LONDON METROPOLITAN UNIVERSITY

ITAHARI INTERNATIONAL COLLEGE

Module Code & Module Title

CS4001NT Programming

Assessment Weightage & Type

10% Individual Coursework

Year and Semester

2024 Autumn

Student Name: Sonam Acharya

London Met ID: 24046086

College ID: np05cp4a240410@iic.edu.np

Assignment Due Date: May 2, 2025

Assignment Submission Date: May 2, 2025

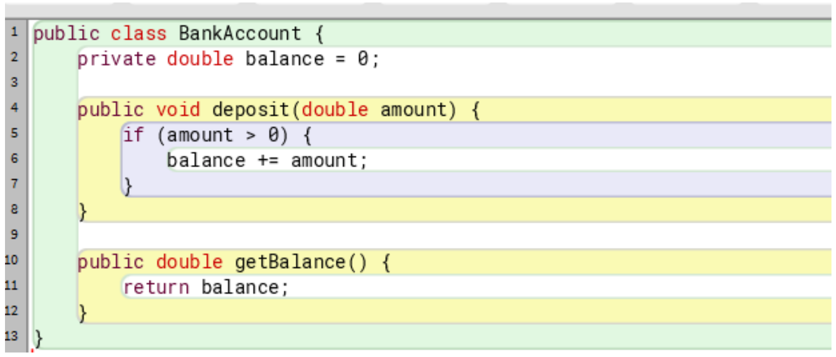
Table of Contents

Table Of Figures

1.a. What are the four main principles of Object-Oriented Programming (OOP)? Provide a brief explanation of each.

Ans. The four main principles of object-oriented programming are as follows:

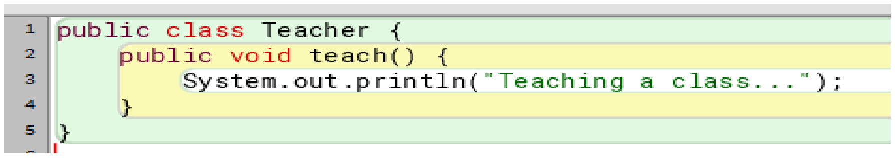
1.1. Encapsulation: The process of hiding the implementation and sharing only the necessary through the public interface is called Encapsulation. Encapsulation is also known as the bundle of attribute or properties and methods into a single unit called a class. The class contains everything needed for a part of creation. For example, if you own the car, you know how to run the car, but you don't know how it works, that's encapsulation where the implementation is hidden.



```
public class BankAccount {
    private double balance = 0;

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

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

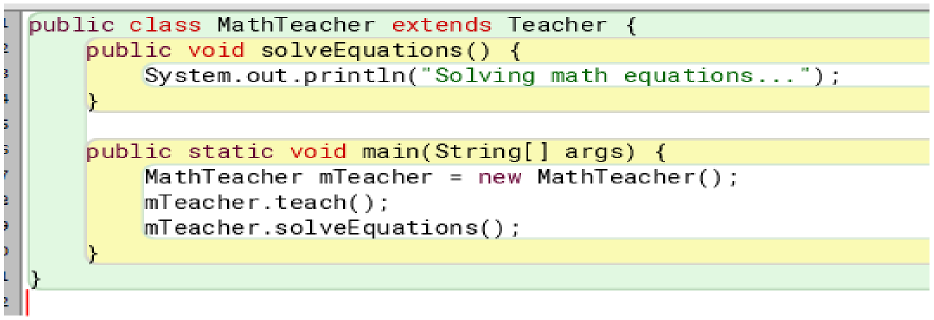
Figure 1: Keeping class data private and using getter/setter methods to control access

1.1. Inheritance: The process of inheriting behaviors and properties from another class (sub class or base class or supper class) is called Inheritance. It has benefits like Code Reusability, Hierarchy Establishment, Maintainability where avoids writing the same code multiple times, Allows the creation of aa hierarchical structure of classes, flexing the relationship between different classes and objects. For example, you have created a bus and car to set the similar item like wheels, handles etc. are inherited at the time the process of inheriting the behaviors and properties from another class is known as inheritance.



```
public class Teacher {
    public void teach() {
        System.out.println("Teaching a class...");
    }
}
```

Figure 2:Parent class teacher



```
public class MathTeacher extends Teacher {
    public void solveEquations() {
        System.out.println("Solving math equations...");
    }

    public static void main(String[] args) {
        MathTeacher mTeacher = new MathTeacher();
        mTeacher.teach();
        mTeacher.solveEquations();
    }
}
```

Figure 3: MathTeacher class to reuse and extend the functionality of the Teacher class.

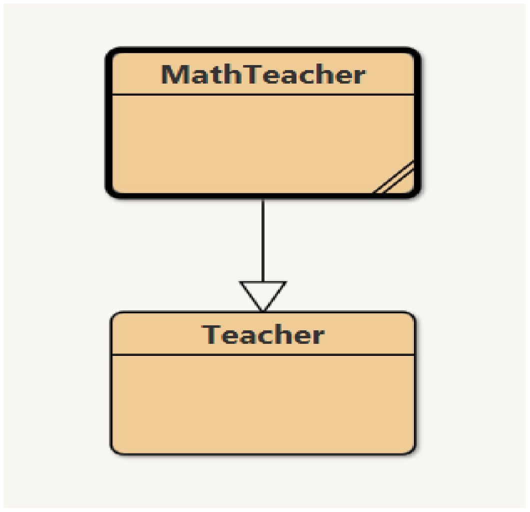
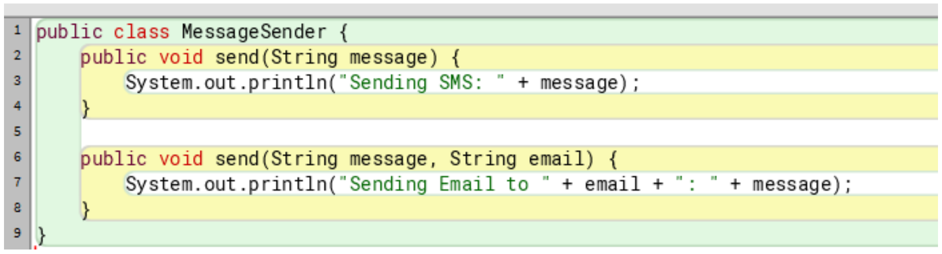


Figure 4:Class Diagram Of Teacher And Math Teacher

1.2. Polymorphism:

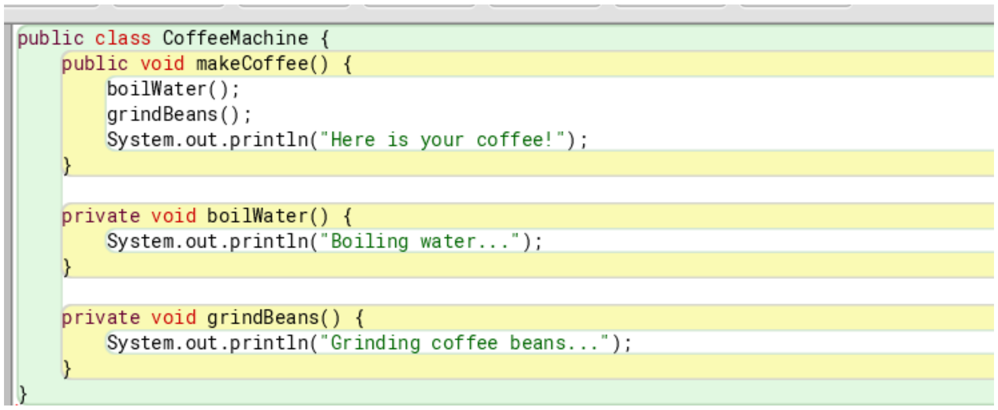
Polymorphism allows objects to behave differently based on specific class type. Polymorphism has features like multiple behaviors like the same method can behave differently depending on the object call this method, In Method Overriding, Method overloading, A child class can define or redefine a method of its parent class or sub class and we can define multiple methods in the same class with different parameters respectively.



