE
            super();
      }
   }
```

eg#2::we can either use super()/this() but not simulatenously
```
  class Test{
      Test(){
           super();
           this();//CE
      }
  }
```

eg#3:: we can use super()/this() only inside the constructor otherwise it would result in
       compile time error.

```
   class Test{
      void methodOne(){
            super();
            this();
      }
   }
```

Note::
 super() => It should be first line in the constructor.
            It should be used only in constructor.
            It will take the control to parent class constructor.

 this() => It should be first line in the constructor.
            It should be used only in constructor.
            It will make the call of current class constructor.

Difference b/w super(),this() and super,this?
    super(),this()
1. These are constructor calls
2. These are used to invoke super class and current class constructor directly
3. We should use only inside the constructor that to first line otherwise we
   get compile time error.

super, this
1. These are reserverd words meant while working with object creation
2. These are used to refer to parent class and child class instance members.
3. We can use anywhere(instance area),except static area otherwise we get compile time error.

```
   eg:: class Test{
            public static void main(String... args){
                  System.out.println(super.hashCode());//CE
            }
      }
```

Constructor Overloading
==========

A class can contain more than one constructor and all these constructors have same name they differ only in the type of argument, hence these constructors are considered as

"Overloaded constructor".

```
eg::
class Test {
      Test(double d) {
            System.out.println("double argument constructor");
      }

      Test(int i) {
            this(10.5);
            System.out.println("int argument constructor");
      }

      Test() {
            this(10);
            System.out.println("no argument constructor");
      }
}
public class MainApp {
      public static void main(String[] args) throws Exception {
            Test t1= new Test();//double int no argument constructor
            Test t2= new Test(10);// double int argument constructor
            Test t3= new Test(10.5);//dobule argument constructor
      }
}
```

Note:: Recursive method call is always a run time exception, where as recursive constructor call is compile time error.

```
eg::
class Test {
      Test(double d) {
            this();
            System.out.println("double argument constructor");
      }

      Test(int i) {
            this(10.5);
            System.out.println("int argument constructor");
      }

      Test() {
            this(10);
            System.out.println("no argument constructor");
      }
```

```
}
public class MainApp {
	public static void main(String[] args) throws Exception {
		new Test();
	}
}
```

Recursive functions
==========
A function can be called in 2 ways
    a. Nested call.
    b. Recursive call.

Nested call
 => calling a function inside another function is called nested call.
 => In nested call, there is a calling function which will call another function.(called function)

```
eg:: public static void m1(){
		m2();
     }
     public static void m2(){
		m1();
     }
```

Recursive call
 => calling a function inside same function is called recursive call.
 => In Recursive call, both calling and called function are same.

```
  eg::public static void m1(){
		m1();
      }
```

StackOverFlowError
==========

```
public class MainApp {
	public static void main(String[] args) throws Exception {
		methodOne();
		System.out.println("Hello");
	}
	public static void methodOne() {
		methodTwo();
	}
	public static void methodTwo() {
		methodOne();
	}
}
```

normal method recursive call, so its Exception.

CompilerRecursivecall
==========

```
class Test {
	Test(int i) {
		this();
		System.out.println("int argument constructor");
	}
	Test() {
		this(10);
		System.out.println("no argument constructor");
	}
}
```

output:: Compile time error.

instance block
==========
 => Both instance block and constructor will be executed automatically for every object
    creation but instance block 1st followed by constructor.
 => Main objective of Constructor is to initialize the object.
 => Other than initialization, if we want to perform any activity for every object creation
    we have to define the activity inside the instance block
 => Both the concept have a different purpose, replacing one concept with another concept is
    not possible.
 => Constructor can take arguments, but instance block can't take any arguments hence we can't
    replace constructor concept with instance block.
 => similarly we can't replace instance block with constructor.

```
eg:: class Test{
      private String name;
      private Integer age;

      {
            name="sachin";
            age =49;
        }
      Test(String name,Integer age){
            this.name=name;
            this.age=age;
      }

      public void display(){
            System.out.println("Name is :: "+name);
            System.out.println("Age is  :: "+age);
      }
     }

   Test t= new Test("dhoni",49);
```

1. JVM uses new Operator through which object is created.
2. JVM sees the class name,checks whether .class file is available inside method area, if not
   found search in the current working directory load the .class file into method area.
3. Check for the instance variables in the loaded class, give memory for the instance variables.
4. Once the variables are found allocate the memory,and then give default value based on the
   datatype.
5. If instance block is found execute the instance block
6. Execute the constructor based on the way the programmer has called the constructor.
7. Get the reference of the object for future usage.

Write a java pgm to keep track of no of objects created in java class?

