

8. Encapsulation & Inheritance

8.1) Intro to OOP Principle

Oops principles are not specific to java.

There are 4 oop principles,

- 1) Encapsulation - only exposing the selected information from an object.
- 2) Abstraction - Hides the complex details to reduce complexity.
- 3) Inheritance - Entity can inherit attributes from another entity.
- 4) Polymorphism - Entities can have more than one form.

8.2) Encapsulation

Class is the combination of variables and methods.

Encapsulation hides internal data and allows access only through methods.

Encapsulation can be implemented in java through these access modifiers public, private, protected.

Getter and setter methods hides the properties and creates methods to expose information.

Maintains integrity by avoiding external interference.

Enhances modularity with the concepts of imports and packages.

8.3) Import and packages

Packages are declared at the beginning of the source file using the package keyword followed by package name.

No 2 classes can have the same name in the same package.

If we have car class in package1 and it is completely valid to have car class in package2 also.

This is how we can use the car class of package2 from package1.

```
package in.Package2;

import java.util.Scanner;

public class Car {
    public static void main(String[] args) {
    }
}
```

```
// need to use the fully-qualified-name which is in.Package2.Car inorder to use it if we are not using the import statement.
```

```
package in.Package1;

public class Car {
    public static void main(String[] args) {
        in.Package2.Car car_class_in_Package2 = new in.Package2.Car();
    }
}
```

File structure was

in/ > (Package1/ > Car.java)& (Package2/ > Car.java)

Packages avoids name collisions.

Packages can be used with the help of import statement.

Using import statement we can import classes, methods, packages and even interfaces.

Types of imports

1) Single_type import - imports a single class or interface from a package.

```
import java.util.Scanner;
```

2) On_demand import - imports all the classes and interfaces form a package.

```
import java.util.*;
```

There are 2 types of packages

1) Built-in packages

2) User-defined packages

Some of the built-in packages of java are java.util, java.lang, java.io etc.,

Built-in packages are those packages which we need not to import them explicitly.

8.4) Access modifiers

In Java, access modifiers are used to define the visibility and accessibility of classes, methods, and variables.

4 types of access modifiers

1) Public - Anything declared as public can be accessible form everywhere.

2) Protected - Anything declared as Protected can be accessed with in the same class and within the same package and also from the subclasses even if they are form different packages

3) Default (No modifier) a/o package-private -

Any thing that wasn't declared (default) can be accessed within the defining class and also across all the classes of that package, members outside the package can not access. That's why it is known as the package private.

From the above example,

Files from in.package2 can access one another, and files form in.Package1 can access one another if they haven't declared anything.

But a class from in.Package1 can not access a class from in.Package2 if it haven't declared public.

4) Private - restricts access in the defined class only. - Any thing that declared as private can be accessible within that class only.

Modifier	Access Level
private	Accessible only within the same class
default (<i>no modifier</i>)	Accessible within the same package only
protected	Accessible within the same package and by subclasses
public	Accessible from any other class

A class can be either public/ default but can not be private/ protected.

Access Modifier	Within Class	Within Package	Outside Package by Subclass Only	Outside Package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

As java is a case-sensitive language, it allows “Public” as the class name but not public.

Anything declared as private in Java can be accessed or modified *indirectly* using getter and setter methods, which are usually declared as public and it is a fundamental concept in encapsulation.

```
package in.Package1;
//car.java
public class Car {
    public String name;
    public String model;
    private int costOfPurchase;
    public Car(String name, String model, int costOfPurchase) {
        this.name = name;
        this.model = model;
        this.costOfPurchase = costOfPurchase;
    }
    public Car() {
    }
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        return sb.append("name= ").append(name).append("\nmodel: ").append(model).append("\n
        CostOf Purchase: ").append(costOfPurchase).toString();
    }
}
```

```
package in.Package1;
//AccessTest.java
public class AccessTest {
    public static void main(String[] args) {
        Car c1 = new Car();
        c1.name = "suzuki";
        c1.model = "swift";
        // c1.costOfPurchase = 5000; Xcan not access private variables directly.
        System.out.println(c1);
        Car c2 = new Car("Toyota", "Fortuner", 50000);
        System.out.println(c2);
    }
}
```

Output:

```
name= suzuki
model: swift
CostOf Purchase: 0
name= Toyota
model: Fortuner
CostOf Purchase: 50000
```

You can read the private field `costOfPurchase` using the overridden `toString()` method from another class. You cannot modify it directly from outside the `Car` class due to its private access.