```
public class MessageSender {
    public void send(String message) {
        System.out.println("Sending SMS: " + message);
    }

    public void send(String message, String email) {
        System.out.println("Sending Email to " + email + ": " + message);
    }
}
```

Figure 5:Different notification types to respond differently to the same method call

1.3. Abstraction: The process of simplifying the complex reality and hiding the complex implementation is known as Abstraction. It mainly focuses on “what” an object does rather than “how”. For example, we use keyboards, mouse of the laptop but how it runs inside we don’t know what is happening inside Is known as the abstraction of data.

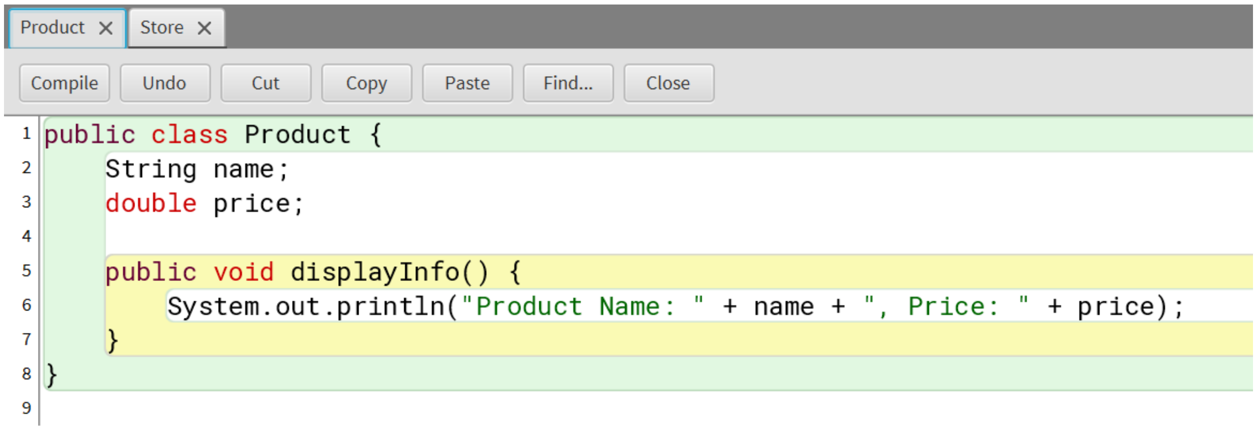


```
public class CoffeeMachine {
    public void makeCoffee() {
        boilWater();
        grindBeans();
        System.out.println("Here is your coffee!");
    }

    private void boilWater() {
        System.out.println("Boiling water...");
    }

    private void grindBeans() {
        System.out.println("Grinding coffee beans...");
    }
}
```

Figure 6:Hiding implementation details while allowing interaction through a general interface.

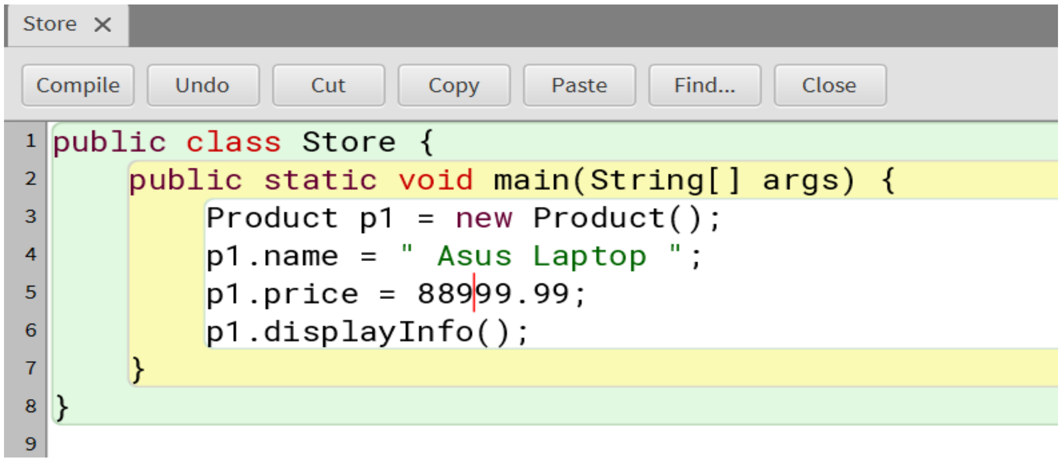
1.b. How does the concept of a class differ from that of an object in Java? Provide examples from the course to illustrate your answer.

Ans. In Java, the concept of class differs from that of an object form the foundation of object-oriented design. The class is also known as the blueprint of the class. It has both properties (fields or attributes) and behavior(methods) that the object created from it will have. For instance, a class called a Bike that includes attributes like Brand, Color, Price and speed and behavior(methods) such as startEngine (), applyBrakes(), applyHorn (). An object is an actual instance of class. It's also known as real world entity representation created by the class . For example, when a class creates something specific like a Red Bullet that moves at the speed of 70 km/h, that the specific red bullet is the object of the class. In a course module, where a Car is demonstrated as class, and it has attributes of model, brand and methods or behavior such as start () and accelerate(). The car is an object and the start () , accelerate () is a method.



