Agenda :-

- Programming Paradigms
- Procedural programming
- Object Oriented Programming
- Access Modifiers

Programming Paradigms

↓

→ Style or standard way of writing program.

Without programming paradigm the code will be:

- Less structured
- Hard to read and understand
- Hard to test
- Difficult to maintain, etc.

Types of Programming Paradigm :-

→ Imperative Programming : telling the comp. how to do the task by giving set of inst. in a part. order, line by line.

```
// For eg:
int a = 10;
int b = 20;
int sum = a + b;
print(sum);
int dif = a - b;
print(dif);
```

→ Procedural Programming :-

we split the entire program into small procedures or functions, which are reusable code blocks.

```
// For eg:
int a = 10;
int b = 20;
addTwoNumbers(a, b);
subtractTwoNumbers(a, b);

void addTwoNumbers(a, b) {
    int sum = a + b;
    print(sum);
}

void subtractTwoNumbers(a, b) {
    int dif = a - b;
    print(dif);
}
```

→ OOP . → will discuss .

→ Declarative Programming :-

e.g., Select * from customers;

In this paradigm, you specify "what" you want the program to do without specifying "how" it should be done.

→ Functional Programming → will discuss in end .

→ Procedural Programming ←

It splits the entire program into small procedures or functions (section of code that perform a specific task) which are reusable code blocks. Eg - C, C++, etc.

Procedure is an oldage name of function/method.
Each procedure may internally call other proceduers.

```
// For eg:
void addTwoNumbers(a, b) {
    int sum = a + b;
    print(sum);
}

void addThreeNumbers(a, b, c) {
    int sum = a + b;
    addTwoNumbers(sum, c);
}

void main() {
    addThreeNumbers(10, 20, 30);
}
```

→ Problems with Procedural Programming ←

→ We are studying OOPs

→ Aman is teaching

→ Satwik is having breakfast.

→ We are listening to Aman's lecture.

Subject + Verb

(Someone is doing something)

Procedural Programming lang (C);

```
printStudent(String name, int age, String gender) {
    print(name);
    print(age);
    print(gender);
}
```

use 'Struct'

↓

It is like a class but not a class.

```
// For eg:
struct Student {
    String name;
    int age;
    String gender;
}
```

All variables in a struct are by default Public.

Something.

```
printStudent(Student st) {
    print (st.name);
    // In some programming languages like C, it's st->name
    print (st.age);
    print (st.gender);
}
```

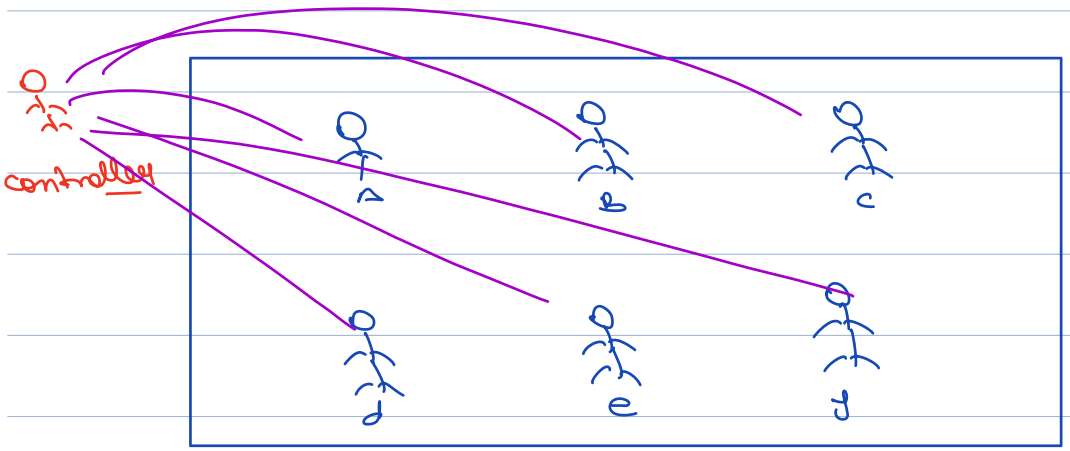
→ Someone.

