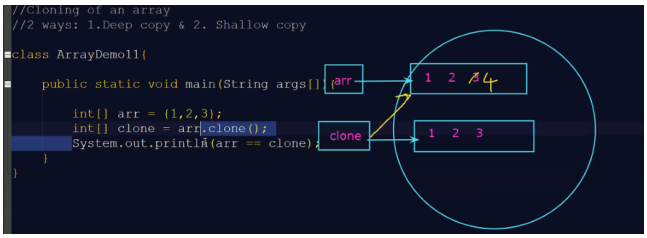
OOPJ_Day 6_Sanket Shalukar

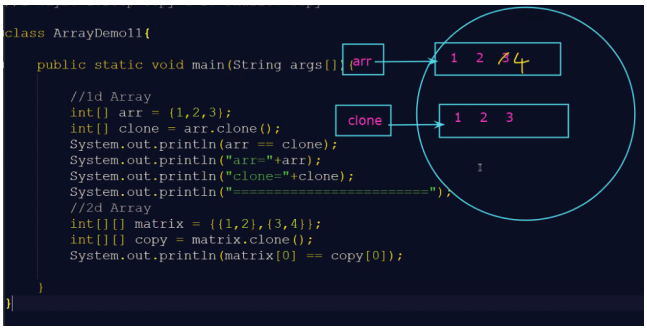
Tuesday, September 02, 2025 10:12 AM

Topics are in the Day_6

- ***Array***
- ***OOPS Pillars***
- ***Abstraction***
- ***Encapsulation***

Cloning of an array :





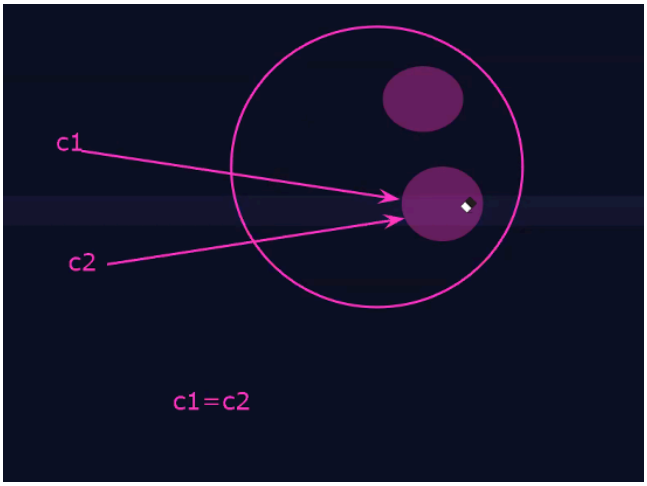
Shallow Copy :

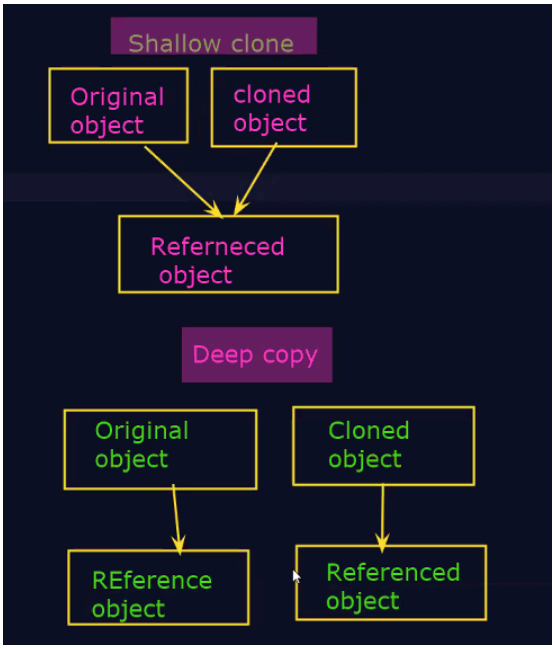
In **Java**, a **shallow clone** means creating a copy of an object **where only the top-level (primitive fields and object references) are duplicated**, but the **nested objects are not cloned**.

- Copies the reference of the object, not the actual data
- Both the original and copied object point to the same memory
- Changes in one object reflect in the other object

Uses of Shallow Copy :

- **Fast & Lightweight** → Because it just copies references instead of creating new objects.
- **Useful when objects are immutable** (like `String`, `Integer`, etc.) since shared references won't cause problems.
- **Good for temporary copies** where you don't plan to modify nested objects.
- Example: Copying a configuration object that holds mostly immutable data.





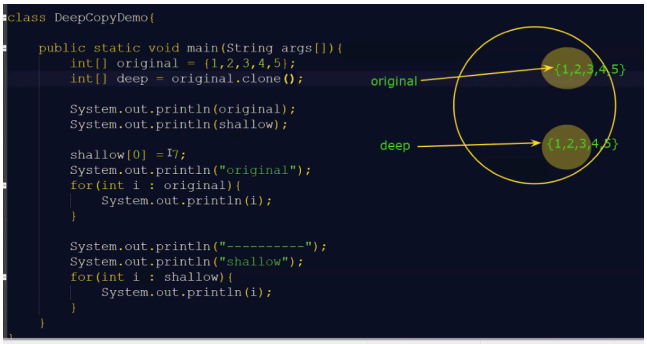
Deep Copy :

A deep copy creates a new object along with **new copies of all referenced objects** inside it.
So the original and copy are **completely independent**.