```java
public class Product {
    String name;
    double price;

    public void displayInfo() {
        System.out.println("Product Name: " + name + ", Price: " + price);
    }
}
```

Figure 7: A class or Blueprint



```java
public class Store {
    public static void main(String[] args) {
        Product p1 = new Product();
        p1.name = " Asus Laptop ";
        p1.price = 88999.99;
        p1.displayInfo();
    }
}
```

Figure 8: An object of a class

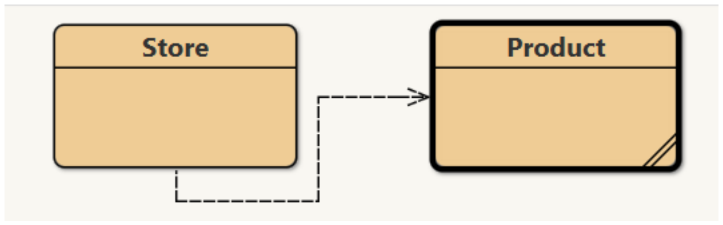
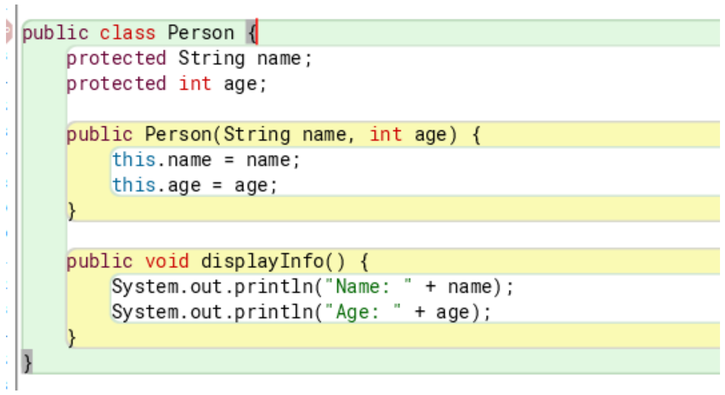


Figure 9:Class Diagram Of store and product

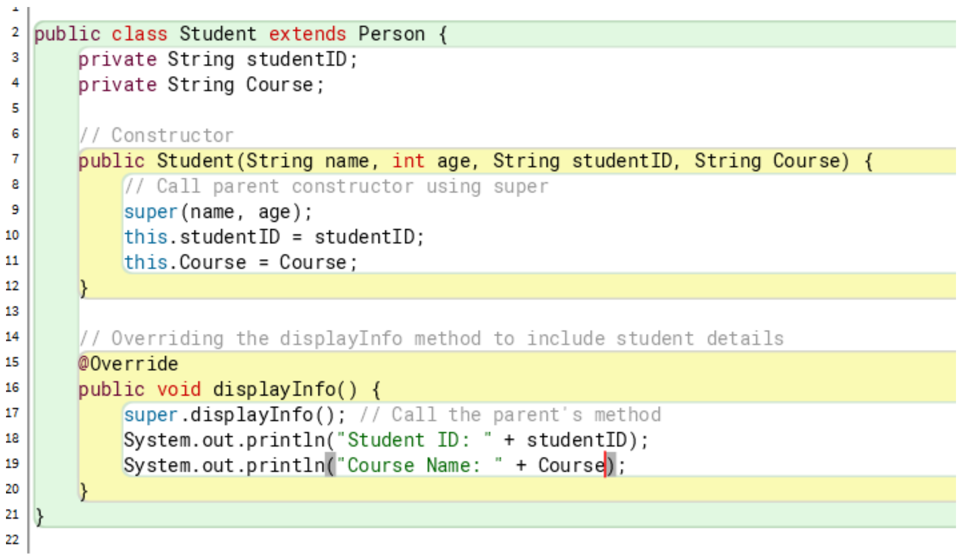
2.a. Describe how you implemented inheritance in one of the examples provided in the course. How does inheritance promote code reuse?

Ans. In the course, object-oriented programming, inheritance is implemented by using the real word models, such as “person” and “student” class. I implemented inheritance by creating the class Person and with common attributes like name, age and along with the displayInfo () method or behavior. Extends keyword in java help to inherit the properties. After creating Person class I have created Student class which extends Person. The student class inherits all the properties and behaviors of Person class. And added some attributes to Student class like studentID and courseName. Implementation allowed the student class to reuse existing code form the Person class. After watching the linked learning helped me to understand how to create relationship between classes like parent and child class make easier to reuse code and decrease code redundancy.



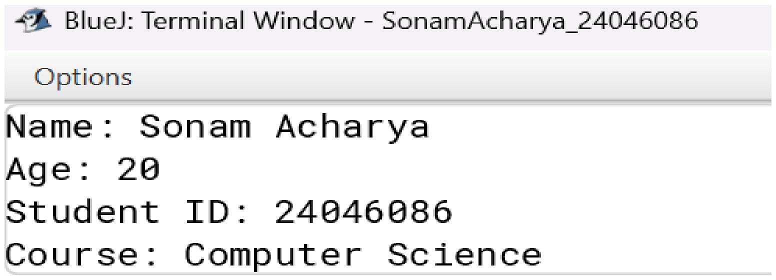
```
public class Person {
    protected String name;
    protected int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}
```

Figure 10: Parent class Person



```java
public class Student extends Person {
    private String studentID;
    private String Course;

    // Constructor
    public Student(String name, int age, String studentID, String Course) {
        // Call parent constructor using super
        super(name, age);
        this.studentID = studentID;
        this.Course = Course;
    }

    // Overriding the displayInfo method to include student details
    @Override
    public void displayInfo() {
        super.displayInfo(); // Call the parent's method
        System.out.println("Student ID: " + studentID);
        System.out.println("Course Name: " + Course);
    }
}
```

Figure 11: Base Class Student



BlueJ: Terminal Window - SonamAcharya_24046086

Options