Someone → Something

Something is happening on Someone.

printStudent(student) v/s student.print();

Procedural world



OOP → Each entity control its procedures, and also their defined behaviour.

```
class Student {
    String name;
    int age;
    String gender;

    void print() {
        Print(name);
        Print(age);
    }
}
```

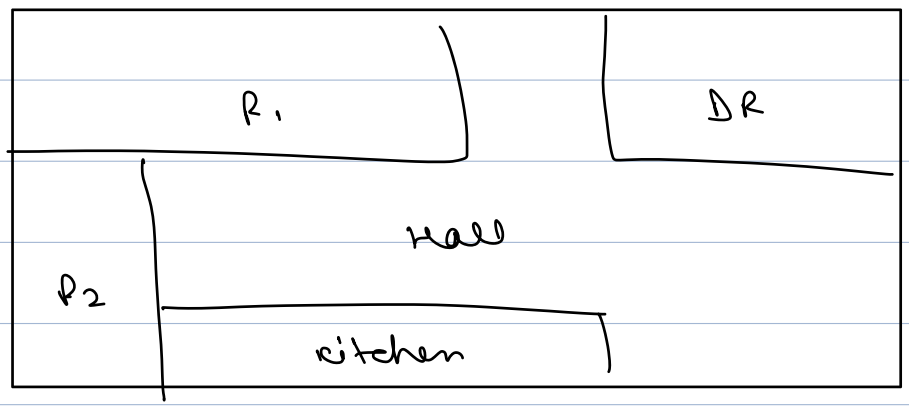
Cons of Procedural programming:

- Difficult to make sense
- Difficult to debug and understand
- Spaghetti code i.e. unstructured and needs to be tracked form multiple locations.

∴ OOP :- here program is build around classes and objects.

class :- Blueprint of an idea.

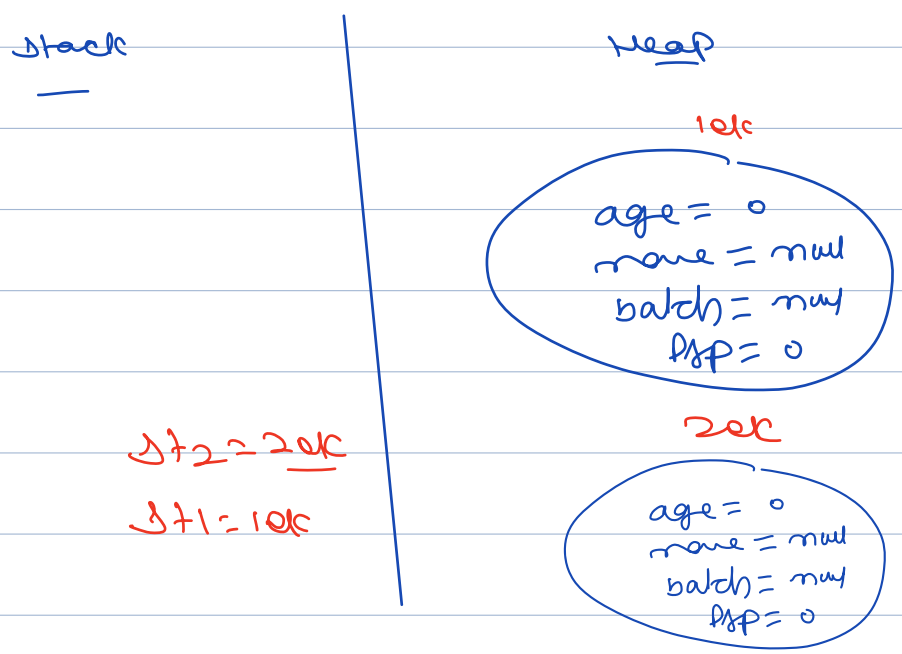
e.x floor of an Apartment.



class → represents structure of the idea.

```
class Student {
    int age;
    String name;
    String batch;
    double psp;
    changeBatch() { }
    pauseCourse() { }
    giveMockInterview();
}
```

```
Student st1 = new Student();
Student st2 = new Student();
```



```java
public class Student {
    String name;
    [redacted]
    String batchName;
    int age;
    double psp;

    void changeBatch(String newBatch) {
        batchName = newBatch;
    }

    void giveMockInterview() {
        System.out.println("Giving mock interview");
    }
```

3

main() {

Student jai = new Student();

jai.name = "jaitho";

jai.age = 16;

