Scenario based questions

1. **A university is developing a student management system where each student has attributes like name, roll number, and courses enrolled. The system needs to allow students to perform actions such as enrolling in courses and checking grades.**
   Question: Explain the relationship between objects, classes, data members, and methods in OOP with reference to this system.

Ans.

In Object-Oriented Programming (OOP), classes, objects, data members, and methods are the core building blocks that help organize and model real-world entities like students in a university system.

- **Class: A class is a blueprint for creating student objects. For example, a class named Student can be created that defines the structure and behaviour of a student. It includes attributes such as name,** roll_number**, and** courses_enrolled**, and actions like** enroll_in_course() **or** check_grades().
- **Object: An object is an instance of a class. For example, when a new student joins the university, a new object of the Student class is created. This object holds the specific details of that student like their name, roll number, and the list of courses they are enrolled in.**
- **Data Members: These are the variables inside the class that hold data specific to each student. In this system, data members can include name,** roll_number**, and** courses_enrolled**.**
- **Methods: These are functions defined inside the class that represent the behaviour of the student. For instance,** enroll_in_course(course_name**) can be a method to allow a student to enroll in a course, and** check_grades() **can be a method to allow the student to view their grades.**

Relationship:

- **A class groups both data members (attributes like** name **and** courses_enrolled**) and methods (like** enroll_in_course() **and** check_grades()**).**
- **An object is a specific student created using the class. It uses the structure of the class to store actual student data and perform actions using the methods defined in the class.**

Thus, in the student management system, OOP allows the university to efficiently model students as objects, define their structure and behaviour in a class, and manage each student's information and actions in a clean, organized way.

- **Code:**

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

// Class Definition
class Student {
private:
    string name;          // Data Member
    string rollNumber;    // Data Member
    vector<string> coursesEnrolled; // Data Member

public:
    // Constructor
    Student(string studentName, string studentRoll) {
        name = studentName;
        rollNumber = studentRoll;
    }

    // Method to enroll in a course
    void enrollInCourse(string course) {
        coursesEnrolled.push_back(course);
        cout << name << " has enrolled in " << course << "." << endl;
    }
```

```
    // Method to display student details
    void displayDetails() {
        cout << "Name: " << name << ", Roll Number: " << rollNumber << endl;
        cout << "Courses Enrolled: ";
        for (string course : coursesEnrolled) {
            cout << course << " ";
        }
        cout << endl;
    }
};

// Main Function
int main() {
    // Creating an object (student)
    Student student1("Alice", "U123");

    // Using methods
    student1.enrollInCourse("Mathematics");
    student1.enrollInCourse("Physics");
    student1.displayDetails();

    return 0;
}
```

---

2. **A company is designing an employee management system. Some employee details, such as salary and bank details, should be hidden from other employees but accessible to HR. Other general details, like name and department, should be visible to everyone.**
   Question: How does the private modifier differ from the protected modifier in OOP, and how can they be used to implement data security in this scenario?

Ans.

Answer:

In Object-Oriented Programming (OOP), access specifiers such as private and protected are used to control the accessibility of class members, and they play a crucial role in implementing data security in systems like an employee management system.

Private Modifier:

- **Members declared as private can only be accessed within the class itself.**
- **They cannot be accessed directly by any object, function, or derived class.**
- **To access private members, public getter and setter methods are typically used.**
- **This is useful for sensitive data like salary and bank details, which should be hidden from other employees but accessible to authorized personnel like HR through controlled functions.**

Protected Modifier:

- **Members declared as protected are similar to private but can also be accessed by derived classes.**
- **This allows for limited visibility where the data is hidden from the outside but available to specific roles or subclasses, such as an HR class that inherits from the Employee class.**
- **This is suitable for general information like name and department, which may not be completely sensitive but should still not be publicly exposed.**

Application in the Employee Management System:

- **Use private for salary and bank details to fully encapsulate sensitive data.**

- **Use protected for name and department so HR (as a derived class) can access them, while keeping them hidden from general access.**
- **Public methods act as interfaces to allow or restrict access to these members in a controlled manner.**

- **C++ Code Example:**

```
#include <iostream>
#include <string>
using namespace std;

// Base class: Employee
class Employee {
private:
    double salary;          // Private: sensitive data
    string bankDetails;     // Private: sensitive data

protected:
    string name;            // Protected: visible to HR
    string department;      // Protected: visible to HR

public:
    // Setter for general details
    void setGeneralDetails(string empName, string dept) {
        name = empName;
        department = dept;
    }

    // Setter for private details
```

```
    void setPrivateDetails(double empSalary, string bank) {
        salary = empSalary;
        bankDetails = bank;
    }

    // Public method to display general info
    void displayPublicInfo() {
        cout << "Employee Name: " << name << endl;
        cout << "Department: " << department << endl;
    }

    // Getter for sensitive data (used by HR)
    double getSalary() {
        return salary;
    }

    string getBankDetails() {
        return bankDetails;
    }
};

// Derived class: HR
class HR : public Employee {
public:
    void viewEmployeeDetails(Employee &emp) {
        emp.displayPublicInfo();
        cout << "[HR Access] Salary: " << emp.getSalary() << endl;
        cout << "[HR Access] Bank Details: " << emp.getBankDetails() << endl;
```

```
    }
};

int main() {
    Employee emp1;
    emp1.setGeneralDetails("Valmik Sir", "Engineering");
    emp1.setPrivateDetails(85000.75, "XYZ Bank, A/C 123456789");

    cout << "--- Public View ---" << endl;
    emp1.displayPublicInfo();

    cout << "\n--- HR View ---" << endl;
    HR hrManager;
    hrManager.viewEmployeeDetails(emp1);

    return 0;
}
```

Summary:

- **The private modifier ensures strict data hiding and is ideal for highly sensitive information.**
- **The protected modifier allows access within derived classes, enabling controlled internal access.**
- **Together, they help build secure, well-structured, and role-aware systems by encapsulating data and limiting access to appropriate classes or functions.**

---

3. **An e-commerce application allows users to apply different discount coupons. Some coupons apply a flat discount, while others apply a percentage-based discount. The system has a method applyDiscount() that behaves differently based on the type of coupon.**
   Question: Explain the difference between compile-time polymorphism and run-time polymorphism with reference to this system.

Ans.

In the context of the e-commerce application where the applyDiscount() method behaves differently based on the type of coupon (flat discount or percentage-based discount), we can understand the difference between compile-time polymorphism and run-time polymorphism using the concepts of method overloading and method overriding in Java:

Compile-time Polymorphism (Method Overloading)

- **Definition: Method overloading is a type of compile-time polymorphism where multiple methods have the same name but different parameter lists.**
- **Application in E-commerce System: Suppose the system has multiple applyDiscount() methods in the same class, but with different parameters like:**