Other than `toString()`, An inner class can access private members of the outer class in java.

```
public class Car {
    private int cost = 1000;

    class Engine {
        void showCost() {
            System.out.println("Cost: " + cost);
        }
    }
}
```

8.5) Getter and Setter methods

- Getters are methods used to retrieve the value of a private variable.
- Setters are methods used to modify the value of a private variable.
- They encapsulate the access logic and provide controlled access to private fields of a class.

```
// Car.java
public class Car {
    private String model;
    private int price;

    // Getter for model
    public String getModel() {
        return model;
    }
    // Setter for model
    public void setModel(String model) {
        this.model = model;
    }
    // Getter for price
    public int getPrice() {
        return price;
    }
    // Setter for price
    public void setPrice(int price) {
        if (price > 0) {
            this.price = price;
        }
    }
}
```

```
// Main.java
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();

        // Using setters to assign values
        myCar.setModel("Honda City");
        myCar.setPrice(12000);
        // Using getters to access values
        System.out.println("Model: " + myCar.getModel()); // Model: Honda City
        System.out.println("Price: $" + myCar.getPrice()); // Price: $12000
    }
}
```

If we use parameterized constructor then there will be no use of setter methods.

Getter and setter methods are typically declared as public, but they can also have default access if restricted to the same package.

CHALLENGES

- 1) Create 2 packages com.example.geometry and com.example.utils, in geometry package create classes like circle and rectangle and in utils create a class calculator to compute the area of these shapes.

```
package com.example.geometry;

public class Circle {
    public int radius;
    public Circle(int radius) {
        this.radius = radius;
    }
}
```

```
package com.example.geometry;

public class Triangle {
    public int base;
    public int height;
    public Triangle(int base, int height) {
        this.base = base;
        this.height = height;
    }
}
```

```
package com.example.utils;

import com.example.geometry.Circle;
import com.example.geometry.Triangle;

class Calculator {
    public static void main(String[] args) {
        Circle c1 = new Circle(5);
        Triangle t = new Triangle(5, 4);
        System.out.println("Area: " + Math.PI * Math.pow(c1.radius, 2));
        System.out.println("Area of Triangle: " + (0.5 * t.base * t.height));
    }
}
```

- 2) Define a BankAccount class with private attributes like accountNumber, accountHolderName, balance. Provide public methods to deposit and withdraw money ensuring that these methods don't allow illegal operations like withdrawing more money than the current balance.

```
public class BankAccount {
    private long accountNumber, balance;
    private String accountHolderName;

    BankAccount(long accountNumber, String accountHolderName) {
        this.accountHolderName = accountHolderName;
        this.accountNumber = accountNumber;
    }
}
```

```

    }
    public void balance() {
        System.out.println("Balance: " + this.balance);
    }
    public void deposit(long money) {
        if (money <= 0) {
            System.out.println("Invalid deposit");
        } else {
            this.balance += money;
            balance();
        }
    }
    public void withdraw(long money) {
        if (this.balance < money)
            System.out.println("Insufficient funds!");
        else {
            this.balance -= money;
            balance();
        }
    }
}
}

```

- 3) Define a class Employee with private attributes name,age,salary and public methods to get and set these attributes, and a package private method to display employee details. Create another class in the same package to test access to the displayEmployeeDetails method.

```

public class Employee {
    private String name;
    private int age;
    private double salary;

    Employee(String name, int age, double salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
    public double getSalary() {
        return salary;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public void setSalary(double salary) {
        this.salary = salary;
    }
    void displayEmployeeDetails() {
        System.out.printf("Name: %S\nAge: %d\nSalary: %d", getName(), getAge(), getSalary());
    }
}

```

```

public class TestGetSet {
    public static void main(String[] args) {
        Employee e = new Employee("Hemanth", 20, 50000);
        e.displayEmployeeDetails();
        e.setAge(20);
        e.displayEmployeeDetails();
    }
}

```

8.6) What is inheritance?

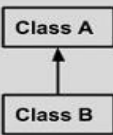
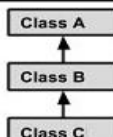
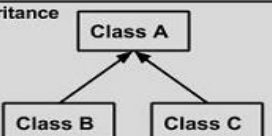
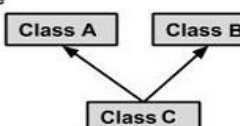
- Allows a new class(sub class/child class) to inherit features from an existing class(super class/parent class).
- Allows code reusability as sub classes can access the methods and variables of the super class.
- Protected access modifier was mostly used in inheritance to allow subclass access to super class members.