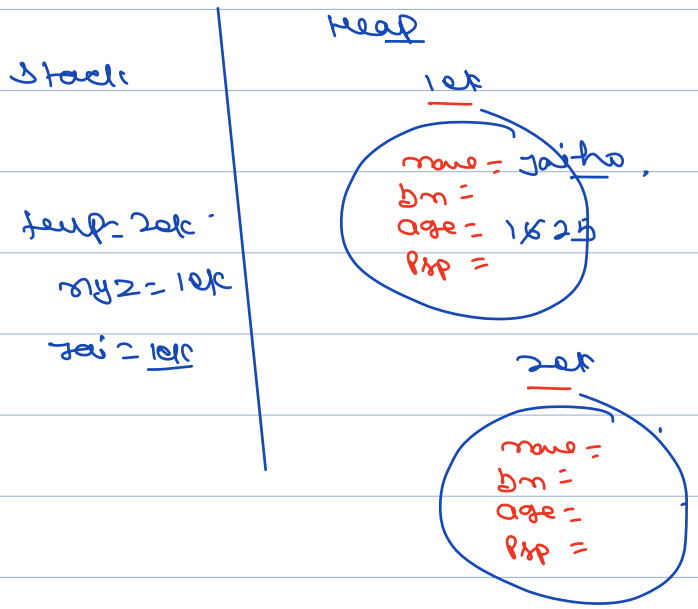
Student xyz = jai;

xyz.age = 25;

Print(jai.age);

Student temp = new Student();

3



Break

8:15 Am - 8:25 Am

28 Am

Pillars of Oop

→ 1 principle → foundation / fundamental concept

→ 3 pillars → support to hold things together.

I will be a good person → principle.

Pillars :-

I'll be truthful

I'll be honest to everyone.

Principle of OOP :-

↳ Abstraction.

Pillars of OOPS :-

→ Inheritance

→ Encapsulation

→ Polymorphism.

→ Ref :- Java : The complete Reference

Abstraction :- Representing in terms of idea.

↓

Purpose → others don't need to know details of the idea.

→ way to represent complex softwares, in terms of ideas.

→ Data

→ behaviours.

→ Encapsulation :-

↳ capsule.

↓

→ To hold the medicine powder together

→ To protect it from outside environment.

Class what do we really store together in program?

→ Attributes & behaviours.

↓ variables ↓ function

Access Modifiers

Encapsulation has two advantages :-

- ONE is it holds data and attributes together and → sorted by class.
- SECOND is it protect members from illegitimate access. You can't access the data from class unless the class allows you to.

↓ access modifiers.

Types of Access Modifiers :-

→ Public → It can be accessed by everyone.

→ Private → accessed by no one, Except that class.

→ Protected → Can be accessed within same pkg, in other pkg only subclass can access it.

→ default

```
Pkg1;
class Student {
    Public int age;
    Privat int rollno;
}

Student st = new Student();
st.age = 10;
```

St.rollno = 20; // not allowed.

	Class	Package	Subclass (same pkg)	Subclass (diff pkg)	World
`public`	+	+	+	+	+
`protected`	+	+	+	+	
no modifier	+	+	+		
`private`	+				

Pkg 2:

class Test Extends Studen {

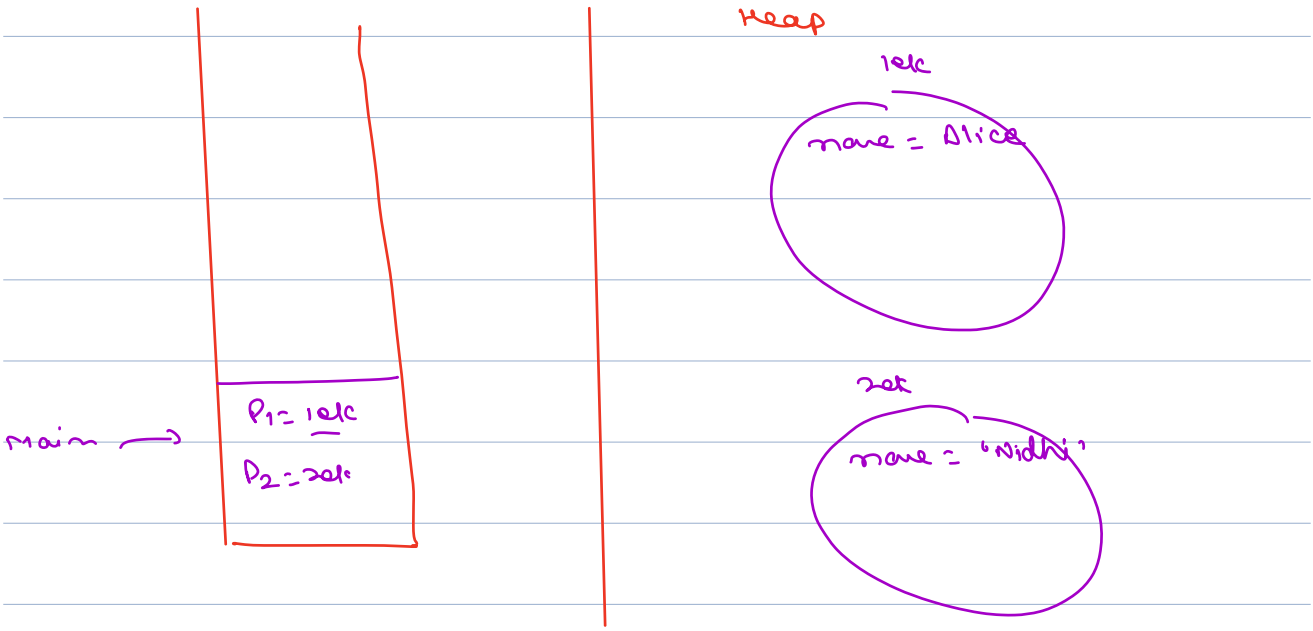
This keyword

↓

refers to current instance of an object.

```
class Person {
    private String name;
    public Person(String name) {
        this.name = name;
    }
    public void introduceYourself() {
        Print("Hello I am" + this.name);
    }
}

main() {
    Person P1 = new Person("Alice");
    Person P2 = new Person("Widhi");
    P1.introduceYourself();
    P2.introduceYourself();
}
```


Heap
1elc
name = Alice
2elc
name = 'Nidhi'
main
P1 = 1elc
P2 = 2elc

```java
package mypackage;

public class AccessModifierExample {
    public int publicVariable = 10; // Public access

    private int privateVariable = 20; // Private access

    protected int protectedVariable = 30; // Protected access

    int defaultVariable = 40; // Default (package-private) access

    public void publicMethod() {
        System.out.println("This is a public method.");
    }

    private void privateMethod() {
        System.out.println("This is a private method.");
    }

    public static void main(String[] args) {
        AccessModifierExample example = new AccessModifierExample();

        System.out.println("Public variable: " + example.publicVariable);
        System.out.println("Private variable: " + example.privateVariable);
        System.out.println("Protected variable: " + example.protectedVariable);
        System.out.println("Default variable: " + example.defaultVariable);
    }
}
```

```java
package otherpackage;

import mypackage.AccessModifierExample; // Import the class from a different package

public class AnotherClass {
    public static void main(String[] args) {
        AccessModifierExample example = new AccessModifierExample();

        System.out.println(example.publicVariable); // Accessing publicVariable is valid
        System.out.println(example.defaultVariable); // Error: Cannot access defaultVariable from a different package

        example.publicMethod();
        example.privateMethod(); // Error: Private method is not accessible outside the class
    }
}
```

Static keyword

↓

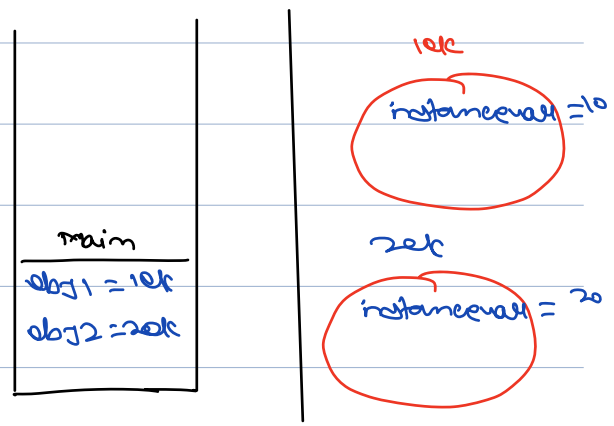
It is a class level member, it doesn't need any Object

staticVar = 0, ↑ 2

```
public class MyClass {
    // Static variable
    static int staticVar = 0;

    // Instance variable
    int instanceVar;

    public MyClass(int value) {
        this.instanceVar = value;
        staticVar++;
    }
```



public static void main (String [] args) {

```
MyClass obj1 = new MyClass(10);
MyClass obj2 = new MyClass(20);
```

```
System.out.println("Static Variable: " + staticVar);
```