```
public void applyDiscount(double flatAmount) {
   // Apply flat discount
}

public void applyDiscount(double percentage, double totalAmount) {
   // Apply percentage-based discount
}
```

Here, the method to be called is determined at compile-time based on the method signature used. This improves code readability and allows flexibility within the same class.

- **Key Characteristics:**
  - **Happens within the same class.**
  - **Methods must have the same name but different parameters.**
  - **Inheritance is not required.**

- **Example use: Overloaded versions of applyDiscount() for different discount types or parameters.**

Run-time Polymorphism (Method Overriding)

- **Definition: Method overriding is a type of run-time polymorphism where a subclass provides a specific implementation of a method that is already defined in its superclass.**
- **Application in E-commerce System: Suppose there is a base class Coupon and two subclasses FlatCoupon and PercentageCoupon, each overriding the applyDiscount() method:**

```
class Coupon {
    public void applyDiscount() {
        // Generic implementation
    }
}

class FlatCoupon extends Coupon {
    @Override
    public void applyDiscount() {
        // Apply flat discount logic
    }
}

class PercentageCoupon extends Coupon {
    @Override
    public void applyDiscount() {
        // Apply percentage discount logic
    }
}
```

Here, the actual method to be called is determined at run-time depending on the object type (FlatCoupon or PercentageCoupon). This allows behavior to be dynamically selected at execution time.

- **Key Characteristics:**
  - **Involves inheritance between classes.**
  - **Methods must have the same name and parameter list.**
  - **Return types must be the same or co-variant.**
  - **Example use: Different coupon types overriding a common applyDiscount() method in their own way.**

Summary Table:

Feature	Method Overloading (Compile-time Polymorphism)	Method Overriding (Run-time Polymorphism)
Method Signature	Same name, different parameters	Same name, same parameters
Binding Time	Compile-time	Run-time
Inheritance	Not required	Required
Purpose	Code readability & flexibility	Custom behavior in subclass
Example in E-commerce	Overloaded applyDiscount() for flat vs. percentage with different parameters	Overridden applyDiscount() in FlatCoupon and PercentageCoupon subclasses

---

So in this e-commerce system, compile-time polymorphism allows different versions of applyDiscount() **to exist in one class, while run-time polymorphism allows dynamic execution of the correct** applyDiscount() **logic based on the coupon type at runtime.**

---

5. A vehicle rental service provides different rental price calculations based on vehicle type. Cars have a base price per day, trucks have an additional weight-based charge, and motorcycles have a mileage-based price.

Question: If a Vehicle class has an abstract method calculate_rental_price(), **how would different vehicle types (Car, Truck, Motorcycle) implement it?**

Ans.

```
public class Main {
    public static void main(String[] args) {
        Vehicle car = new Car();
        Vehicle truck = new Truck();
        Vehicle motorcycle = new Motorcycle();

        System.out.println("Car Rental (3 days): " + car.calculateRentalPrice(3));
        System.out.println("Truck Rental (2 days): " + truck.calculateRentalPrice(2));
        System.out.println("Motorcycle Rental (4 days): " + motorcycle.calculateRentalPrice(4));
    }
}

// Abstract class
abstract class Vehicle {
    abstract double calculateRentalPrice(int days);
}

// Car class
class Car extends Vehicle {
    @Override
    double calculateRentalPrice(int days) {
        return days * 500;
    }
```

```
}

// Truck class
class Truck extends Vehicle {
    @Override
    double calculateRentalPrice(int days) {
        return days * 1000;
    }
}

// Motorcycle class
class Motorcycle extends Vehicle {
    @Override
    double calculateRentalPrice(int days) {
        return days * 300;
    }
}
```

---

4. **A banking application needs to handle multiple types of exceptions, such as** InsufficientFundsException, InvalidAccountException**, and** NetworkFailureException**. These exceptions can occur simultaneously when processing transactions.**
   Question: How can you handle multiple exceptions in Java to ensure smooth transaction processing?

Ans.

How to handle multiple exceptions in Java:

1. Use multiple catch blocks

Java allows you to write multiple catch blocks after a try block to handle different exception types individually.

2. Order from specific to general

Catch the most specific exceptions first, and catch the general Exception type last.

3. Use a finally block

A finally block is used to perform cleanup actions like closing resources or logging, and it executes regardless of whether an exception occurred.

4. Use custom exception classes

Define own exception types.

- **Code:**

```
// Main class with main method
public class Main {
    public static void main(String[] args) {
        try {
            // Simulate a transaction
            processTransaction("ACC123", 15000); // Try changing amount or account for different exceptions
        } catch (InvalidAccountException e) {
            System.out.println("Invalid account: " + e.getMessage());
        } catch (InsufficientFundsException e) {
            System.out.println("Insufficient funds: " + e.getMessage());
```

```
        } catch (NetworkFailureException e) {
            System.out.println("Network failure: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("Unexpected error: " + e.getMessage());
        } finally {
            System.out.println("Transaction attempt completed.");
        }
    }

    // Simulate transaction logic
    public static void processTransaction(String accountNumber, double amount)
            throws InvalidAccountException, InsufficientFundsException,
NetworkFailureException {

        if (accountNumber == null || accountNumber.isEmpty()) {
            throw new InvalidAccountException("Account number is empty.");
        }

        if (amount > 10000) {
            throw new InsufficientFundsException("Balance is insufficient for the requested
amount.");
        }

        if (Math.random() < 0.2) {
            throw new NetworkFailureException("Unable to connect to the server.");
        }

        System.out.println("Transaction successful for account: " + accountNumber);
    }
```

```
}

// Custom exception: Invalid account
class InvalidAccountException extends Exception {
    public InvalidAccountException(String message) {
        super(message);
    }
}

// Custom exception: Insufficient funds
class InsufficientFundsException extends Exception {
    public InsufficientFundsException(String message) {
        super(message);
    }
}