8.7) Types of inheritance

We can inherit the properties of one class at a time that's why multiple inheritance is not supported in java. The concept of interfaces was used to implement multiple inheritance.

4 types of inheritance are

- 1) Simple / single
- 2) Multiple (Not supported in java)
- 3) Multi-level
- 4) Hierarchical

Single Inheritance  <pre> graph BT B[Class B] --> A[Class A] </pre>	<pre> public class A { } public class B extends A { } </pre>
Multi Level Inheritance  <pre> graph BT C[Class C] --> B[Class B] B --> A[Class A] </pre>	<pre> public class A { } public class B extends A { } public class C extends B { } </pre>
Hierarchical Inheritance  <pre> graph BT B[Class B] --> A[Class A] C[Class C] --> A </pre>	<pre> public class A { } public class B extends A { } public class C extends A { } </pre>
Multiple Inheritance  <pre> graph BT A[Class A] --> C[Class C] B[Class B] --> C </pre>	<pre> public class A { } public class B { } public class C extends A,B { } // Java does not support multiple inheritance </pre>

8.8) Object class

In Java, the Object class is the superclass of all classes. Every class in Java implicitly extends Object, unless it explicitly extends another class. Since Java does not support multiple inheritance with classes, a class can extend only one other class at a time.

In any type of inheritance, such as single inheritance, if class B extends class A, then class A implicitly extends the Object class. This means that all classes in Java—either directly or through a chain of inheritance—ultimately inherit from the Object class.

Some of the methods of the object class are

- toString()
- equals()
- hashCode()
- getClass()

8.9) Equals and hashCode

hashCode() is not designed to determine full equality, but to narrow down comparisons. It acts as a fast pre-check: if two objects have different hash codes, they are definitely not equal. If they have the same hash code, then we use equals() to do a full comparison.

If a class has many fields (say, 100), and you want to compare two objects using equals(), then comparing each of those 100 fields can be expensive in terms of time and resources — especially in large data structures like HashSet or HashMap.

That's where hashCode() provides a performance optimization, two objects can have the same hash code if their content is the same even if they belong to different classes, especially if they override the hashCode() method.

Hashcode is of type int.

If the hashcodes of both the objects are same then equals() may return true/ false.

If 2 objects returns true on equals() method then their hashcodes must be equal.

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



```

public Employee(String name, int age, String id) {
    this.name = name;
    this.age = age;
    this.id = id;
}
public String getName() {
    return name;
}
public int getAge() {
    return age;
}
public String getId() {
    return id;
}
public void setName(String name) {
    this.name = name;
}
public void setAge(int age) {
    this.age = age;
}
public void setId(String id) {
    this.id = id;
}
@Override
public String toString() {
    return "EqualsCheck [name=" + name + ", age=" + age + ", id=" + id + "]";
}
}

```

```

public class EqualsCheck {
    public static void main(String[] args) {
        Employee e1 = new Employee("Hemanth", 20, "004738");
        Employee e2 = new Employee("Hemanth", 20, "004738");
        System.out.println(e1 == e2); // false -> as == compares the references
        System.out.println(e1.equals(e2)); // false -> as equals method here also compares the
            references, so we need to override equals method
    }
}
// toString method in String class doesn't compares the references as it overrides equals() by
    default.

```

After overriding the equals method,

instanceof is a keyword that was used to check if the object is the instance of particular class / subclass.

```

public class Employee {
    private String name;
    private int age;
    private String id;

    public Employee(String name, int age, String id) {
        this.name = name;
        this.age = age;
        this.id = id;
    }
    public String getName() {

```

```

        return name;
    }
    public int getAge() {
        return age;
    }
    public String getId() {
        return id;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public void setId(String id) {
        this.id = id;
    }
    @Override
    public String toString() {
        return "EqualsCheck [name=" + name + ", age=" + age + ", id=" + id + "]";
    }
    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Employee) {
            Employee e = (Employee) obj;
            return e.name.equals(name) && e.age == age && e.id.equals(id);
        } else {
            return false;
        }
    }
}

```

```

public class EqualsCheck {
    public static void main(String[] args) {
        Employee e1 = new Employee("Hemanth", 20, "004738");
        Employee e2 = new Employee("Hemanth", 20, "004738");
        System.out.println(e1 == e2); // false -> as == compares the references
        System.out.println(e1.equals(e2)); // true -> compares the content
    }
}