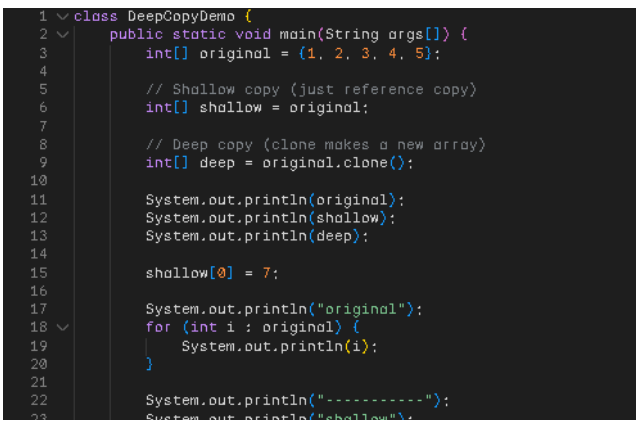
- Copies the actual data into a new object.
- The original and copied object are independent.
- Changes in one object do not affect the other.

Uses of Deep Copy :

- **Ensures independence** → Changes in one object do not affect the other.
- **Required when objects are mutable** (like arrays, lists, or custom classes).
- **Safer in multi-threaded programs** → each thread can work on its own copy without conflicts.
- Example: Copying a student record with address details, where each student must have a separate copy of the address.



Code that explains Shallow and Deep Copy



```
class DeepCopyDemo {
    public static void main(String args[]) {
        int[] original = {1, 2, 3, 4, 5};

        // Shallow copy (just reference copy)
        int[] shallow = original;

        // Deep copy (clone makes a new array)
        int[] deep = original.clone();

        System.out.println(original);
        System.out.println(shallow);
        System.out.println(deep);

        shallow[0] = 7;

        System.out.println("original");
        for (int i : original) {
            System.out.println(i);
        }

        System.out.println("----------");
```

```
        for (int i : shallow) {
            System.out.println(i);
        }

        System.out.println("-----------");
        System.out.println("deep");
        for (int i : deep) {
            System.out.println(i);
        }
    }
}

```

OOPS Pillars :

1. **Abstraction:**
2. **Encapsulation**
3. **Inheritance**
4. **Polymorphism**

- ### Abstraction :

It is a process of Hiding internal implementation details and showing only the necessary features to the user.

Defining a method : Writing the full code of the method (its signature + body) that tells what the method will do when it is called.

Real Life Example: When you use an ATM machine, you just press buttons for withdrawal or deposit. You don't know how the bank's backend processes your request.

Key Features :

1. Hides complex details and shows only essential things.
2. Achieve abstraction using 'abstract' or 'interface'

Techniques :

1. Abstract class - (Partial abstraction)
2. Interface - 100% abstraction)

Concrete Class :

- It is a fully implemented class in java
- It has all methods implemented (No abstract method) and can be instantiated directly.
- We can create objects of the class
- We can contain instance variable, constructors and method (fully implemented)
- Can extend abstract classes or implement

```
class Book{
    private String title;

    Book(String title){
        this.title = title;
    }

    void readbook(){
        System.out.println("Read method: read()");
    }
}
class ConcreteDemo{

    public static void main(String args[]){
        Book b1 = new Book("The Alchemist");
        b1.readbook();

    }
}
```

Anonymous Class :

- An anonymous class is a class without a name, Created on at runtime while instantiating objects.
- It is often used to provide immediate implementation of abstract class or interface.
- It cannot have constructor. (Because it has no name)
- Usually used for one time use.

```java
interface Book{

    void readbook();
}
public class AnonymousDemo{

    public static void main(String args[]){
        //Anonymous class implementing Book interface
        Book b1 = new Book(){
            //method implementation
            @Override //annotation
            public void readbook(){
                System.out.println("Read method: read()");
            }

        };


        b1.readbook();

    }
}
```

Abstract Class :

An **abstract class** in Java is a class declared with the `abstract` keyword that cannot be instantiated. It is designed to be inherited and can contain both abstract methods (without body) and concrete methods (with implementation).

- Declared class with 'abstract' Keyword:
- Can have abstract method(no body) and concrete method. (with body)
- We cannot instantiate abstract class directly, require subclass (Inheritance)

If you want to create abstract class

abstract class Book {

Abstract void display (); //abstract method

}

```java
abstract class Shape{//abstract class
    String color;

    Shape(String color){
        this.color = color;
    }

    abstract double area();//abstract method

    void display(){
        System.out.println("Abstract Class : Example");
    }

}

public class AbstractDemo{

    public static void main(String args[]){

    }
}
```

class: shape

-color
-shape()
-abstract double area();
-void display() { }

Advantages of Abstract Class.