→ 2

```
System.out.println("Instance Variable (obj1): " + obj1.instanceVar);
System.out.println("Instance Variable (obj2): " + obj2.instanceVar);
```

→ 10

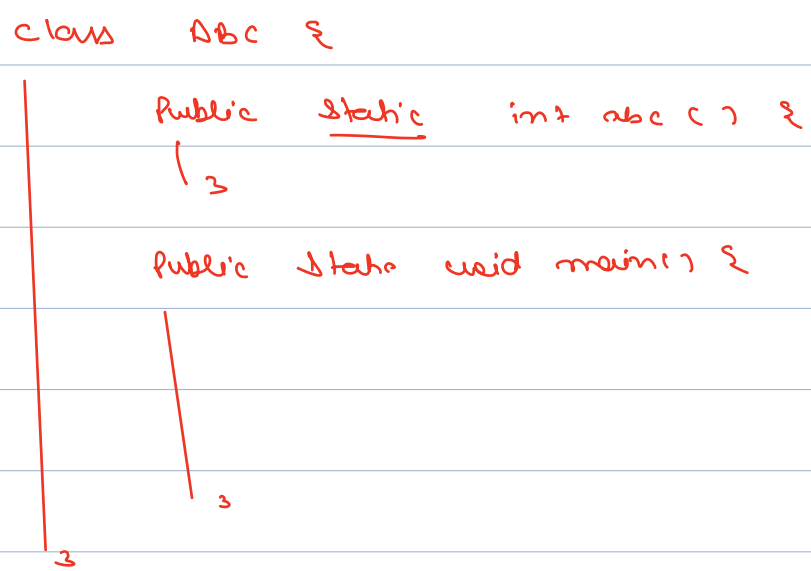
→ 20.

}

}

1. **Static Variables (Class Variables):** When you declare a variable as "static" within a class, it becomes a class variable. These variables are shared among all instances of the class. They are initialized only once when the class is loaded, and their values are common to all objects of the class.

2. **Static Methods (Class Methods):** When you declare a method as "static," it becomes a class method. These methods are invoked on the class itself, not on instances of the class. They can access static variables and perform operations that don't require access to instance-specific data.



* Variable of a Scope :-

1) class / static Scope ;
2) Instance Scope ;
3) method / local Scope
4) Block Scope .

```
public class ScopeExample {
    // Class-level variable (static scope)
    static int classVar = 10;

    // Instance variable (instance scope)
    int instanceVar = 20;

    public void exampleMethod() {
        // Method-level variable (method scope)
        int methodVar = 30;

        if (true) {
            // Block-level variable (block scope)
            int blockVar = 40;
            System.out.println(classVar + instanceVar + methodVar + blockVar);
        }
        // The 'blockVar' is out of scope here.
    }

    public static void main(String[] args) {
        ScopeExample obj = new ScopeExample();
        obj.exampleMethod();
        // The 'methodVar' and 'blockVar' are out of scope here.
    }
}
```

1. **Class/Static Scope:** Variables declared as `static` within a class have class-level scope. These variables are associated with the class itself rather than with instances (objects) of the class. They can be accessed using the class name and are shared among all instances of the class.

2. **Instance Scope:** Variables declared within a class but outside any method or constructor have instance scope. These are often referred to as instance variables, and they are associated with specific instances (objects) of the class. Each object has its own copy of these variables.

3. **Method/Local Scope:** Variables declared within a method or a block of code have method or local scope. These variables are only accessible within the specific method or block where they are defined. They go out of scope when the method or block's execution is complete.

4. **Block Scope:** Variables declared within a pair of curly braces `{}` have scope limited to that block. These variables are only accessible within the block in which they are defined.