Agenda

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

Programming Paradigms

style / way / guidelines of writing programs

If no paradigm :-

→ Less structured

→ Hard to understand

→ Hard to test

→ Hard to maintain

Types of Programming Paradigms

- Imperative Programming

  solve instructions line by line

```
int a = 2
int b = 3
int sum = a + b;
print(sum);
int diff = a - b;
print(diff);
```

- Procedural Programming

  Entire program is split into functions / procedures which are reusable code blocks.

```
int sumTwoNums(a, b) {
    return a+b;
}
a = 5
b = 7
int c = sumTwoNums(a, b);
int d = sumTwoNums(c, 5);
;
print(d);
```

- Object Oriented Programming
  main entity → object
- Declarative Programming
  select * from Customer; (SQL query)

Procedural Programming

students [ ]
instructors [ ]
⋮
} data

To get the info of 1 student.
→ students [ ]
→ psh[ ]
→ batches[ ]

```
changeStudentBatch(id, batch){
    ...
    ...
}
```

Real world → X is doing some action Y.


```
struct Student{
    String name;
    int age;
    int psh;
    String address;
    String gender;
    String batch;
}
changeStudentBatch(Student st, String batch){
    st.batch = batch;
}
main(){
    Student st ...
    changeStudentBatch(st, "XYZ");
}
```

controlled by main()
(can be done by any fn.)

Object Oriented Programming

- Entity is at the core
- Every entity will have some attributes and behavior

We build the whole OOPS Program using objects and classes.

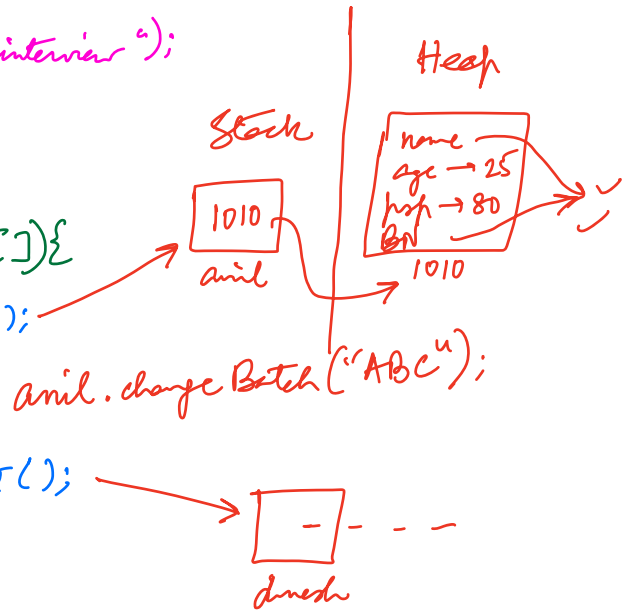
class → Blueprint of an idea/entity.

```
class Student{
    String name;
    int age;
    int psh;
    String address;
    String gender;
    String batch;
}
```

→ class info is stored in 'Method Area'.
→ Not a real entity
→ multiple instances of this class

objects → instances of a class.

```
public class Student {
    String name;
    String batchName;
    int age;
    double psp;
    void changeBatch(String newBatch){
        batchName = newBatch;
    }
    void giveMockInterview(){
        System.out.println("giving mock interview");
    }
}
class OtherClass {
    public static void main(String args[]){
        Student anil = new Student();
        anil.name = "Anil";
        anil.age = 24;
        Student dinesh = new Student();
        dinesh.name = "Dinesh";
        dinesh.age = 19;
    }
}
```



OOPS → 3 Pillars → Inheritance, Polymorphism, Encapsulation.
1 Principle → Abstraction

(Java: The complete reference)

[Break till 10:38 PM]

Abstraction → Representation in terms of idea (not the inner details) | what this fn. does
Not how
X

↓

of data
and
anything that has behavior

Encapsulation

capsule :-
→ holds items together
→ helps items to avoid mixing with each other
→ protects from outside environment

In Programming → "Attributes" and "Behavior" are those items.

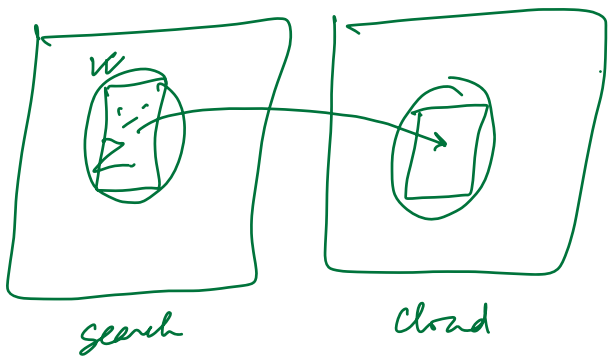
Class brings about encapsulation.

Access Modifiers

Encapsulation → holds data and attributes together (class)
Encapsulation → Protects from illegitimate access

↓

Access specifiers



4 access modifiers :-

→ Public ⇒ accessible to any one
→ Private ⇒ only accessible to same class
→ Protected ⇒ only accessible to same package or sub-class in diff. package
→ Default ⇒ only accessible to same package
(No modifier)

	class	Package	Subclass (same pkg)	Subclass (diff pkg)	World
public →	✓	✓	✓	✓	✓
protected →	✓	✓	✓	✓	✗
<no modifier> →	✓	✓	✓	✗	✗
private →	✓	✗	✗	✗	✗

```
public class Student {
    private String name;
    public Student(String name) {
        this.name = name;
    }
    public void intro() {
        S.o.pln("Hello, I am " + this.name);
    }
    :
}
```

this.name → name variable of the current object.

```
package mypackage;
public class AccessModifierExample {
    public int a = 10;
    private int b = 20;
    protected int c = 30;
    int d = 40;
    public void method1() {
        ...
    }
    private void method2() {
        ...
    }
    public static void main(String args[]) {
        AccessModifierExample obj = new AccessModifierExample();
        Sopln(obj.a);
        Sopln(obj.b);
        obj.method1();
        obj.method2();
    }
}
```

obj.c ✓
obj.d ✓

```
package otherpackage;
import mypackage.AccessModifierExample;
public class AnotherClass {
    public static void main(String args[]) {
        AccessModifierExample obj = new AccessModifierExample();
        Sopln(obj.a);
        Sopln(obj.b);
        obj.method1();
        obj.method2();
    }
}
```

Private

Private

obj.c ✗
obj.d ✗

static

class-level member or method.

not owned by the objects.

static variables

`static int x;` — shared across all instances of the classes

initialized when the class is loaded.

static methods

invoked on the class level.

only work with static (and local) variables.

```
int a=0
static int b=0 ──→ shared b/w obj1, obj2.
void xyz(){
    a++
    b++
}
```

obj1.xyz() ⟶ a=1, b=1

obj2.xyz() ⟶ a=1, b=2.

Scope of a variable

1) Class/static scope

2) Instance scope
   carry values for a specific instance of a class

3) Method / local scope
   within the method

4) Block scope
   within { }

```
class Example {
    // class-level var (static)
    public static int x = 5;
    // instance var (instance scope)
    int y = 4;

    public void exMethod() {
        int z = 5; // method scope
        if (x == 5) {
            int w = 8; // Block scope.
            :
        }
    }
    p s v main (. ~) {
        x ✓
        y (with an object) ✓
        z ✗
        w ✗
        exMethod (with an object) ✓
    }
}
```