1. Code reusability: Common methods can be written once in the abstract class and reused by subclasses.
2. Provides a base for subclasses: Ensures that certain methods must be implemented in child classes.

Disadvantages of Abstract Class.

1. No multiple inheritance: A class can extend only one abstract class.
2. Cannot be instantiated: You cannot create objects directly from an abstract class.

- Differentiate between Abstraction, Abstract Class, and Abstract Method (IMP –Interview Question)

1. Abstraction

- It is a concept of hiding implementation details and showing only the necessary features.
- It is achieved using abstract classes and interfaces.

2. Abstract Class

- It is a class declared with the `abstract` keyword.

- It cannot be instantiated and may contain both abstract and concrete methods.

3. Abstract Method

- It is a method declared with the `abstract` keyword without a body.
- It must be implemented by the subclass that extends the abstract class.

- **Interface (100% Abstraction) :**
  1. Interface is a blueprint for a class
  2. It contain abstract methods and constant
  3. Classes implements interface using 'implements' Keyword.
  4. From Java 8, interfaces can also have default (instance method) and static methods

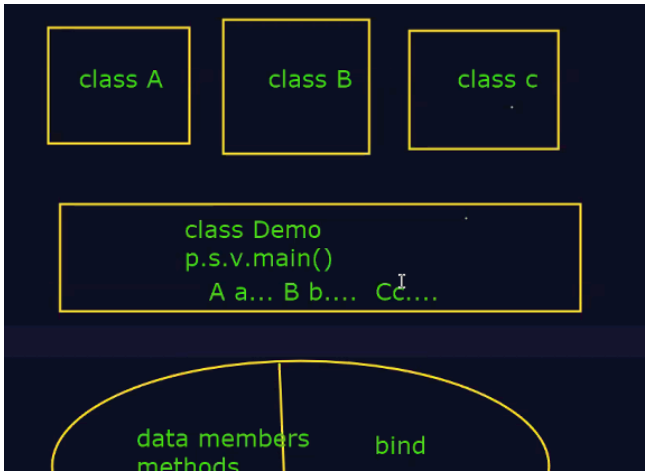
Example -

```
interface Shape {
        Double area(); //abstract method
        }
        Void display(){
        }
}
Class Demo
```

```java
interface Shape{//interface
    double calculateArea();// abstract method
}

class Rectangle implements Shape{
    double length, width;

    Rectangle(double length, double width){
        this.length = length;
        this.width = width;
    }

    @Override
    public double calculateArea(){
        return length * width;
    }

}

public class InterfaceDemo{

    public static void main(String args[]){

        //Rectangle r1 = new Rectangle(4.0,5.0);
        Shape r1 = new Rectangle(4.0,5.0);
        //Shape s1 = new Shape();//Error: object cannot be instantiated
        double res = r1.calculateArea();
        System.out.println("Result = "+ res);

    }
}
```

- **Encapsulation :**

Wrapping data (variables) and methods (functions) together into a single unit (class) and restricting direct access to data.
Example: In a capsule (medicine), different salts are wrapped together. You can only consume the capsule, not directly access the salts inside.



Class: Instance variable, method ()

- It is one of the fundamental principle of OOP that binds data methods into single, unit, hiding internal implementation details and providing controlled access through public methods.
- It combine data members and methods into single unit.
- Restrictes directs access to data members.
- It provides public getter and Setter methods to access and modify private data.
- It ensures data hiding and data security.
- Improves code reusability and modularity.

Encapsulation Implementation :

1. Declaring instance variable as 'Private'
2. Providing 'getter-setter' methods to access and modify the private variables.
3. Implementation data validation within setter methods.

```
class Employee{
    private int id;
    private String name;

    Employee(){
        this.name = "Kiran";
    }

    //Getter Methods
     public int getId(){
         return id;
     }

     public String getName(){
         return name;
     }

     //Setter methods
      public void setId(int id){
          this.id = id;
      }

       public void setName(String name){
          this.name = name;
       }


}
public class EncapsulationDemo{

    public static void main(String args[]){

        Employee e1 = new Employee();
        e1.setName("Purva");
        e1.setId(111);

        System.out.println("Emp Id="+e1.getId()+" Emp Name="+e1.getName());

        Employee e11 = new Employee();
        e11.setName("");
        e11.setId(222);

        System.out.println("Emp Id="+e11.getId()+" Emp Name="+e11.getName());
```

- **Inheritance :**

The process where one class acquires the properties and behaviors (methods) of another class.
Example: A child inherits physical features and qualities from parents. Similarly, a Car class can inherit from a Vehicle class.

- **Polymorphism :**

The ability of a method or object to take many forms, allowing the same name to be used with different behaviors.
Example: A person can be a student in school, a customer in a shop, and a player on the ground — same person, different roles **depending** on the situation.

Annotation in Java :

An annotation in Java is a special form of metadata (extra information) that you attach to classes, methods, variables, parameters, or packages.
It does not change the code's logic, but it gives instructions to the compiler, tools, or frameworks about how to treat that code.

Common Built-in Annotations

1. **@Override** → checks if method overrides superclass method.