// Custom exception: Network failure
class NetworkFailureException extends Exception {
    public NetworkFailureException(String message) {
        super(message);
    }
}
```

---

5. **Identify and Fix the Error in the Given Code**
   // Interface Animal interface Animal { void makeSound(); } // Interface Cat interface Cat { void purr(); } 5) A banking application needs to handle multiple types of exceptions, such as InsufficientFundsException, InvalidAccountException, and NetworkFailureException. These exceptions can occur simultaneously when processing transactions. Question: How can you handle multiple exceptions in Java to ensure smooth transaction processing? // Dog class implementing both interfaces class Dog extends Animal, Cat { public void makeSound() { System.out.println("Dog barks!"); } public void purr() { } } System.out.println("Dog cannot purr, but method must be implemented!"); // Main class public class Main { public static void main(String[] args) { Dog dog = new Dog(); dog.makeSound(); dog.purr(); } }

Ans.

The provided code has a few syntax and structural errors.

- **Errors :**

1. **class Dog extends Animal, Cat is invalid syntax.**

   Reason: In Java, a class can only extend one class, but it can implement multiple interfaces using the implements keyword.

   Fix: Use implements Animal, Cat

2. **The System.out.println("Dog cannot purr...") line is placed outside of any method, which is not allowed in Java.**

   Fix: Move it inside a method (e.g., purr())

- **Code:**

```
// Interface Animal
interface Animal {
    void makeSound();
}

// Interface Cat
interface Cat {
```

```
    void purr();
}

// Dog class implementing both interfaces
class Dog implements Animal, Cat {
    public void makeSound() {
        System.out.println("Dog barks!");
    }

    public void purr() {
        System.out.println("Dog cannot purr, but method must be implemented!");
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.makeSound();
        dog.purr();
    }
}
```

6. **A flight booking system allows customers to enter their age while booking a ticket. If the user enters a negative number, the system should throw an exception. However, missing a required field should cause a different exception. Question: Differentiate between checked and unchecked exceptions in Java and explain how they can be used to handle such cases in the flight booking system.**

Ans.

In Java, exceptions are categorized into checked and unchecked exceptions based on how they are handled at compile time.

---

1. Checked Exceptions

- **Definition: Checked exceptions are checked at compile-time. The compiler forces the programmer to handle them using try-catch or declare them using throws.**
- **Use in Flight Booking System:**
  A missing required field (like name or passport number) is a recoverable and predictable error, so it should be handled as a checked exception.

Example:

```
class MissingFieldException extends Exception {
   public MissingFieldException(String message) {
      super(message);
   }
}
```

---

2. Unchecked Exceptions

- **Definition: Unchecked exceptions are not checked at compile-time. They are subclasses of RuntimeException and usually occur due to programming errors (e.g., logic issues).**
- **Use in Flight Booking System:**
  Entering a negative age is a logic error, which can be treated as an unchecked exception because it should not happen under normal conditions.

Example:

```
class InvalidAgeException extends RuntimeException {
   public InvalidAgeException(String message) {
```

```
        super(message);
    }
}
```

- **Code:**

```
// Custom checked exception for missing required fields
class MissingFieldException extends Exception {
    public MissingFieldException(String message) {
        super(message);
    }
}

// Custom unchecked exception for invalid input like negative age
class InvalidAgeException extends RuntimeException {
    public InvalidAgeException(String message) {
        super(message);
    }
}

public class Main {

    // Flight booking method
    public static void bookTicket(String name, int age) throws MissingFieldException {
        if (name == null || name.isEmpty()) {
            throw new MissingFieldException("Name is required.");
        }

        if (age < 0) {
```

```
            throw new InvalidAgeException("Age cannot be negative.");
        }

        System.out.println("Ticket booked for " + name + ", Age: " + age);
    }

    // Main method
    public static void main(String[] args) {
        try {
            // Try different test cases:
            // bookTicket("", 25);     // Will throw MissingFieldException
            // bookTicket("Alice", -5); // Will throw InvalidAgeException
            bookTicket("Alice", 30);   // Successful booking

        } catch (MissingFieldException e) {
            System.out.println("Error: " + e.getMessage());
        } catch (InvalidAgeException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Difference:

Feature	**Checked Exception**	**Unchecked Exception**
Checked at compile time	**Yes**	**No**
Inheritance	**Extends Exception**	**Extends** RuntimeException

Feature	Checked Exception	Unchecked Exception
Example	IOException, MissingFieldException	ArithmeticException, InvalidAgeException
Flight System Use	**Missing required input**	**Invalid logical input (like negative age)**

---

7. **What will happen if an exception occurs and there is no catch block?**

Ans.

If an exception occurs and there is no catch block to handle it, the following will happen in Java:

---

1. The program will terminate abnormally

- **Java will not know how to handle the exception, so it will stop program execution at the point where the exception occurred.**

---

2. Java will throw a default exception message

- **The Java Runtime Environment (JRE) will print a stack trace showing:**
  - **Type of exception**
  - **Error message (if any)**
  - **Line number where the error occurred**
  - **The call stack leading to the error**

- **Example Code:**

```
public class Main {
   public static void main(String[] args) {
      int a = 10;
      int b = 0;
      int result = a / b;  // This will throw ArithmeticException
```

```
        System.out.println("Result: " + result); // This line won't run
    }
}
```

---

8. **A software development company is working on an employee attendance system. Each employee has a unique ID and login time. The system initializes employee details when they log in and provides methods to retrieve and update their attendance data.**
   Question: What is the difference between a method and a constructor in OOP? Explain with reference to the employee attendance system.

Ans.

Here's a precise and well-structured answer to your question, with reference to the employee attendance system and using the provided comparison chart:

Q: What is the difference between a method and a constructor in OOP? Explain with reference to the employee attendance system.

In Object-Oriented Programming (OOP), constructors and methods serve different purposes. Here's a detailed comparison using the employee attendance system as context:

Constructors	Methods
A constructor is a block of code used to initialize a newly created object.	A method is a block of code that performs operations and may return a value.
It is called automatically when an object is created using new.	It must be explicitly called by the programmer.
It has no return type, not even void.	It must have a return type (e.g., void, int, String).
The name must match the class name.	The name can be any valid identifier.

Constructors	Methods
It sets up the object's initial state.	**It performs actions on an already created object.**
Constructors cannot be inherited.	**Methods can be inherited and overridden in subclasses.**

- **Example in the Employee Attendance System:**

```
// Employee class
class Employee {
    String employeeId;
    String loginTime;

    // Constructor - called automatically when an object is created
    public Employee(String id, String loginTime) {
        this.employeeId = id;
        this.loginTime = loginTime;
    }

    // Method - retrieves login details
    public String getLoginDetails() {
        return "Employee ID: " + employeeId + ", Login Time: " + loginTime;
    }

    // Method - updates login time
    public void updateLoginTime(String newLoginTime) {
        this.loginTime = newLoginTime;
    }
}
```

```
// Main class
public class Main {
    public static void main(String[] args) {
        // Creating an employee object using the constructor
        Employee emp = new Employee("EMP123", "09:00 AM");

        // Display initial login details
        System.out.println(emp.getLoginDetails());

        // Update login time using a method
        emp.updateLoginTime("09:30 AM");

        // Display updated login details
        System.out.println(emp.getLoginDetails());
    }
}
```

- **The constructor** Employee(String id, String loginTime**) is automatically called when an employee logs in to create a new Employee object with their ID and login time.**
- **The methods like** getLoginDetails() **and** updateLoginTime() **are used after the object is created to retrieve or update attendance data.**

In the attendance system, constructors are used to initialize each employee's data at login, while methods are used to manage and manipulate that data afterward. Both are essential but serve distinct roles in OOP design.