```
public class Test{
      static int count;
      {
            count++;
        }

      Test(){

      }
```

```
	Test(int a){

	}

	Test(int a,int b){

	}

	Test(int a,int b, int c){

	}

	public void display(){
		System.out.println("No of objects created is ::"+count);
	}
}
public class TestApp{
	public static void main(String... args){
		new Test();
		new Test(10);
		new Test(10,20);
		Test t= new Test(10,20,30);
		t.display();
	}
}
```

instance control flow

```
class Sample{
	int i=10;
	{
		methodOne();
		System.out.println("First instance block");
	}
	Parent(){
		System.out.println("Inside constructor");
	}
	public static void main(String... args){
		Sample s= new Sample();
		s.methodOne();
	}
	public void methodOne(){
		System.out.println(j);
	}
	{
		System.out.println("Second instance block");
	}
	int j=20;
}
```

Analysis

```
=======
 i=0[RIWO]
 j=0[RIWO]
 i=10[R&W]
 j=20[R&W]
```

Rule ::

```
 Identification of instance variables and instance block from top to bottom
 Execution of instance variables assignement and instance block from top to
bottom
 Execution of constructor.
```

Output:

```
 0
```

```
First instance block
Second instance block
Inside constructor
 20
```

Note::
 => Instance control flow is not one time activity, it happens for every object creation.
 => Object creation is most costly operation in java and hence if their is not specific
    requirement never recomonded to create objects.

Note::
case1::Compiler is responsible for the following checkings
==========
a. Compiler will check whether the programmer has wrote any constructor or not, if he didnt wrote
   any constructor then compile will keep default constructor with super() as the first line.
b. If the programmer wrote any constructor the compiler will check whether he wrote super()/this()
   in the first line, if the programmer didn't wrote any of these then compiler will keep
   always super().
c. Compiler will also check whether is there any chance of recursive invocation or not,if there is
   a possibility then compiler will raise compile time error.

case2::
```
class Parent{}
class Child extends Parent{}
```

     ====compiler internally generates===

```
class Parent extends Object {Parent(){super();}}
class Child  extends Parent{Child(){super();}}
```

#2.
```
  class Parent{ Parent(){}}
  class Child extends Parent{}
```

   ====compiler internally generates===

```
class Parent extends Object {Parent(){super();}}
class Child  extends Parent {Child(){super();}}
```

#3.
```
  class Parent{ Parent(int i){}}
  class Child extends Parent{Child(){super();}}
```

    ===compile time error==========

If parent class contains a parameterized constructor, while writing a child class we should take
special care w.r.t constructors.
When ever we are writing any parameterized constructor,it is highly recomended to write no argument constructor also.

case3::
```
  class Parent{  Parent()throws java.io.IOException{} }
  class Child extends Parent {}
```

====Compile time error======
If a parent class constructor throws checkedexception then compulsorily the child class constructor also should throw the same exception otherwise its parent exception otherwise it
would result in "Compile time Error".

solution::

```
class Parent{  Parent()throws java.io.IOException{} }
class Child extends Parent { Child()throws Exception{}}
```

========Compiler generated code =======

```
class Parent extends Object{  Parent()throws java.io.IOException{super();} }
class Child  extends Parent { Child()throws Exception{super();}}
```

Constructor role in inheritance
==========

=>Parent class constructor by default wont be available to the child class, so Inheritance concept is not applicable for constructors and hence overriding is not applicable for constructors.

=> Constructors can be overloaded.

=> Constructors we can write in any java class it can be concrete class, abstract class.

=> Constructors are not applicable for interfaces.

Role of final variables
==========

If a variable is declared as final, then compulsorily we should initialize the variable at one of the 3 differnt places

a. At the time of declaration
b. inside instance block
c. inside constructor

if we fail to do so it would result in compile time error.

eg#1.

```
class Demo{
      final int i=10;
}
```

eg#2.

```
class Demo{
      final int i;
      {
            i=10;
      }
}
```

eg#3.

