OOPS

Agenda:
1. What is class?
2. What is an object?
3. Constructor:
   a. Default
   b. Parameterised
   c. Copy Constructor
4. Deep Copy & Shallow Copy

- **class** → blueprint that defines what properties (attributes) and actions (methods) an entity will have.
  - not real entity and hence it doesn't occupy space in memory.

Ex:
```
class student {
    attributes:    String name, email;
                   int marksIcp, marksWD, marksMaths;
                   double cgr;
    functionalities: solveProblems() {----}
                     applyLeaves() {----}
}
```

- **Abstraction** → process of hiding implementation details, showing only essential features of an object or system.

Constructors → special method called automatically when an object is created.
- allocates memory and initialise attributes.
- It has same name as class name.

Types

(1) **Default Constructor:** It is automatically created only when we don't create our own constructor for class
- no parameter
- assign default value to the attributes

```
class Student {
    String name;
    int age;
    double psp;
    String univName;

    Student() {
        name = null;
        age = 0;
        psp = 0.0;
        univName = null;
    }
}
```

automatically created

② **Parameterised Constructor**: Created by user when we want to initialise the attributes based on values given by user or our own default values.

```
class student {
    String name;
    int age;
    Student (String stName, int age){
        name = stName;
        this.age = age;
    }
}
```

→ instance variables

```
Student st = new Student ("Riya", 18);
```

~~Student st2 = new Student();~~

★ `this` keyword is used to refer the current instance of the class.

---

③ **Copy Constructor**: This is essentially a parameterised constructor that takes an object of the same class as its parameter, creating a new object that is a copy of the existing one.

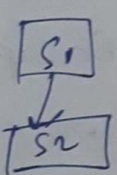
```
class student {
    String name;
    int age;

    Student() {
        name = null;
        age = 0;
    }

    Student (Student st) {
        name = st.name;
        age = st.age;
    }
}
```

→ Copy constructor

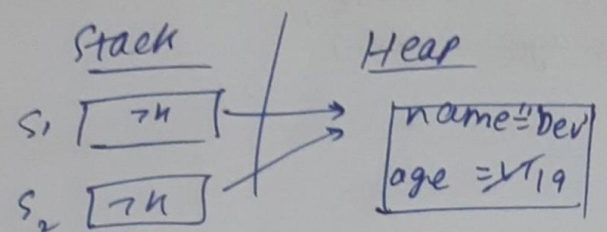
```
Student s2 = new Student(s1);
```



ShallowCopy vs DeepCopy

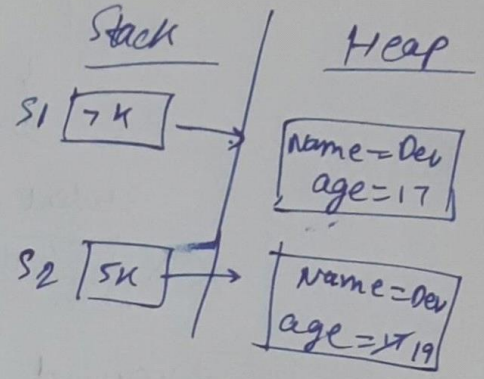
SHALLOW COPY

```
Student s1 = new Student ("Dev", 17);
Student s2 = s1;
s2.age = 19
print (s1.age); → (19)
```



DEEP COPY

```
Student s1 = new Student ("Dev", 17);
Student s2 = new Student (s1);
s2.age = 19
print (s1.age) → (17)
```



OOPs Benefits

- Modularity/code-reuse → Put related data + behaviour together in a class, extend it later.
- Abstraction → hide complex implementation behind simple interfaces.
- Encapsulation → keep an object's internal state protected.
- Polymorphism → use the same name to mean different behaviour (overloading/overriding).
- Inheritance → create new classes from existing ones to reuse and extend behaviour

(1) Inheritance (code-reuse) → let one class extend another:

```
Public class Student { /* base class */ }

Public class SIT Student extends Student {
    // extra fields / methods
}
```

- `SSTStudent` automatically has members from Student (unless they are private).
- Use `super(...)` in a subclass constructor to call the parent constructor

Ex: instead of copying code, extend a base class and add only what's new.

```
import java.util.*
class Student {
    String name;
    # constructor for student
    public Student (String name){
        this.name = name;
    }
}
# SSTstudent inherits from student
class SSTStudent extends Student {
    int marksInEnglish;

    # Constructor for SSTStudent
    public SSTStudent ( String name , int marksInEnglish){
        super(name);   # Call parent class constructor
        this.marksInEnglish = marksInEnglish;
    }
}

# Main class to run the Program
public class Main {
    public static void main (String[] args ) {
        SSTStudent s1 = new SSTStudent ("Amit", 90);
        Sopln ("Name:" + s1.name);
        Sopln ( "Marks in English:" + s1.marksInEnglish);
    }
}
```

s1 is an object

output: Name: Amit
Marks in English: 90

POLYMORPHISM ⇝ "many forms, same name"

A. Method Overloading (compile-time / static polymorphism)

Same method name, different parameter lists in the same class:

```
class Vehicle {
    void accelerate() { }
    void accelerate(int amount) { }
    void accelerate(double amount, int time) { }
}
```

Compiler chooses which method to call based on argument types and counts.