9. **A bank application stores customer account details securely. The account balance should not be directly accessible but can be retrieved or updated using specific methods. Question: Explain how getter and setter methods contribute to encapsulation. Provide a code example demonstrating how they can be used in this banking scenario.**

Ans.

Encapsulation in Java means wrapping data (variables) and code (methods) together as a single unit and restricting direct access to some components.

In a bank application, account details like balance should be private and only accessible via getter and setter methods. This protects the data from unauthorized or incorrect access.

Role of Getter and Setter Methods:

Method Type	Purpose
getter	**Safely retrieve private data**
setter	**Safely update private data with validation logic**

- **Java Code:**

```
// BankAccount class with encapsulation
class BankAccount {
    private String accountNumber;
    private double balance;

    // Constructor
    public BankAccount(String accountNumber, double initialBalance) {
        this.accountNumber = accountNumber;
        this.balance = initialBalance;
    }

    // Getter method for balance
```

```
    public double getBalance() {
        return balance;
    }

    // Setter method for balance (with validation)
    public void setBalance(double newBalance) {
        if (newBalance >= 0) {
            balance = newBalance;
        } else {
            System.out.println("Invalid balance. Cannot set a negative value.");
        }
    }

    // Method to display account info
    public void displayAccountInfo() {
        System.out.println("Account: " + accountNumber + ", Balance: " + balance);
    }
}

// Main class to test
public class Main {
    public static void main(String[] args) {
        // Create a new bank account
        BankAccount acc = new BankAccount("123456789", 1000.0);

        // Display initial account info
        acc.displayAccountInfo();
```

```
        // Try to update balance using setter
        acc.setBalance(1500.0);  // Valid update
        System.out.println("Updated Balance: " + acc.getBalance());

        acc.setBalance(-500);    // Invalid update
        System.out.println("After invalid update attempt: " + acc.getBalance());
    }
}
```

Explanation:

- **The balance field is private, so it cannot be accessed directly from outside the class.**
- **The getBalance() method returns the current balance.**
- **The setBalance() method updates the balance, but only if the new value is valid (non-negative).**
- **This ensures secure and controlled access to sensitive data, which is the core idea of encapsulation.**

---

10. A university wants to model its staff hierarchy. A Professor is a type of Employee, and a Department has multiple Professors. Question: Describe the "is-a" and "has-a" relationships in OOP with reference to this scenario. Provide an example for each.

Ans.

🔹 **"is-a" Relationship (Inheritance)**

- **The "is-a" relationship in OOP refers to inheritance.**
- **It means one class is a specialized version of another.**
- **In your scenario:**
  → A Professor *is a* type of Employee

Example:

```
// Superclass
class Employee {
    String name;
    int id;

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

    void displayInfo() {
        System.out.println("Employee ID: " + id + ", Name: " + name);
    }
}

// Subclass - Inherits from Employee
class Professor extends Employee {
    String subject;

    Professor(String name, int id, String subject) {
        super(name, id);
        this.subject = subject;
    }

    void teach() {
        System.out.println(name + " teaches " + subject);
    }
}
```

🔸 **In this example, Professor extends Employee, so Professor is-a Employee.**

🔹 **"has-a" Relationship (Composition/Aggregation)**

- **The "has-a" relationship represents composition or aggregation.**
- **One class contains or uses another class.**
- **In your scenario:**
  → A Department *has-a* list of Professors

Example:

```
import java.util.ArrayList;
import java.util.List;

class Department {
    String name;
    List<Professor> professors;

    Department(String name) {
        this.name = name;
        this.professors = new ArrayList<>();
    }

    void addProfessor(Professor p) {
        professors.add(p);
    }

    void listProfessors() {
        System.out.println("Department: " + name);
        for (Professor p : professors) {
            p.displayInfo();
```

```
            p.teach();
        }
    }
}
```

🔸 **In this example, Department has-a list of Professor objects, forming a has-a relationship.**

- **Code:**

```
import java.util.ArrayList;
import java.util.List;

// Superclass: Employee
class Employee {
    String name;
    int id;

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

    void displayInfo() {
        System.out.println("Employee ID: " + id + ", Name: " + name);
    }
}

// Subclass: Professor is-a Employee
class Professor extends Employee {
```

```
    String subject;

    Professor(String name, int id, String subject) {
        super(name, id);
        this.subject = subject;
    }

    void teach() {
        System.out.println(name + " teaches " + subject);
    }
}

// Department class has-a list of Professors
class Department {
    String name;
    List<Professor> professors;

    Department(String name) {
        this.name = name;
        this.professors = new ArrayList<>();
    }

    void addProfessor(Professor p) {
        professors.add(p);
    }

    void listProfessors() {
        System.out.println("Department: " + name);
```

```
        for (Professor p : professors) {
            p.displayInfo();
            p.teach();
            System.out.println();
        }
    }
}

// Main class
public class Main {
    public static void main(String[] args) {
        // Creating professors
        Professor prof1 = new Professor("Dr. Smith", 101, "Physics");
        Professor prof2 = new Professor("Dr. Alice", 102, "Mathematics");

        // Creating department and adding professors
        Department scienceDept = new Department("Science");
        scienceDept.addProfessor(prof1);
        scienceDept.addProfessor(prof2);

        // Listing professors
        scienceDept.listProfessors();
    }
}
```

- **Summary:**

Relationship	Meaning	Example
is-a	Inheritance	Professor is-a Employee
has-a	Composition	Department has-a Professor(s)

---

11. **A game development team is using both C++ and Java to design a game engine. They notice that Java requires interface and implements keywords, while C++ does not.**
    Question: Why does C++ not require the interface or implements keyword like Java? Explain with a code example.

Ans.

🔹 **Short Answer:**

C++ does not require the interface or implements keywords because it uses abstract classes to define interfaces.
In C++, an abstract class (a class with at least one pure virtual function) can act as an interface.
Any class that inherits from it and overrides the pure virtual functions is considered to have implemented the interface.