```
Name: Sonam Acharya
Age: 20
Student ID: 24046086
Course: Computer Science
```

Figure 12:Demonstration of Inheritance in Java using Course example

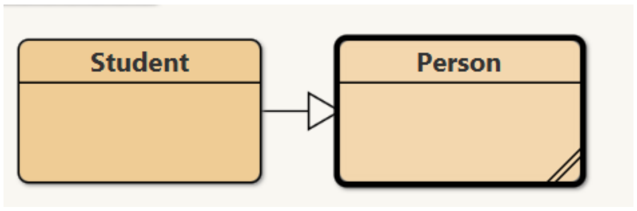
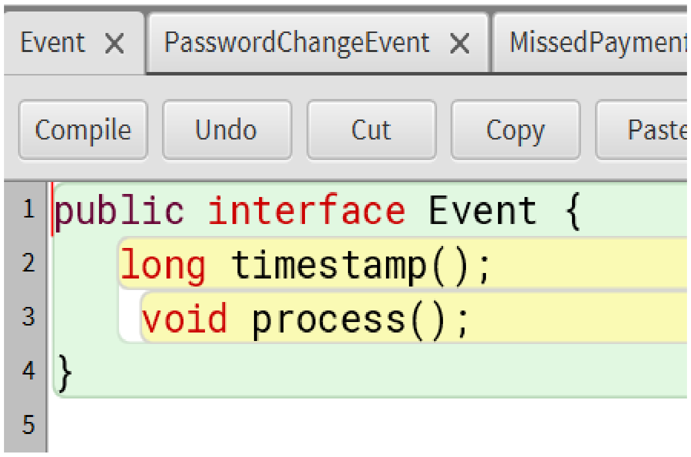


Figure 13:Class Diagram of Person And student

2.b. How did the course demonstrate the use of interfaces in Java? Provide an example of an interface you created during the course and explain its purpose.

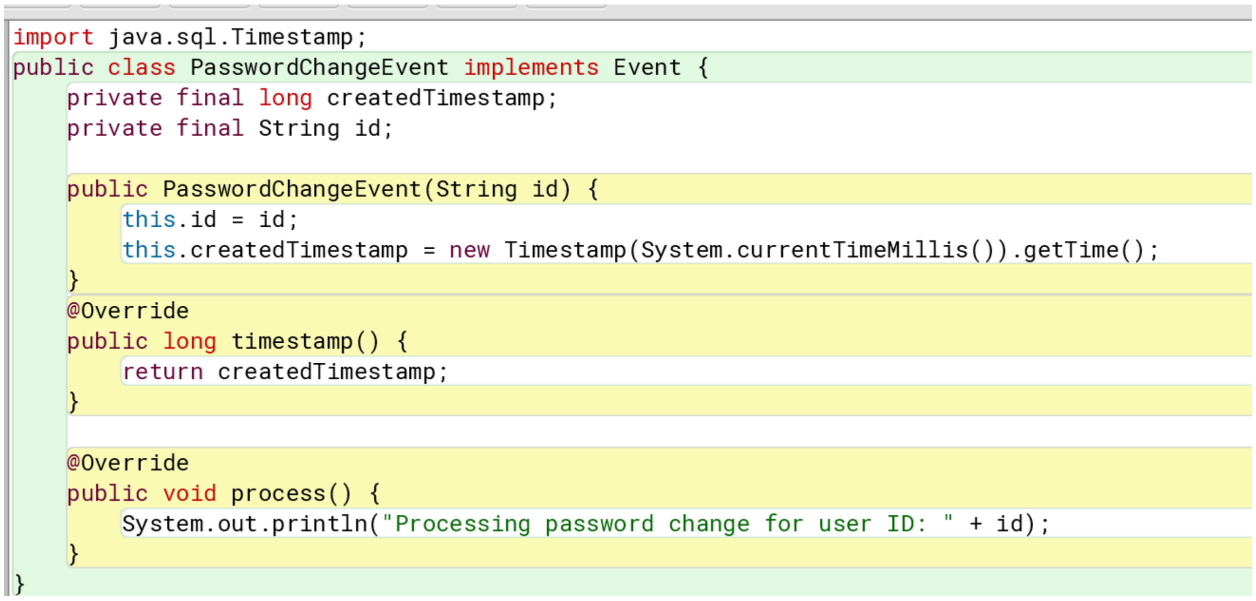
Ans: The course demonstrates the use of interfaces in java are a way to define a set of method that a class must implement, helping with the same structure for different parts of program.

One of the examples of an interface created during the course was “Event” was the interface and the “passwordChangeEvent” was the class and we use implements keywords for providing the functionality defined in the main interface class. missedPayement and accountTransfer class was also created. Event has the method named: timestamp () and process (). The main purpose of the interface was to ensure consistency across all types of events, keep the code modular and easy to extend and understand. Enabling the polymorphism to keep that all the events can be handled by using the reference of same interface.



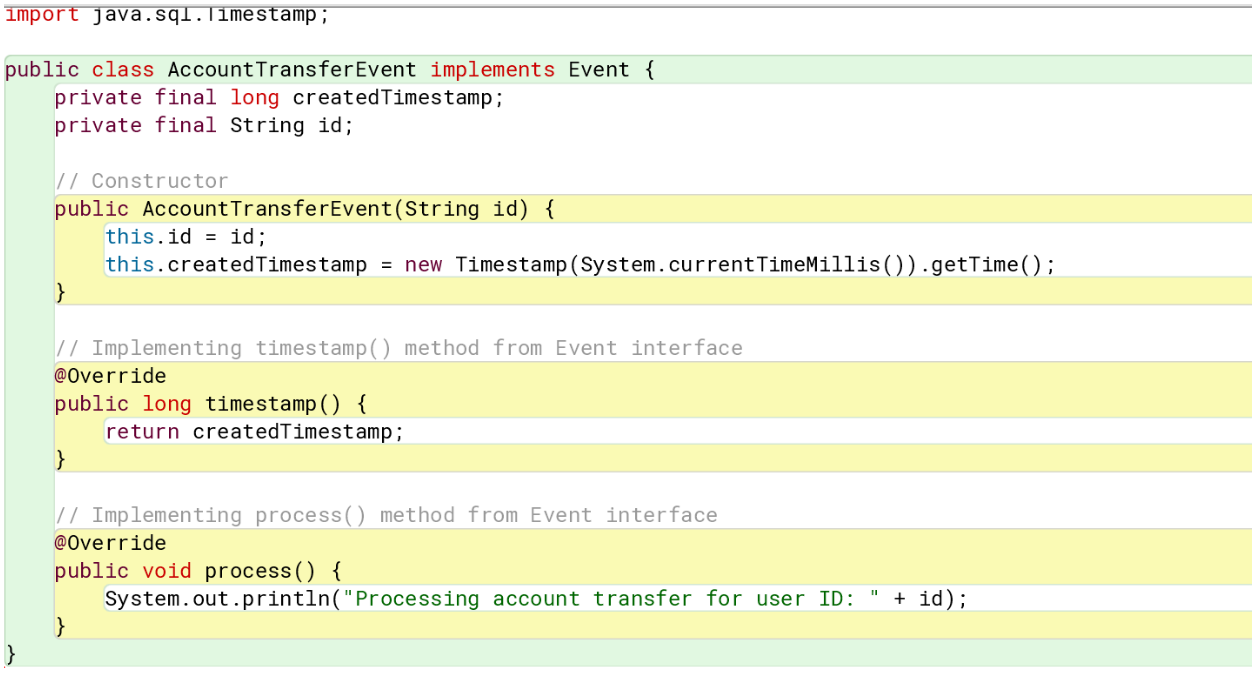
```java
public interface Event {
    long timestamp();
     void process();
}
```

Figure 14:Event Interface



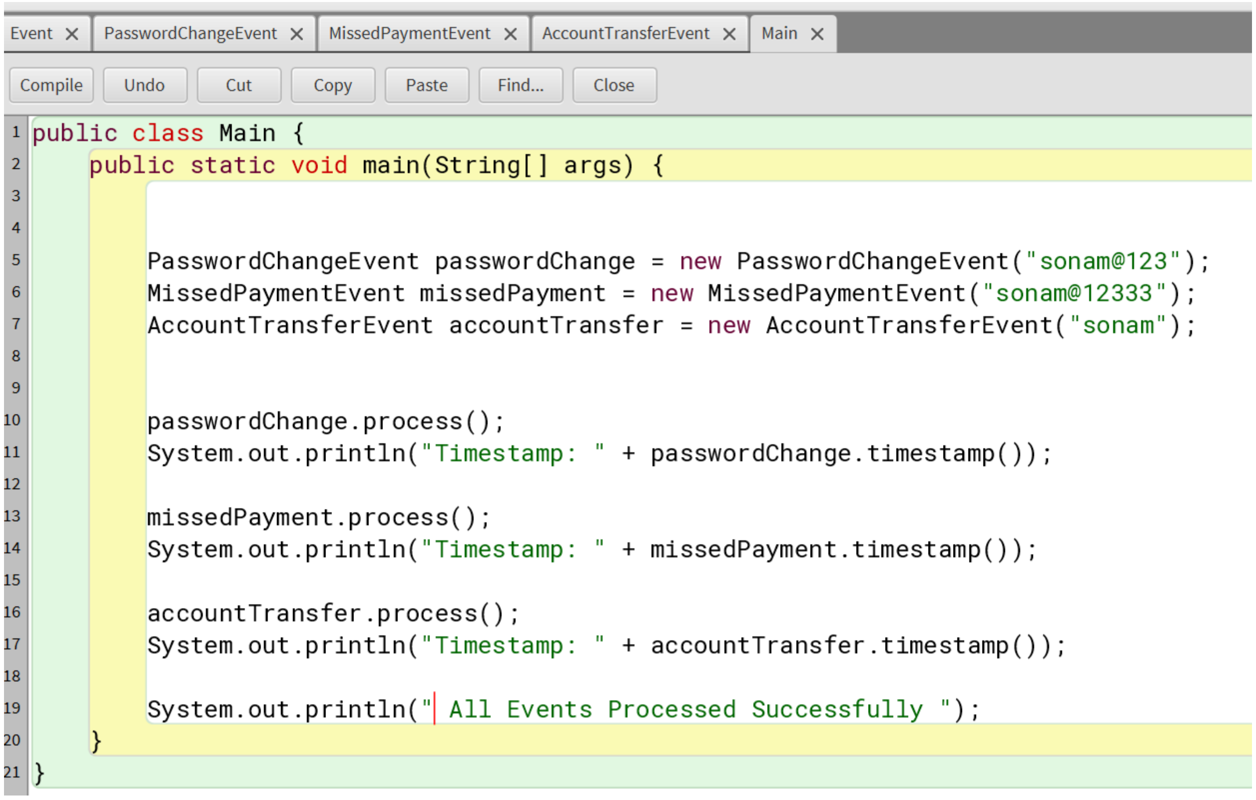
```java
import java.sql.Timestamp;
public class PasswordChangeEvent implements Event {
    private final long createdTimestamp;
    private final String id;