```

Below is the Summary of equals-hashcode contract,

- If equals is true, hashCode must be equal. (REQUIRED)
- If hashCodes differ, equals must be false. (EXPECTED)
- If hashCodes are the same, equals can be either true or false. (DEPENDS)

Below is the system generated methods of hashcodes and equal methods that are done through

Right click > source action > generate hashCode() and equals()

```

//Employee.java
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;

```

```

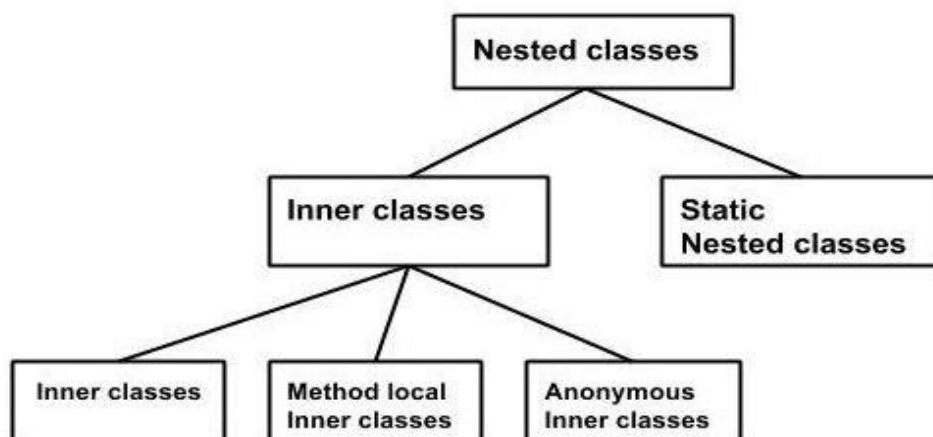
        result = prime * result + ((name == null) ? 0 : name.hashCode());
        result = prime * result + age;
        result = prime * result + ((id == null) ? 0 : id.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Employee other = (Employee) obj;
        if (name == null) {
            if (other.name != null)
                return false;
        } else if (!name.equals(other.name))
            return false;
        if (age != other.age)
            return false;
        if (id == null) {
            if (other.id != null)
                return false;
        } else if (!id.equals(other.id))
            return false;
        return true;
    }
}

```

8.10) Nested and Inner classes

Nested classes are defined within another class .



➤ Static Nested Classes

Act like static members of the outer class.

Can only access static members of the outer class directly.

Do not require an instance of the outer class to be created.

➤ **Inner (Non-static) Classes**

Belong to an instance of the outer class.

Can access all members of the outer class, including private members.

Require an instance of the outer class to be instantiated.

➤ **Local Inner Classes**

Defined inside a method or block.

Scope is limited to that method/block.

Cannot be accessed outside their defined scope.

➤ **Anonymous Inner Classes**

Classes without a name, created for one-time use (usually to implement interfaces or abstract classes).

Defined and instantiated in a single expression.

➤ **Top-Level Classes**

Cannot be declared private or protected.

Can only be public or package-private (default).

Inner classes, however, can be private, protected, default, or public.

a non-static inner class cannot be accessed without creating an object of the outer class.

```
public class Outer {
    class Inner {
        void show() {
            System.out.println("Non-static inner class");
        }
    }

    public static void main(String[] args) {
        Outer outerobj = new Outer();
        Outer.Inner inner = outerobj.new Inner();
        //or Outer.Inner inner=new Outer().new Inner();
        inner.show();
    }
}
```

Static Nested Class Does not require an instance of the outer class. Only needs to be accessed using the outer class name.

```
public class Outer {
    static class Inner {
        void show() {
            System.out.println("Static nested class");
        }
    }

    public static void main(String[] args) {
        Outer.Inner inner = new Outer.Inner(); // ✔
        inner.show();
    }
}
```

Local Inner Class (Defined in a method/block) are accessible only within the block where it is defined.

```
public class Outer {
    void method() {
        class Inner {
            void show() {
                System.out.println("Local inner class");
            }
        }

        Inner inner = new Inner(); // ✔️Only accessible here
        inner.show();
    }
    public static void main(String[] args) {
        new Outer().method();
    }
}
```

Upcasting allows run-time polymorphism → ParentClass obj=new ChildClass();