In contrast, Java explicitly uses interface and implements keywords to define and use interfaces.

- **C++ vs Java Interface Example:**

Java Example:

```
interface GameEntity {
   void update();
}
class Player implements GameEntity {
   public void update() {
```

```
        System.out.println("Player updated.");
    }
}
```

- **Equivalent C++ Example:**

```
#include <iostream>
using namespace std;

// Abstract class acting as an interface
class GameEntity {
public:
    virtual void update() = 0;  // Pure virtual function
    virtual ~GameEntity() {}    // Virtual destructor
};

// Class implementing the interface
class Player : public GameEntity {
public:
    void update() override {
        cout << "Player updated." << endl;
    }
};

int main() {
    GameEntity* entity = new Player();
    entity->update();
    delete entity;
    return 0;
}
```

- **Explanation:**

Java	C++
Uses interface and implements	Uses abstract class with pure virtual function
Syntax is explicit	Syntax is implicit (via = 0 pure virtual)
Interfaces can't have fields	Abstract class can have fields
Multiple inheritance via interfaces is allowed	Multiple inheritance of abstract classes is allowed

---

Conclusion:

C++ doesn't need special keywords like interface or implements because its powerful inheritance model allows any class with pure virtual functions to behave like an interface. This makes C++ more flexible, though less explicit than Java.

---

12. **A social media platform is implementing user profiles. Some details, such as username and profile picture, are visible to everyone, while other details, like contact number and email, are hidden and can only be accessed via a request system.**
    Question: What is the difference between Encapsulation and Abstraction in OOP? How would you use these concepts to design the privacy settings in this social media application?

Ans.

Key Difference: Encapsulation vs Abstraction

Feature	Encapsulation (Data Hiding)	Abstraction
Definition	Hiding internal data and allowing access only through methods	Hiding implementation details and showing only essential features
Purpose	Protect data from unauthorized access	Simplify design and interaction

Feature	Encapsulation (Data Hiding)	Abstraction
Implementation	Using private fields + getters/setters	Using abstract classes or interfaces
Focus	How data is protected	What the system does

Example: Social Media Privacy

In a social media app, user profile data can be:

- **Public (e.g., username, profile picture) → visible to all**
- **Private (e.g., contact number, email) → accessible only via request**

Using Encapsulation:

Encapsulation is used to hide sensitive data like contact number or email, allowing controlled access.

```
class UserProfile {
    private String username;
    private String profilePicture;
    private String contactNumber; // private
    private String email;        // private

    public UserProfile(String username, String profilePicture, String contactNumber, String email) {
        this.username = username;
        this.profilePicture = profilePicture;
        this.contactNumber = contactNumber;
        this.email = email;
    }

    // Public getters for visible info
```

```
    public String getUsername() {
        return username;
    }

    public String getProfilePicture() {
        return profilePicture;
    }

    // Controlled access via request
    public String requestContactInfo(boolean isAuthorized) {
        if (isAuthorized) {
            return contactNumber;
        } else {
            return "Access denied";
        }
    }

    public String requestEmail(boolean isAuthorized) {
        if (isAuthorized) {
            return email;
        } else {
            return "Access denied";
        }
    }
}
```

Using Abstraction:

Abstraction is used to define what operations can be done, without showing how they are done.
You might define an abstract interface for access control or profile visibility.

```
interface ProfilePrivacy {
   String getPublicInfo();
   String getPrivateInfo(boolean isAuthorized);
}

class BasicUserProfile implements ProfilePrivacy {
   private UserProfile profile;

   public BasicUserProfile(UserProfile profile) {
      this.profile = profile;
   }

   public String getPublicInfo() {
      return "Username: " + profile.getUsername() +
           "\nProfile Picture: " + profile.getProfilePicture();
   }

   public String getPrivateInfo(boolean isAuthorized) {
      return "Email: " + profile.requestEmail(isAuthorized) +
           "\nContact: " + profile.requestContactInfo(isAuthorized);
   }
}
```

- **Code :**

```
// Class to demonstrate Encapsulation
    class UserProfile {
        private String username;
        private String profilePicture;
        private String contactNumber;
        private String email;

        public UserProfile(String username, String profilePicture, String contactNumber, String
    email) {
            this.username = username;
            this.profilePicture = profilePicture;
            this.contactNumber = contactNumber;
            this.email = email;
        }

        // Public getters for non-sensitive info
        public String getUsername() {
            return username;
        }

        public String getProfilePicture() {
            return profilePicture;
        }

        // Controlled access to private info
        public String getContactNumber(boolean isAuthorized) {
            if (isAuthorized) {
                return contactNumber;
```

```
        } else {
            return "Access denied";
        }
    }

    public String getEmail(boolean isAuthorized) {
        if (isAuthorized) {
            return email;
        } else {
            return "Access denied";
        }
    }
}

// Interface to demonstrate Abstraction
interface ProfilePrivacy {
    String getPublicInfo();
    String getPrivateInfo(boolean isAuthorized);
}

// Class implementing the abstraction
class BasicUserProfile implements ProfilePrivacy {
    private UserProfile profile;

    public BasicUserProfile(UserProfile profile) {
        this.profile = profile;
    }
```

```
    public String getPublicInfo() {
        return "Username: " + profile.getUsername()
                + "\nProfile Picture: " + profile.getProfilePicture();
    }

    public String getPrivateInfo(boolean isAuthorized) {
        return "Email: " + profile.getEmail(isAuthorized)
                + "\nContact: " + profile.getContactNumber(isAuthorized);
    }
}

// Main class to run the program
public class Main {
    public static void main(String[] args) {
        UserProfile user = new UserProfile("john_doe", "profile.jpg", "123-456-7890",
"john@example.com");
        BasicUserProfile profileView = new BasicUserProfile(user);

        System.out.println("=== Public Info ===");
        System.out.println(profileView.getPublicInfo());

        System.out.println("\n=== Private Info (Unauthorized) ===");
        System.out.println(profileView.getPrivateInfo(false));

        System.out.println("\n=== Private Info (Authorized) ===");
        System.out.println(profileView.getPrivateInfo(true));
    }
}
```

Summary:

- **Encapsulation is used to protect sensitive profile data (email, phone) via private fields and access methods.**
- **Abstraction is used to provide a clean, simple interface (ProfilePrivacy) to the rest of the app, hiding how access is managed.**

---

13. A student information system needs to store student records in a file. It should allow reading student details when needed and updating them efficiently. Question: Explain how to read and write files in C++ using fstream. Provide an example of how the student information system could implement this feature.