    public PasswordChangeEvent(String id) {
        this.id = id;
        this.createdTimestamp = new Timestamp(System.currentTimeMillis()).getTime();
    }
    @Override
    public long timestamp() {
        return createdTimestamp;
    }

    @Override
    public void process() {
        System.out.println("Processing password change for user ID: " + id);
    }
}
```

Figure 15: Password Change Event that implements Event



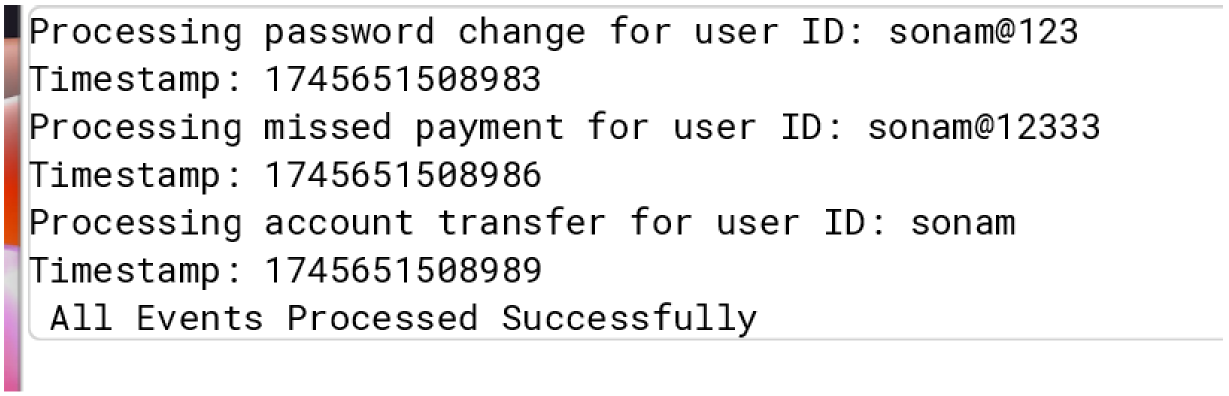
```java
import java.sql.Timestamp;

public class AccountTransferEvent implements Event {
    private final long createdTimestamp;
    private final String id;

    // Constructor
    public AccountTransferEvent(String id) {
        this.id = id;
        this.createdTimestamp = new Timestamp(System.currentTimeMillis()).getTime();
    }

    // Implementing timestamp() method from Event interface
    @Override
    public long timestamp() {
        return createdTimestamp;
    }

    // Implementing process() method from Event interface
    @Override
    public void process() {
        System.out.println("Processing account transfer for user ID: " + id);
    }
}
```

Figure 16:AccountTransferEvent that implements Event



```java
public class Main {
    public static void main(String[] args) {


        PasswordChangeEvent passwordChange = new PasswordChangeEvent("sonam@123");
        MissedPaymentEvent missedPayment = new MissedPaymentEvent("sonam@12333");
        AccountTransferEvent accountTransfer = new AccountTransferEvent("sonam");


        passwordChange.process();
        System.out.println("Timestamp: " + passwordChange.timestamp());

        missedPayment.process();
        System.out.println("Timestamp: " + missedPayment.timestamp());

        accountTransfer.process();
        System.out.println("Timestamp: " + accountTransfer.timestamp());

        System.out.println(" All Events Processed Successfully ");
    }
}
```

Figure 17:Main class demonstrates creating objects of different event types