CHALLENGES

1. Start with a base class LibraryItem that has itemID, title, author and methods checkOut(), returnItem(). create sub class like book, magazine, DVD, each inheriting from LibraryItem. Add unique attributes to each subclass like ISBN for book, issueNumber for magazine and duration for DVD.

```
package in.Package2;

public class LibraryItem {
    private int itemID;
    private String title;
    private String author;
    public LibraryItem(int itemID, String title, String author) {
        this.itemID = itemID;
        this.title = title;
        this.author = author;
    }
    public int getItemID() {
        return itemID;
    }
    public String getTitle() {
        return title;
    }
    public String getAuthor() {
        return author;
    }
}
```

```
    public void setItemID(int itemID) {
        this.itemID = itemID;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public void setAuthor(String author) {
        this.author = author;
    }
}
```

```

@Override
public String toString() {
    return "LibraryItem [itemID=" + itemID + ", title=" + title + ", author=" + author +
    "]\n";
}
public void checkOut() {
    System.out.println(this.toString() + " Checked out!");
}
public void returnItem() {
    System.out.println(this.toString() + " returned successfully");
}
}

```

```

package in.Package2;

public class Book extends LibraryItem {
    public int ISBN;
    public Book(int itemID, String title, String author, int isbn) {
        super(itemID, title, author);
        ISBN = isbn;
    }
    public int getISBN() {
        return ISBN;
    }
    @Override
    public String toString() {
        return super.toString() + "Book [ISBN=" + ISBN + "]\n";
    }
    public void setISBN(int isbn) {
        ISBN = isbn;
    }
    public static void main(String[] args) {
        Book b1 = new Book(4738, "Noting", "who knows", 143);
        System.out.println(b1);
    }
}

```

```

package in.Package2;

public class Magazine extends LibraryItem {
    private int issueNumber;
    public Magazine(int itemID, String title, String author, int issueNumber) {
        super(itemID, title, author);
        this.issueNumber = issueNumber;
    }
    public int getIssueNumber() {
        return issueNumber;
    }
    @Override
    public String toString() {
        return super.toString() + "Magazine [issueNumber=" + issueNumber + "]\n";
    }
    public void setIssueNumber(int issueNumber) {
        this.issueNumber = issueNumber;
    }
    public static void main(String[] args) {
        Magazine m1 = new Magazine(123, "vov", "vvit", 25);
        m1.checkOut();
    }
}

```

```

package in.Package2;

public class Dvd extends LibraryItem {
    private int duration;
    public Dvd(int itemID, String title, String author, int duration) {
        super(itemID, title, author);
        this.duration = duration;
    }
    public static void main(String[] args) {
        Dvd d1 = new Dvd(154, "my project", "me", 50);
        d1.returnItem();
    }
    public int getDuration() {
        return duration;
    }
    @Override
    public String toString() {
        return super.toString() + "Dvd [duration=" + duration + "]";
    }
    public void setDuration(int duration) {
        this.duration = duration;
    }
}

```

2. Create a person class with name and age as attributes and override equals to compare person objects based on their attributes and override hashCode() consistent with the definition of equals.

```

package in.Package2;

public class Person {
    public String name;
    public int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((name == null) ? 0 : name.hashCode());
        result = prime * result + age;
        return result;
    }
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Person other = (Person) obj;
        if (name == null) {
            if (other.name != null)
                return false;
        } else if (!name.equals(other.name))
            return false;
        if (age != other.age)

```

```

        return false;
    }
    return true;
}

```

3. Create a class ArrayOperations with a static nested class Statistics. Statistics should have methods like mean(), median() which operates on an array.

```

package in.Package2;

import java.util.Arrays;

public class ArrayOperations {

    static class Statistics {

        public void mean(int[] arr) {
            double sum = 0;
            for (int i : arr)
                sum += i;
            System.out.println("Mean: " + (sum / arr.length));
        }

        public void median(int[] arr) {
            Arrays.sort(arr);
            double median;
            int mid = arr.length / 2;
            if (arr.length % 2 == 0) {
                median = (arr[mid - 1] + arr[mid]) / 2.0;
            } else {
                median = arr[mid];
            }
            System.out.println("Median: " + median);
        }
    }

    public static void main(String[] args) {
        ArrayOperations.Statistics obj = new ArrayOperations.Statistics();
        obj.mean(new int[] {3,1,4,2});
        obj.median(new int[] {3,1,4,2});
    }
}

```

KEY POINTS

- Parent classes are known as general classes, whereas child classes are considered specific classes.
- The access modifier of the object class can be either public or default, but it cannot be private or protected as it is not an inner class.
- A static nested class cannot access instance variables of the outer class.
- Override both the toString() and hashCode() methods to maintain consistency.