Ans.

Explanation:

In C++, you can use the <fstream> header for file operations:

- **ofstream → for writing to files**
- **ifstream → for reading from files**
- **fstream → for both reading and writing**

- **C++ Example: Student Record File Handling**

  This program:

1. **Writes student data to a file.**
2. **Reads the data from the file.**
3. **Updates student records (by rewriting the file).**

- **Code:**

```
#include <iostream>
```

```
#include <sstream>
#include <vector>
using namespace std;

class Student {
public:
    string name;
    int roll;
    float marks;

    void inputData() {
        cout << "Enter Name: ";
        getline(cin, name);
        cout << "Enter Roll Number: ";
        cin >> roll;
        cout << "Enter Marks: ";
        cin >> marks;
        cin.ignore();
    }

    void displayData() const {
        cout << "Name: " << name << ", Roll: " << roll << ", Marks: " << marks << endl;
    }

    string toCSV() const {
        return name + "," + to_string(roll) + "," + to_string(marks);
    }
```

```
    static Student fromCSV(const string& line) {
        Student s;
        stringstream ss(line);
        getline(ss, s.name, ',');
        string rollStr, marksStr;
        getline(ss, rollStr, ',');
        getline(ss, marksStr, ',');
        s.roll = stoi(rollStr);
        s.marks = stof(marksStr);
        return s;
    }
};

// Simulate a file using a vector of strings
vector<string> fakeFileStorage;

void writeStudentRecord() {
    Student s;
    s.inputData();
    fakeFileStorage.push_back(s.toCSV());
    cout << "Student record added.\n";
}

void readStudentRecords() {
    cout << "\n--- All Student Records ---\n";
    if (fakeFileStorage.empty()) {
        cout << "No records found.\n";
        return;
```

```
    }

    for (const string& line : fakeFileStorage) {
        Student s = Student::fromCSV(line);
        s.displayData();
    }
}

int main() {
    int choice;
    do {
        cout << "\n1. Add Student\n2. View All Students\n0. Exit\nEnter your choice: ";
        cin >> choice;
        cin.ignore(); // consume newline after number input

        switch (choice) {
            case 1:
                writeStudentRecord();
                break;
            case 2:
                readStudentRecords();
                break;
            case 0:
                cout << "Exiting...\n";
                break;
            default:
                cout << "Invalid choice.\n";
        }
```

```
    } while (choice != 0);


    return 0;
}
```

How it works:

- **Writing: Appends new student records in the format name,roll,marks to the file.**
- **Reading: Reads and displays all student records line by line.**
- **Updating: For real-world use, update logic would involve reading into memory, modifying, and rewriting the file.**

---

14. Identify and Fix the Error in the Given Code public class ExceptionExample { public static void main(String[] args) { try { int num = 5 / 0; } catch () { System.out.println("Cannot divide by zero"); } } }

Ans.

The error in your code is in the catch block. In Java, a catch block must specify the type of exception it is catching.

- **Problem:**

catch () { **// Missing exception type**

- **Fix:**

Add the specific exception type you want to handle—ArithmeticException **for divide-by-zero:**

- **Correct Code:**

```
public class ExceptionExample {
    public static void main(String[] args) {
        try {
            int num = 5 / 0;
        }
        catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero");
        }
    }
}
```

---

15. **A payment gateway processes multiple types of transactions. While executing a transaction, different exceptions may occur, such as** InsufficientBalanceException, InvalidCardException, **or** TransactionTimeoutException**. The system must handle all these exceptions to ensure smooth transactions.**
    Question: How can multiple exceptions be handled in Java to make sure the payment gateway runs without failure? Provide a suitable example.

Ans.

In Java, multiple exceptions can be handled using either:

1. **Multiple catch blocks – one for each exception type.**
2. **Multi-catch block – handle several exceptions in one catch.**

- **Example: Handling Multiple Exceptions in a Payment Gateway**

```
public class PaymentExample {

    // Custom Exceptions as inner classes
    static class InsufficientBalanceException extends Exception {
```

```
    public InsufficientBalanceException(String message) {
        super(message);
    }
}

static class InvalidCardException extends Exception {
    public InvalidCardException(String message) {
        super(message);
    }
}

static class TransactionTimeoutException extends Exception {
    public TransactionTimeoutException(String message) {
        super(message);
    }
}

// Transaction Processing Class
static class PaymentGateway {
    public void processTransaction(String cardNumber, double balance)
            throws InsufficientBalanceException, InvalidCardException,
TransactionTimeoutException {

        if (cardNumber == null || cardNumber.length() != 16) {
            throw new InvalidCardException("Card number is invalid.");
        }

        if (balance < 1000) {
```

```
            throw new InsufficientBalanceException("Insufficient balance.");
        }

        if (Math.random() > 0.7) { // Simulate a timeout randomly
            throw new TransactionTimeoutException("Transaction timed out.");
        }

        System.out.println("Transaction successful!");
    }
}

// Main method
public static void main(String[] args) {
    PaymentGateway gateway = new PaymentGateway();

    // Test values
    String cardNumber = "1234567890123456";
    double balance = 800; // Change to 1500 to simulate success

    try {
        gateway.processTransaction(cardNumber, balance);
    } catch (InvalidCardException e) {
        System.out.println("Error: " + e.getMessage());
    } catch (InsufficientBalanceException e) {
        System.out.println("Error: " + e.getMessage());
    } catch (TransactionTimeoutException e) {
        System.out.println("Error: " + e.getMessage());
    } finally {
```

```
        System.out.println("Transaction attempt completed.");
    }
  }
}
```

Key Concepts:

- **Custom exceptions allow you to represent real-world issues in code.**
- **Each catch handles a specific type of error.**
- **finally ensures cleanup or final messaging, regardless of success or failure.**

---

16. A hospital management system stores patient records. Each patient has a unique ID, name, and medical history. The system should allow doctors to create and access patient records efficiently.
Question: What is the difference between a Class and an Object in OOP? Explain using this hospital management system as an example.

Ans.

Conceptual Explanation:

Class	Object
A class is a blueprint for creating objects.	An object is an instance of a class.
It defines attributes (like ID, name, history) and behaviors (like create or access records).	It holds actual data for a specific patient.
No memory is allocated for a class by itself.	Memory is allocated when the object is created.
In the hospital system, Patient is a class.	patient1, patient2 are objects of that class.