```
Processing password change for user ID: sonam@123
Timestamp: 1745651508983
Processing missed payment for user ID: sonam@12333
Timestamp: 1745651508986
Processing account transfer for user ID: sonam
Timestamp: 1745651508989
 All Events Processed Successfully
```

Figure 18:Output Of Interface implements

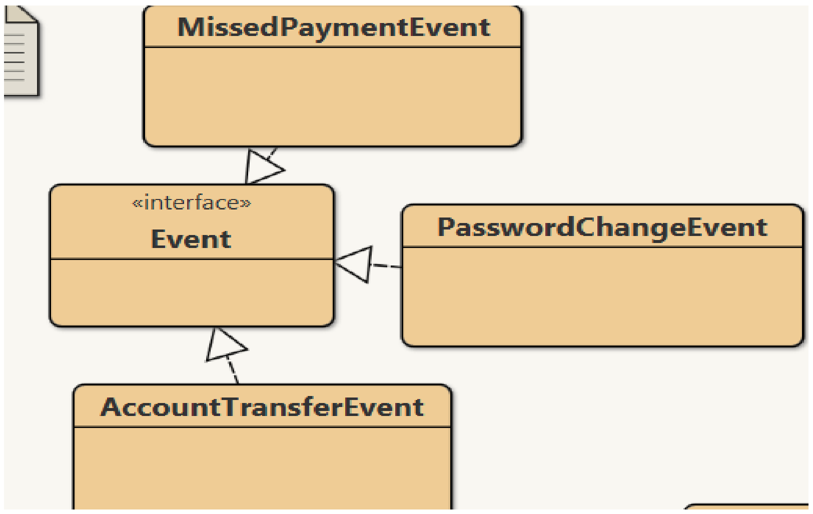


Figure 19: Class Diagram of interface

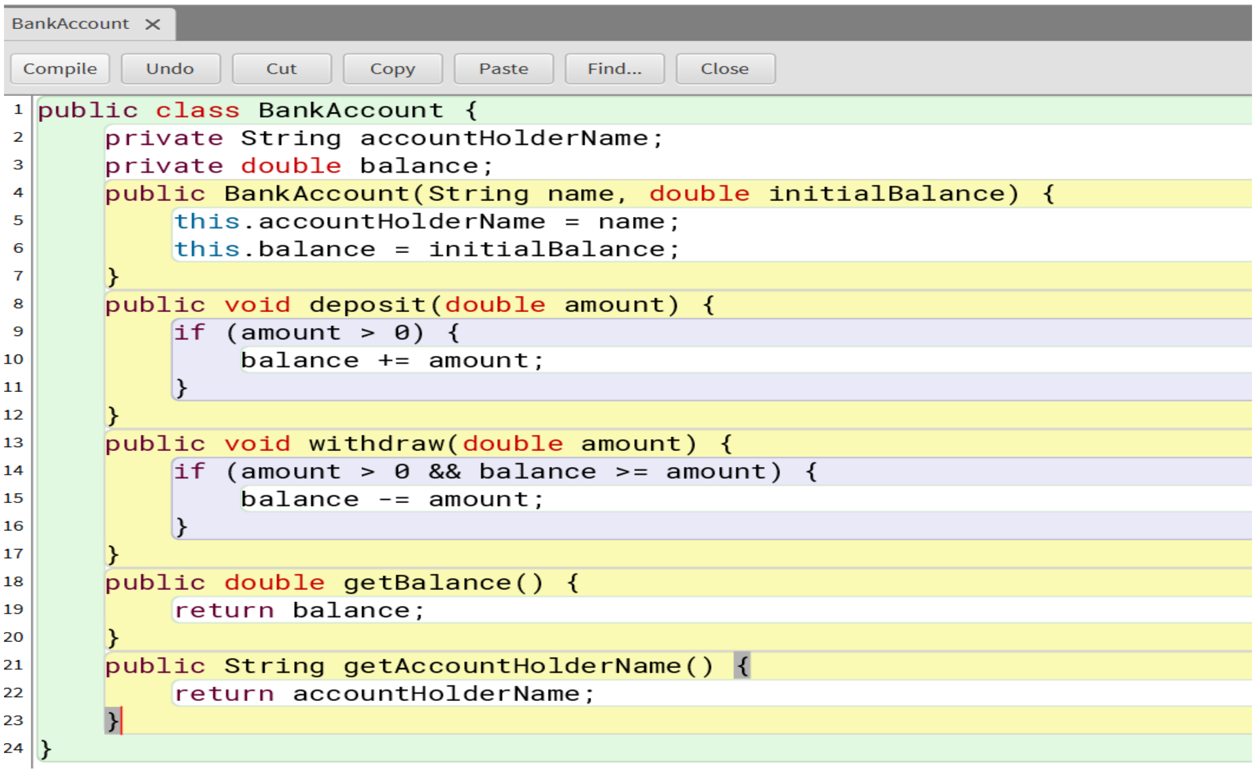
3.a. Explain how encapsulation is achieved in Java. Why is encapsulation important in software development?

Ans: In LinkedIn learning OOP course, Encapsulation is a fundamental principle of oop (object-oriented programming) that was clearly shown throughout the course. Encapsulation is the method to bundle the data and methods into a single unit. class and restricting access to the inner working of that class. Making variable private and only allowing access through getter setter methods.

For example, on one of the courses, we have created a bankAccount class where the balance variable is set as private. This means balance couldn't be access by or directly accessed or modifieds outside of the class . If we want any changes outside of the class then we have to give getter or setter methods .

Encapsulation I not just a technique to hide the code, it's an essential part of the code to develop secure, maintainable and neat and clean software. It ensures object to reduces bug, hiding the code complexity and making it easier to make change without affecting any parts of the system. In software development encapsulation keeps everything well organized and managed. Allowing each class to manage its own data and behavior which leads better security, reduced risk of error ,easier maintenance and improved reusability

. In summary, the course helps us to encapsulation promote clean and reliable code in the real world.



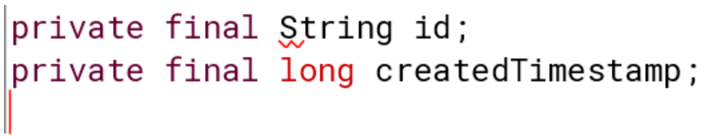
```
public class BankAccount {
    private String accountHolderName;
    private double balance;
    public BankAccount(String name, double initialBalance) {
        this.accountHolderName = name;
        this.balance = initialBalance;
    }
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }
    public void withdraw(double amount) {
        if (amount > 0 && balance >= amount) {
            balance -= amount;
        }
    }
    public double getBalance() {
        return balance;
    }
    public String getAccountHolderName() {
        return accountHolderName;
    }
}
```

Figure 20:Demo Of Encapsulation

3.b. In the course, you worked with access modifiers like `private`, `protected`, and `public’. Describe a scenario where each would be appropriately used.

Ans: During Linked learning course, we have used or worked with access modifiers like Private, Protected and public which help us to manage access to class members(variables), promoting safe, clean and understandable code. Reducing complexity. One of the exercises we have used modifiers especially when working with interfaces and their implementations. The scenario where each modifier was demonstrated as follows:

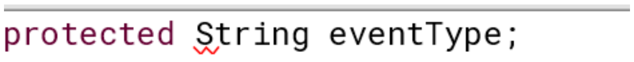
1.private: The process of protecting internal data or used to hide internal details of class and couldn’t be accessed directly outside of the class is private access modifier. It provides unauthorized access and changes to variables from outside. For example, in the passwordChangeEvent class implement the Event interface, where we have used access modifier as private to secure internal data like event id and createdTimestamp.