B. Method Overriding (runtime / dynamic polymorphism)

A subclass provides a specific implementation for a method declared in the parent class. Method signatures must match.

```
class Parent {
    void job() { Sopln("Parent job"); }

class Child extends Parent {
    @Override
    void job() { Sopln("Child job"); }
}
```

class → reference variable → creates new object in memory

```
// Parent reference, child object
Parent P = new Child();
P.job(); # calls Child.job() at runtime
```

- dynamic dispatch

Multiple Inheritance

↳ Java don't support multiple inheritance of classes (i.e., class C extends A, B is illegal) because it creates ambiguity (the "diamond problem").

Feature	Overloading (compile-time)	Overriding (Runtime)
• When decided	Compile Time	Runtime
• Method Signature	Must differ	must be same
• Class Relationship	Same class	Parent-child classes
• Return Type	Can differ	Must be same or covariant
• static Methods	Can be overloaded	Cannot be overridden

Agenda:
1. Static Keyword
2. Encapsulation
3. Access Modifier
4. Building our own ArrayList Class
5. Nested class

(1) • Static means "belonging to the class, not to any object"
↳ allows variables, methods, blocks, and nested classes to be shared among all objects of a class.

```
class Student {
    String name;
    int RollNo;
    static String college = "IIT Delhi;   // shared by all

Student(String n, int r) { name = n; rollNo = r; }
void show() { Sopln (name + " (" + rollNo + ") - " + college); }

public static void main (String[] args) {
    Student s1 = new Student ("Amit", 1);
    Student s2 = new Student ("Riya", 2);
    s1.show(); → Amit (1) - IIT Delhi
    s2.show(); → Riya (2) - IIT Delhi
    Student.college = "IIT Bombay"; // change once, reflects everywhere
    s1.show(); → Amit (1) - IIT Bombay
    s2.show(); → Riya (2) - IIT Bombay
}
}
```

Student.college ☑ // Allowed - belongs to class
Student.name ☒ // Not Allowed - belongs to object

• non-static variable ~~referenced~~ cannot be referenced from a static context.

- Static Block - Don't need an object.

like a tool box

```
class MathUtil {
    static int square (int x) {return x * x;}
    public static void main (String[] args) {
        Sop ln (MathUtil.square(5)); // no object needed
    }
}
```

---

⑥ Static Block - One-time Setup

- **Static methods** are loaded into memory once when the class is loaded (before any objects exist)
- **Instance variables** belong to specific objects, and exist only after you create an objects using new keyword.
- So, inside a static method, there's no specific object to know which instance's variable you're referring to.

```
class Config {
    static { Sopln ("setting up system....."); }
    public static void main (String[] args) {
        sopln ("Running main ....");
    }
}
```

IMP: Static can't access non-static (instance variable) data directly.
(methods/variables) (object-specific)

```
class Demo {
    int a = 10;
    static void test () {
        // Sopln (a); ✗ not allowed (needs an object)
    }
}
```

Static methods don't follow rules of Polymorphism

- Static methods not duplicated in child
- Both parent and child share the same memory for it.
- changing via one affects all.

```
class Parent {
        static String College = "IIT Delhi";
    }
class Child extends Parent {
        void show() {
            Sopln("College:" + college);
        }
    }
public class Test {
        public static void main(String[] args) {
            Child.college = "IIT.Bombay";
            Sopln(Parent.college);      //IITBombay
            new Child().show();         // IITBombay (Copy)
        }
    }
```

Static Methods are not Overridden

~~if parent is static and the~~

```
class Parent {
        static void display(){ Sopln("Static Parent"); }
            void show() { Sopln("Instance Parent"); }
    }
class Child extends Parent{
        static void display(){ Sopln("Static Child");}
            void show() { Sopln("Instance Child");}
    }
public class Demo{
public static void main(String[] args) {
        Parent p = new Child();
        p.display();    // Static Parent
        p.show();       // Instance Child
    }
}
```

Instance Methods vs Static Methods in Inheritance

If a parent defines a non-static method and child defines a static one (or vice versa), it's an error - you can't override an instance method with a static one or hide a static method with an instance one.