- **Java Code:**

```
class Patient {
    int id;
    String name;
    String medicalHistory;

    // Constructor
    public Patient(int id, String name, String medicalHistory) {
        this.id = id;
        this.name = name;
        this.medicalHistory = medicalHistory;
    }

    // Method to display patient details
    public void displayRecord() {
        System.out.println("Patient ID: " + id);
        System.out.println("Name: " + name);
        System.out.println("Medical History: " + medicalHistory);
    }
}

// Main class with main() method
public class Main {
    public static void main(String[] args) {
        // Creating patient objects
        Patient patient1 = new Patient(1, "Rahul Sharma", "Allergy to penicillin");
        Patient patient2 = new Patient(2, "Meera Joshi", "Asthma, High BP");

        // Displaying patient records
```

```
        System.out.println("=== Patient 1 Record ===");
        patient1.displayRecord();

        System.out.println("\n=== Patient 2 Record ===");
        patient2.displayRecord();
    }
}
```

Summary:

- **Patient class defines what a patient is (structure).**
- **patient1 and patient2 are actual records stored and used in the system.**

---

17. A company is developing a role-based access control system where different users have different levels of access. Admin can modify and delete data, Manager can view and edit, while Employees can only view certain details. Question: Create a comparison table of access modifiers (private, protected, public, and default) and explain how they can be used in this system to restrict access appropriately.

Ans.

- **Access Modifiers Comparison Table:**

Context	Default	Private	Protected	Public
Same Class	**Yes**	**Yes**	**Yes**	**Yes**
Same Package Subclass	**Yes**	**No**	**Yes**	**Yes**
Same Package Non-Subclass	**Yes**	**No**	**Yes**	**Yes**
Different Package Subclass	**No**	**No**	**Yes**	**Yes**
Different Package Non-Subclass	**No**	**No**	**No**	**Yes**

How Access Modifiers Can Be Used in This System

1. Admin (Full Access)

- **Can access and modify private/protected data via public admin methods.**
- **Admin methods are declared as public.**

2. Manager (View + Edit)

- **Inherits from a base user class with protected edit methods.**
- **Restricted from using private data or delete functions.**

3. Employee (View Only)

- **Only has access to public view methods.**
- **Cannot access protected or private operations.**

- **Java code:**

```
class UserData {
    private String confidentialData = "Secret";

    protected void editData() {
        System.out.println("Editing data...");
    }

    public void viewData() {
        System.out.println("Viewing data...");
    }

    private void deleteData() {
        System.out.println("Deleting data... (Admin only)");
    }
}
```

```
class Manager extends UserData {
    public void accessFeatures() {
        viewData();       // Allowed
        editData();       // Allowed (protected)
        // deleteData();  // Not allowed (private)
    }
}

class Employee extends UserData {
    public void accessFeatures() {
        viewData();       // Allowed
        // editData();    // Not allowed (protected)
    }
}

public class Main {
    public static void main(String[] args) {
        Manager manager = new Manager();
        manager.accessFeatures();

        Employee employee = new Employee();
        employee.accessFeatures();
    }
}
```

---

18. **A robotics startup is using C++ and Java to develop a robotic arm control system. While coding the interfaces for different sensors, they observe that Java requires interface and implements keywords, whereas C++ does not.**
Question: Why does C++ not require the interface or implements keyword like Java? Explain with a code example.

Ans.

Sure. Here's a clear explanation without emojis, along with the code examples in both Java and C++.

Question: Why does C++ not require the interface or implements keyword like Java?

Explanation:

- **Java is a purely object-oriented language. It does not support multiple inheritance of classes to avoid ambiguity. Therefore, Java introduced interfaces as a way to allow a class to inherit behavior from multiple sources without inheriting implementation.**
    - **Interfaces in Java are defined using the interface keyword.**
    - **Classes implement them using the implements keyword.**
- **C++, on the other hand, supports multiple inheritance. You can use abstract classes with pure virtual functions to define an interface-like structure.**
    - **There is no need for special keywords like interface or implements.**
    - **A class with one or more pure virtual functions acts like an interface.**

- **Java Example (Using interface and implements):**

```
// Sensor.java
interface Sensor {
    void readData();
}

// TemperatureSensor.java
class TemperatureSensor implements Sensor {
    public void readData() {
```

```
        System.out.println("Reading temperature data...");
    }
}

public class Main {
    public static void main(String[] args) {
        Sensor sensor = new TemperatureSensor();
        sensor.readData();
    }
}
```

- **C++ Example (Using Abstract Class with Pure Virtual Function):**

```
#include <iostream>
using namespace std;

// Abstract base class (acts like an interface)
class Sensor {
public:
    virtual void readData() = 0; // Pure virtual function
};

// Concrete class implementing the interface
class TemperatureSensor : public Sensor {
public:
    void readData() override {
        cout << "Reading temperature data..." << endl;
    }
};
```

```
int main() {
    Sensor* sensor = new TemperatureSensor();
    sensor->readData();
    delete sensor;
    return 0;
}
```

Summary Table:

Feature	Java	C++
Interface Keyword	**Yes (interface)**	**No**
Implements Keyword	**Yes (implements)**	**No**
Multiple Inheritance	**Not supported (for classes)**	**Supported**
Interface Representation	**Interface**	**Abstract class with pure virtual methods**

---

19. A school management system has a Student class that inherits from the Person class. The Person class has a method getDetails(), which returns basic information, but the Student class needs to provide additional student specific details.
Question: How do the super and this keywords work in Java? Explain their use in this school management system.

Ans.

In Java, the keywords super and this are used to differentiate between:

- **super: Refers to the parent class (superclass).**
- **this: Refers to the current class (subclass).**

Use in a School Management System

In this system:

- **The Person class has a method** getDetails() **that returns general info.**
- **The Student class inherits from Person and overrides** getDetails() **to add more information.**
- **The super keyword is used in Student to call the** getDetails() **method from Person.**
- **The this keyword can be used to refer to instance variables when needed.**

- **Java Code :**

```
// Superclass
class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name; // 'this' refers to current object's variables
        this.age = age;
    }

    String getDetails() {
        return "Name: " + name + ", Age: " + age;
    }
}

// Subclass
```

```
class Student extends Person {
    String studentId;

    Student(String name, int age, String studentId) {
        super(name, age); // 'super' calls parent constructor
        this.studentId = studentId; // 'this' refers to the current class's variable
    }

    @Override
    String getDetails() {
        // Use 'super' to call the parent method
        return super.getDetails() + ", Student ID: " + studentId;
    }
}

// Main class to test
public class Main {
    public static void main(String[] args) {
        Student student = new Student("Alice", 18, "STU123");
        System.out.println(student.getDetails());
    }
}
```

Summary Table

Keyword	Refers To	Use Case
this	**Current class instance**	**To refer to instance variables, or call constructors in the same class**

Keyword	Refers To	Use Case
super	Parent class (superclass)	To call parent class methods or constructors

---

20. A vehicle rental management system allows customers to rent vehicles. Customers can specify rental duration, vehicle type, or even add extra services like insurance.
Question: How would you implement method overloading for a rent_vehicle() method to support different customer needs? Provide an example.