```
private final String id;
private final long createdTimestamp;
```

Figure 21:Example of private access modifier

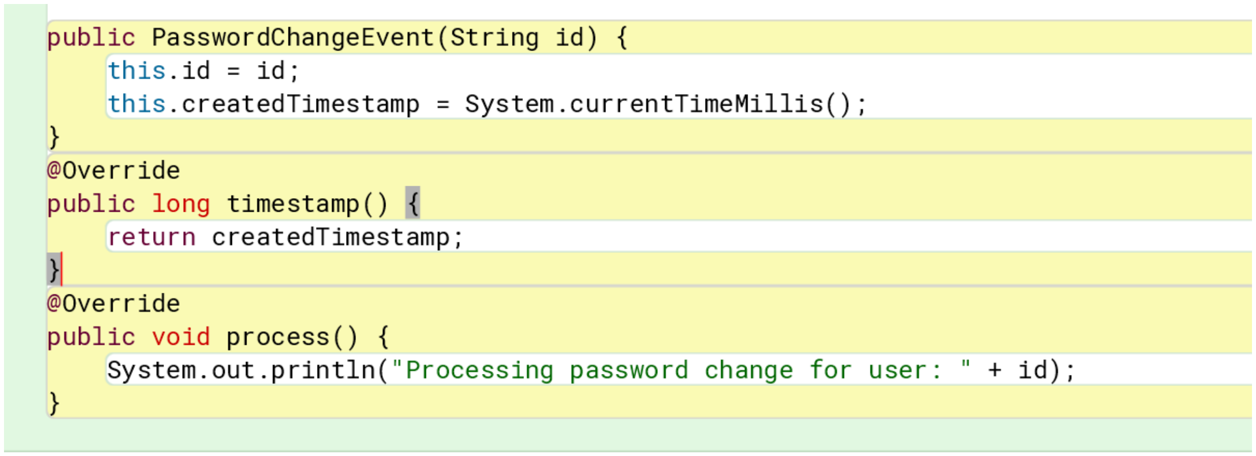
2.protected: The process of modifier allows access to members with the same package or from subclasses is protected access modifier. It is also known as inheritable but controlled. It's helpful for designing the class hierarchies, like base event class that share properties to other class with specific events. For example, Hypothetically If we have abstract class BaseEvent then multiple events inherited from like MissedPayementEvent, AccountTransferEvent, a filed like eventType be protected so subclasses can use it. This makes a code that can be reuse and customized event related behavior in child classes.



```
protected String eventType;
```

Figure 22: Example of protected access modifier

3.public: The modifier used to make method, fields or variables must be accessed from anywhere such as without defining extra getter setter methods or methods that define core behavior or are part of an interface. It exposes only necessary behavior to the outside world. For example, the Event interface defines two methods as a public and uses @override to define its behavior for these methods . These methods are public, so they can be accessed across all the event types.



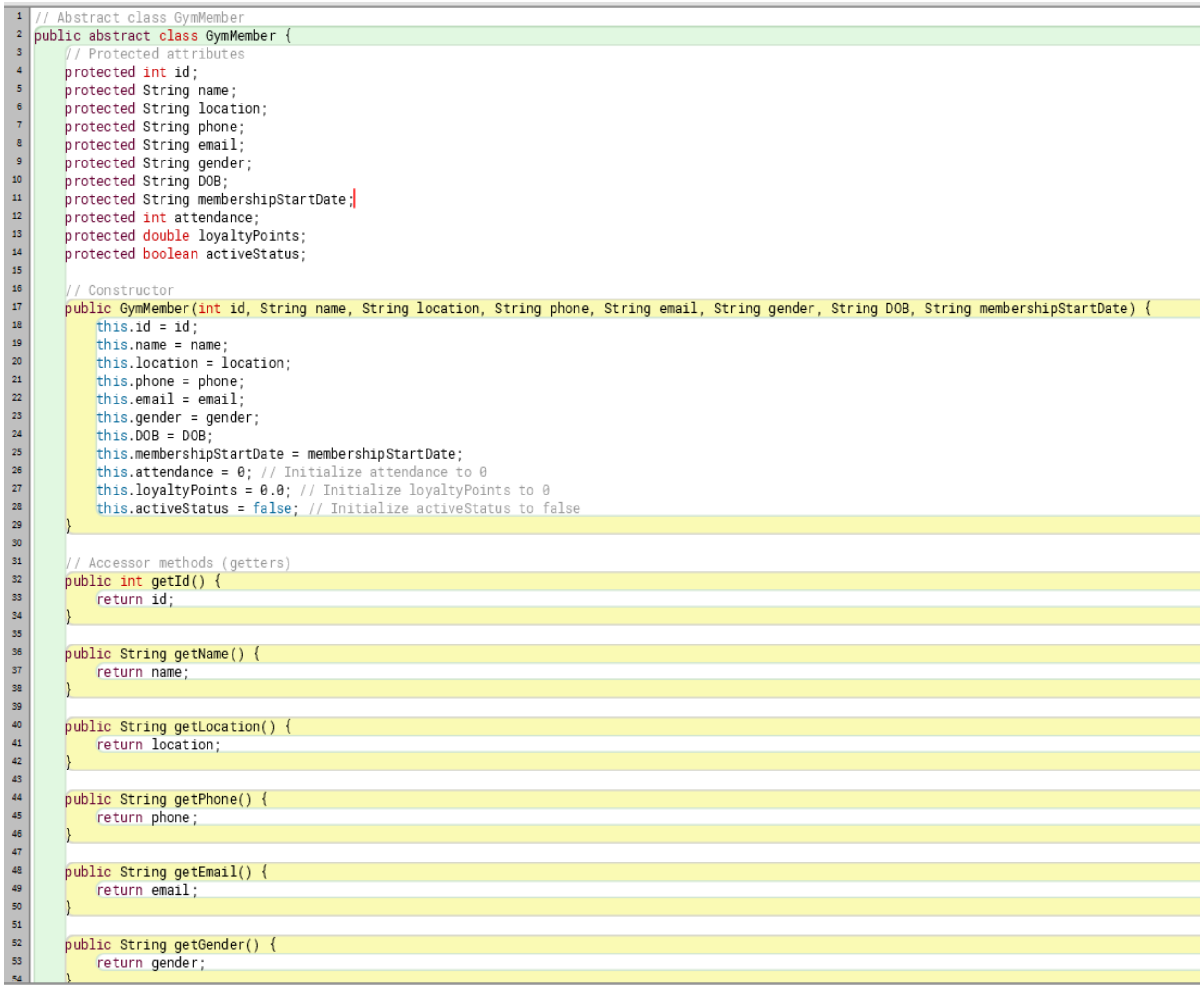
```
public PasswordChangeEvent(String id) {
    this.id = id;
    this.createdTimestamp = System.currentTimeMillis();
}
@Override
public long timestamp() {
    return createdTimestamp;
}
@Override
public void process() {
    System.out.println("Processing password change for user: " + id);
}
```

Figure 23: Example of public access modifier

4.a. How did the course explain the concept of abstraction? Give an example of how you applied abstraction in one of your projects.

Ans: In Likened Learning, abstraction is introduced as the pillars of OOP (object-oriented programming language. The process of hiding the complex details of methods and showing only the required features to client interface is called Abstraction. This allows the programmer to understand what an object or system can do to their task. It is also known as hiding the implementation.

Abstraction is mostly used in interfaces and abstract classes. A class is defined as blueprint for other classes including methods where abstract classes have abstract methods without implementation that must be implemented by subclasses.