```
class Demo{
      final int i;
      Demo(){
            i=10;
      }
}
```

Control flow in static
======================

```
class Sample{
      static int i=10;
      static{
            methodOne();
            System.out.println("First static block");
      }
      public static void main(String[] args){
            methodOne();
            System.out.println("Main method");
      }
      public static void methodOne(){
            System.out.println(j);
      }
        static{
            System.out.println("Second static block");
      }
      static int j=20;
 }
```

Rules for control flow
======================

1. Identification of static memebers from top to bottom
2. Execution of static varaibles assignements and static blocks from top to bottom.
3. Execution of main method

Analysis
========

i=0 [RIWO]
j=0 [RIWO]
i=10[R&W]
j=20[R&W]

ouput::

```
 0
First static block
Second static block
20
Main method
```

Rules of RIWO
=============

=> With in a static block, if we are trying to read any variable then that read is called as
   "Direct Read".
=> With in a method,if we are trying to read static variable then that read is called as
   "InDirect Read".
=> If a variable is in RIWO state then we can't perform read operation directly,if we try to
   do so it would result in compile time error saying "illegal forward reference".

eg#1::

```
class Test{
      static int i=10;
      static{
            System.out.println(i);
            System.exit(0);
      }
}
```

output:: 10

```
eg#2
class Test{
      static{
            System.out.println(i);
      }
      static int i=10;
}
output:: Compile time error:illegal forward reference

eg#3.
class Test{
      static{
            methodOne();
      }
      public static void methodOne(){
            System.out.println(i);
      }
      public static void main(String... args){

      }
      static int i=10;
}
output::0
```

static block
============
=> These are the blocks which gets executed automatically at the loading the .class files
=> If we want to perform any activity at the time of loading .class file we have to define
   that activity inside the static block.
=> We can write any no of static blocks, those static blocks will be executed from top to
   bottom.

```
eg#1.
public class Test{
      static{
            System.out.println("First static block");
      }
      static{
            System.out.println("Second static block");
      }
      public static void main(String... args){

      }
 }
output::
First static block
Second static block
```

Important questions on static block
==================================
1.Without using main() method is it possible to print some statements on the console?
Ans. Yes,by using static block.

```
eg#1.
class Google{
      static{
            System.out.println("Hello i can print");
            System.exit(0);
      }
```

```
}
```
It is valid till jdk1.6 from 1.7 onwards compulsorily we need to keep main method.

Examples applicable only for JDK1.6 examples

eg#2

Without using main() and static block is it possible to print some statements on the console

```
eg#1
class Test{
      static int i=methodOne();
      public static int methodOne(){
            System.out.println("Hello i can print");
            System.exit(0);
            return 10;
      }
}

eg#2.
class Test{
      static Test t=new Test();
      Test(){
            System.out.println("Hello i can print");
            System.exit(0);
      }
}

eg#3.
class Test{
      static Test t =new Test();
      {
            System.out.println("Hello i can print");
            System.exit(0);
      }
}
```

Without using System.out.println() can we print some statement of console?

```
 eg#1
 class Test{
      pubilc static void main(String... args){
            System.err.println("Hello i can print");
      }
 }
```

static variables made as final

==========

If static variable is made as final, then we need to make sure those variables are initialized
before the class loading process finishes its execution.

```
eg#1.
class Test{
      static int i;
      public static void main(String[] args) {
            System.out.println(i);
      }
}
eg#2.
class Test{
      final static int i;//CE
      public static void main(String[] args) {
            System.out.println(i);
```

```
	}
}
eg#3.
class Test{
	final static int i=10;
	public static void main(String[] args) {
		System.out.println(i);//10
	}
}

eg#4.
class Test{
	final static int i;
	static
	{
		i=100;
	}
	public static void main(String[] args) {
		System.out.println(i);//100
	}
}
```

Note:: If any exception occurs during static variables assignement and staic block execution then
it would result in "ExceptionInitalizerError".

```
eg#1.
class Test{
	static int a=10/0;
}
ouput: Expection called "java.lang.ExceptionInitalizerError"

eg#2.
class Test{
	static{
		String s=null;
		System.out.println(s.length());
	}
}
output:: Exception called "java.lang.ExceptionIntializerError"

eg#3.
public class Test3
{
	static int[] a;
	static{
		a[0]=10;
	}
	public static void main(String[] args) {

	}
}
eg#4.
public class Test3
{
	static int[] a=new int[-5];
	static{
		a[0]=10;
	}
	public static void main(String[] args) {

	}
}
```

Two or more methods is said to be Overloaded if both methods have same name but change in the
argument type.

```
     eg:: abs(int)
          abs(float)
          abs(long)
            ..
            ..
          etc
```

Overloading mechanism in java would reduce the burden of programmer(complexity is reduced).
In case of Overloading, compiler is responsible for method resolution based on the reference type of the argument,it will not bind based on the run time object so we say Overloading as
"CompileTime/Static/EarlyBinding".

eg1::

```
   class Test
   {
      public void methodOne(){System.out.println("Zero arg method");}
      public void methodOne(int i){System.out.println("Int arg method");}
      public void methodOne(Double f){System.out.println("Float arg
method");}
      public static void main(String[] args)
      {
           Test t= new Test();
                 t.methodOne(10);
            t.methodOne();

           Test t1=new Test();
           t1.methodOne(22.5);