- The Type static or not must stay consistent.

```
class Parent {
    void greet() { }
}

class Child extends Parent {
    static void greet() { }    // x Compile-time error
}
```

Encapsulation

Encapsulation means restricting direct access to the internal details of a class and allowing interaction only though controlled methods.

Purpose of Encapsulation:

- DATA PROTECTION: Prevents unauthorised or invalid modifications.
- VALIDATION: It ensures data validation - every modification can be checked for correctness before being applied to the object.
- LOGGING: Every update can be tracked for debugging or audit. Tracking changes becomes easier which is crucial for debugging, auditing, or maintaining version history.
- MODULARITY: By isolating the internal logic, encapsulation also makes code modular - internal implementations can change without affecting how other parts of the program interact with it.

How to Enforce Encapsulation?

- Using Access Modifiers.
- Access Modifiers - decide who can see or use a class, variable or method.

Access Modifier	Same Class	Same Package	Subclass (Different Package) (child)/(extends)	Other Packages
• PRIVATE (most restricted)	☑	X	X	X
• DEFAULT (Package Level)	☑	☑	X	X
• PROTECTED (Inheritance Level)	☑	☑	☑ (via inheritance)	X
• PUBLIC (most open)	☑	☑	☑	☑

- Private → usage: used for internal data (eg: variables like cgr)
- default → usage: for helper classes or methods not meant for external packages.
- Protected → usage: used when subclasses need access to parent members.
- public → usage: Used for APIs, public methods, constructors.

Building own ArrayList Class

ArrayList in Java is a powerful example of encapsulation - it hides complex internal details like resizing arrays and tracking capacity, exposing only safe and simple methods to the user.

↳ Java ArrayList internal working:

① **Private Structure**: Uses a private Object element Data to store elements.
→ Maintains size (no. of used slots).
→ Auto-resizes when full.

② **Public Methods**: Methods like add(), remove(), get(), etc.,
user interacts through these only.

③ **Encapsulation Benefit**: Prevents direct access to internal array.
→ Ensures index safety (IndexOutOfBoundsException).
→ Simplifies usage - no manual array management.

Step-by-Step Design:

#refer code in vsc

1. **Define Private Structure:**

```
private int[] arr;
private int size, capacity;
```

2. **Initialise Base State (Constructor Initialisation):**

```
Public MyList() {
    capacity = 5;
    arr = new int[capacity];
    size = 0;
}
```

3. **Add Method Logic:** If full → call expandArray(), Else insert new valu

Expansion Logic: Double capacity and Copy all Elements to a new, larger array.

```
Public void add(int val) {
    if (size == capacity) expandArray();
    arr[size++] = val;
}
private void expandArray() {
    capacity *= 2;
    arr = Arrays.copyOf(arr, capacity);
}
```

4. **Provide Controlled methods**

- add (E element)
- get (int index)
- set (int index, E element)
- contains (E element)
- indexOf (E element)
- size()
- isEmpty()
- clear()

Nested class → class defined another class

```
class Outer {
    class Inner {
        void msg () { Sopln ("Hello from inner class"); }
    }
}
```

Non-static Inner Class (Regular Inner Class)

- Exists within an object of the outer class.
- Can access outer class members directly, even in private.
- Needs an outer object to be created.

```
class Outer {
    private String msg = "Hello from Outer";

    class Inner {
        void display() {
            Sopln ("Inner says: " + msg);
        }
    }

    public static void main (String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();
        inner.display();          //needs outer object
    }
}
```

Static Nested Class

- Declared using static
- Doesn't need an outer class object to be created.
- Can only access static members of the outer class.

```
class Outer{
    Static String name = "Outer Class";
    Static class Inner {
        void show() {
            Sop ln ("Accessing:" + name);
        }
    }
    Public static void main (String [] args ){
        Outer.Inner in = new Outer.Inner(); //no outer object needed
        in.show ();
    }
}
```

Function calling

```
Public class solution{
    public int solve (int[]A , int B ) {
        { Logic }
        return sum;
    }
Public static void main ( String[]args ){
    Scanner sc = new Scanner ( System.in);
    int n =sc.nextInt();
    int [ ]A = new int [n];
    for( int i=0 ; i<n ; i++ ){
        A[i] =sc.nextInt ();
    }
    int B = sc.nextInt();
    Solution obj = new Solution ();
    int result = obj. solve (A,B) ;
    Sop ln ( result) ;
```