Ans.

Concept:

- **Method Overloading: Multiple methods with the same name but different parameters.**
- **This allows customers to rent a vehicle by specifying:**
  - **Only vehicle type**
  - **Vehicle type + rental duration**
  - **Vehicle type + rental duration + extra service**

- **Java Code:**

```
class RentalService {
   // Rent by vehicle type
   void rentVehicle(String vehicleType) {
      System.out.println("Renting a " + vehicleType);
   }

   // Rent by vehicle type and duration
   void rentVehicle(String vehicleType, int duration) {
```

```
        System.out.println("Renting a " + vehicleType + " for " + duration + " days.");
    }

    // Rent by vehicle type, duration, and extra service
    void rentVehicle(String vehicleType, int duration, boolean insurance) {
        System.out.println("Renting a " + vehicleType + " for " + duration + " days.");
        if (insurance) {
            System.out.println("Insurance service added.");
        } else {
            System.out.println("No insurance added.");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        RentalService rental = new RentalService();

        // Different overloaded method calls
        rental.rentVehicle("Car");
        rental.rentVehicle("Bike", 3);
        rental.rentVehicle("Truck", 5, true);
    }
}
```

---

21. **A banking application processes transactions. If an error occurs, such as insufficient funds or invalid account number, the system should handle it gracefully instead of crashing.**
Question: Define Exception Handling and Error Handling and explain how they help in building reliable banking applications.

Ans.

- **Difference Between Exception Handling and Error Handling**

Aspect	Exception Handling	Error Handling
Definition	A mechanism to handle runtime anomalies (e.g., invalid input, database connection issues).	A broader term that includes exception handling and system-level faults.
Scope	Deals with issues in the program's logic or flow.	Can also deal with critical system-level issues (e.g., memory overflow).
Recoverable?	Most exceptions are recoverable.	Many errors (like hardware failure) are not recoverable.
Handled using	try, catch, finally, throw, and throws in Java.	May involve low-level diagnostics, logs, alerts, or shutting down safely.
Usage in Banking	To manage failed transactions, login errors, insufficient funds, etc.	To detect major system failures, out-of-memory errors, etc.

How They Help in a Banking Application

- **Exception Handling:**
    - **Ensures smooth user experience by catching issues like:**
        - **Invalid account number**
        - **Insufficient balance**
        - **Transaction timeout**
    - **Prevents crashes and allows showing user-friendly error messages.**
- **Error Handling:**

- **Ensures the system handles critical errors such as:**
  - **File system failure**
  - **Server outage**
  - **Memory leaks**
- **May involve system rebooting, logging, and recovery procedures.**

- **Java code:**

```
class BankTransaction {
    void processTransaction(String accountNumber, double balance, double amount) {
        try {
            if (accountNumber == null || accountNumber.isEmpty()) {
                throw new IllegalArgumentException("Invalid account number.");
            }
            if (amount > balance) {
                throw new Exception("Insufficient funds.");
            }

            System.out.println("Transaction successful. Withdrawn: ₹" + amount);
        } catch (IllegalArgumentException e) {
            System.out.println("Error: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("Transaction failed: " + e.getMessage());
        } finally {
            System.out.println("Transaction attempt completed.");
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        BankTransaction bt = new BankTransaction();
        bt.processTransaction("", 5000, 2000);  // Invalid account
        bt.processTransaction("ACC123", 1000, 2000);  // Insufficient funds
        bt.processTransaction("ACC123", 5000, 3000);  // Success
    }
}
```

Conclusion:

- **Exception handling provides fault tolerance and better user interaction.**
- **Error handling ensures system reliability and resilience.**
- **Both are essential in a secure and reliable banking application.**

---

22. Identify and Fix the Error in the Given Code class Student { private int age = 20; } public class Main { public static void main(String[] args) { Student s = new Student(); System.out.println(s.age); } }

Ans.

The error in this code is due to accessing a private member (age) of the Student class directly from outside the class, which is not allowed in Java.

Fix: Use a getter method to access the private field.

- **corrected code:**

```
class Student {
    private int age = 20;

    // Getter method to access private variable
```

```
    public int getAge() {
        return age;
    }
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student();
        System.out.println("Student age: " + s.getAge());
    }
}
```

---

23. A flight booking system allows customers to book tickets. If an invalid seat number is entered, the system should immediately generate an exception. However, some exceptions, like invalid passport numbers, should be declared in the method signature for the calling function to handle.
Question: What is the difference between throw and throws in Java? Explain with reference to this flight booking system.

Ans.

- **Key Differences Between throw and throws:**

Feature	**throw**	**throws**
Purpose	**Used to actually throw an exception**	**Declares that a method might throw an exception**
Placement	**Inside the method body**	**In the method declaration**
Used With	**An instance of an exception (e.g., new Exception())**	**Exception class names**

Feature	throw	throws
Number of Exceptions	One at a time	Can declare multiple exceptions, comma-separated

---

- **Code:**

```
class BookingSystem {

    // Method that checks seat number and throws an exception
    void bookSeat(int seatNumber) {
        if (seatNumber <= 0 || seatNumber > 100) {
            throw new IllegalArgumentException("Invalid seat number!");
        }
        System.out.println("Seat " + seatNumber + " booked successfully.");
    }

    // Method that declares a checked exception using throws
    void verifyPassport(String passportNumber) throws Exception {
        if (passportNumber.length() != 8) {
            throw new Exception("Invalid passport number!");
        }
        System.out.println("Passport verified.");
    }
}

public class Main {
    public static void main(String[] args) {
        BookingSystem booking = new BookingSystem();
```

```
        // Handle unchecked exception from bookSeat()
        try {
            booking.bookSeat(105); // Invalid seat
        } catch (IllegalArgumentException e) {
            System.out.println("Booking Error: " + e.getMessage());
        }

        // Handle checked exception from verifyPassport()
        try {
            booking.verifyPassport("A12345"); // Invalid passport
        } catch (Exception e) {
            System.out.println("Passport Error: " + e.getMessage());
        }
    }
}
```

Summary:

- **Use throw to actually throw an exception (e.g., invalid seat).**
- **Use throws in method signature to declare an exception that the caller must handle (e.g., invalid passport).**