      }
   }
```

eg2::

```
class Test
   {
      public void methodOne(int i){System.out.println("Int arg method");}
      public void methodOne(Float f){System.out.println("Float arg
method");}
      public static void main(String[] args)
      {
           Test t= new Test();
                 t.methodOne('a');//int arg
                 t.methodOne(25L);//float arg
                 t.methodOne(25.5);//CE
      }
   }
```

eg3::

```
class Test
   {
      public void methodOne(int i,float f){System.out.println("Int,Float
arg method");}
      public void methodOne(float f,int i){System.out.println("Float,Int
arg method");}
      public static void main(String[] args)
      {
```

```
            Test t= new Test();
                 t.methodOne(10,10.5f);
                 t.methodOne(10.5f,10);
            t.methodOne(10,10);//CE
                 t.methodOne(10.5f,10.5f);//CE
      }
   }

eg4::
class Test
   {
      public void methodOne(String s){System.out.println("String arg
method");}
      public void methodOne(Object o){System.out.println("Object arg
method");}
      public static void main(String[] args)
      {
            Test t= new Test();
                 t.methodOne("sachin");
                 t.methodOne(new Object());
            t.methodOne(null);//String arg method
      }
   }

ouput::Compiler will give preference for child type when it tries to
perform method               resolution

eg5::
class Test
   {
      public void methodOne(String s){System.out.println("String arg
method");}
      public void methodOne(StringBuilder
sb){System.out.println("StringBuilder arg method");}
      public static void main(String[] args)
      {
            Test t= new Test();
                 t.methodOne("sachin");
                 t.methodOne(new StringBuilder("sachin"));
            t.methodOne(null);//CE::Ambigous b/w String and StringBuilder
      }
   }

eg6::
 class Animal{}
 class Monkey extends Animal{}
 class Demo{
      public void methodOne(Animal a){System.out.println("Animal
version");}
      public void methodOne(Monkey d){System.out.println("Monkey
version");}
      public static void main(String[] args)
        {
                Demo d = new Demo();

           Animal a=new Animal();
                d.methodOne(a);//Animal version

                Monkey m=new Monkey();
                d.methodOne(m);//Monkey version
```

```
                Animal a1=new Monkey();
                d.methodOne(a1);// Animal version(because a1 type is
Animal)

      }
 }

eg7::
 class Test
 {
      public void methodOne(){System.out.println("No arg no return
type");}
      public int  methodOne(){System.out.println("No arg with return
type"); return 5;}

        public static void main(String[] args){
             Test t=new Test();
           t.methodOne();
           System.out.println(t.methodOne());
        }
 }
```

Note::When compiler binds the method call, it will refer to methodSignature only

In java methodSignature is

=> methodName(arg list)

Overloading w.r.t var args method

==========

```
eg1::
 class Test
   {
      public void methodOne(int ...i){System.out.println("Var arg
method");}
      public void methodOne(int i){System.out.println("Int arg method");}

      public static void main(String[] args)
      {
           Test t= new Test();
                t.methodOne(10);//Int arg method
            t.methodOne();//Var arg method
            t.methodOne(10,20,30);//Int arg method
      }
   }
```

Note:: var arg method will get last chance,just like default statement of switch meaning if none of match is found then vararg method will be executed by JVM.

Overloading of main method

==========

It is possible to overload main method also, but jvm will always call main method with the following argument only.

```
        public static void main(String[] args)

eg1::
   class Test
   {
      public static void main(String[] args)
        {
```

```
            System.out.println("String[] main method");
        }
        public static void main(int[] args)
        {
            System.out.println("int[] main method");
        }
    }
```
output:: String[] main method
Note:: The other overloaded method programmer should call.

eg2::
```
  class Parent{
        public static void main(String[] args)
        {
            System.out.println("String[] main method");
        }
  }
  class Child extends Parent{

  }
```
upon compilation, it will generate 2 .class files called Parent.class and Child.class
```
  javac Parent
  java  Parent => String[] main method
  java  Child  => String[] main method
```
Inheritance is applicable even for static methods also, if the method is not available in child class then from the parent class whatever method is available same method will be executed.

eg3::
```
  class Parent{
        public static void main(String[] args)
        {
            System.out.println("Parent main method");
        }
  }
  class Child extends Parent{
      public static void main(String[] args)
        {
            System.out.println("Child main method");
        }
  }
```
upon compilation, it will generate 2 .class files called Parent.class and Child.class
```
  javac Parent
  java  Parent => Parent main method
  java  Child  => Child main method
```
By seeing the output, we might refer it as Overriding but on static method Overriding is not possible it MethodHiding.(refer to MethodHiding).