```java
// Abstract class GymMember
public abstract class GymMember {
    // Protected attributes
    protected int id;
    protected String name;
    protected String location;
    protected String phone;
    protected String email;
    protected String gender;
    protected String DOB;
    protected String membershipStartDate;
    protected int attendance;
    protected double loyaltyPoints;
    protected boolean activeStatus;

    // Constructor
    public GymMember(int id, String name, String location, String phone, String email, String gender, String DOB, String membershipStartDate) {
        this.id = id;
        this.name = name;
        this.location = location;
        this.phone = phone;
        this.email = email;
        this.gender = gender;
        this.DOB = DOB;
        this.membershipStartDate = membershipStartDate;
        this.attendance = 0; // Initialize attendance to 0
        this.loyaltyPoints = 0.0; // Initialize loyaltyPoints to 0
        this.activeStatus = false; // Initialize activeStatus to false
    }

    // Accessor methods (getters)
    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public String getLocation() {
        return location;
    }

    public String getPhone() {
        return phone;
    }

    public String getEmail() {
        return email;
    }

    public String getGender() {
        return gender;
```

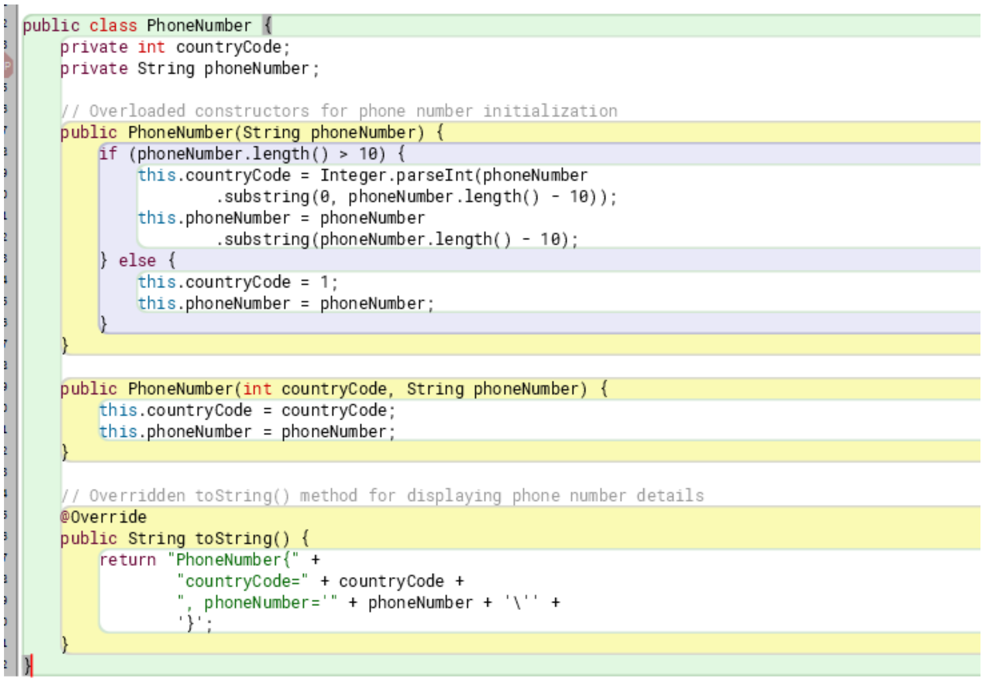
Figure 24:Example of abstract class

The main reason why I applied abstraction to my project is GymMember Class. This class acts as a blueprint of many classes such as RegularMember and PremiumMember, the abstract class allows me to add the same functionality to Both classes. The benefits of abstraction are simplified interaction, flexibility for further changes, and Code reusability.

4.b. Provide a real-world example where polymorphism would be useful in a Java application. How was polymorphism implemented in one of the course exercises?

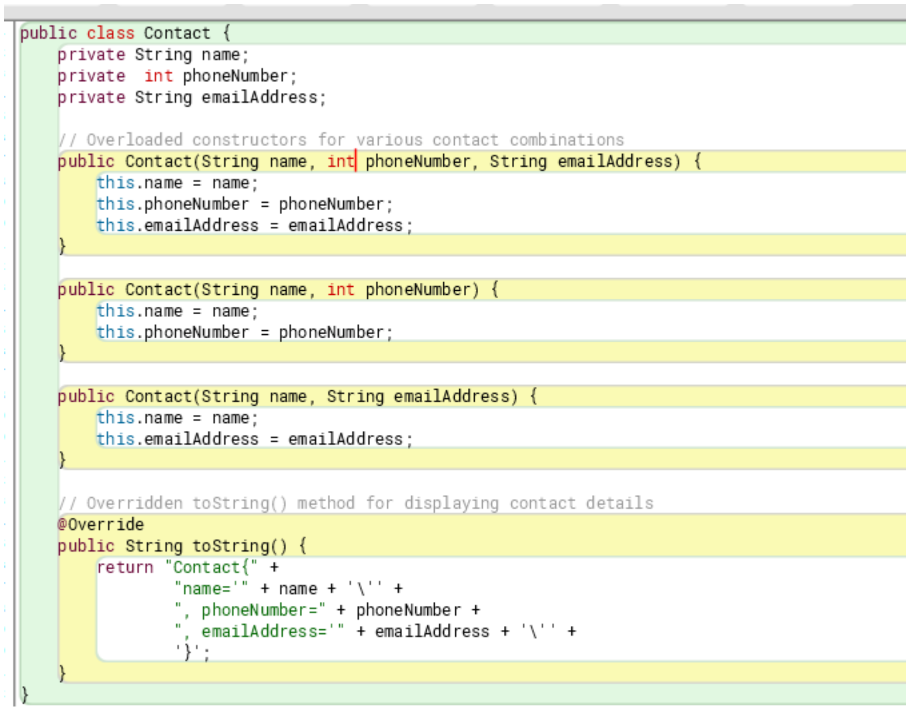
Ans: As we all know, Polymorphism is also pillars of OOP (object-oriented programming. It allows objects of different types to be treated the same as objects of common super class. It also refers to the ability of an entity to make multiple methods. It enhances the reusability of code and flexibility too.

In a real-world java application, polymorphism is mostly useful especially in scenarios like managing various types of names, address, contacts. For example, suppose we must store the details of the company such as name, contact, email then polymorphism would allow us to these objects in a very proper way even after they have different types if objects are represented differently. Overall, polymorphism is the concept to simplifies working with different objects in Java.



```java
public class PhoneNumber {
    private int countryCode;
    private String phoneNumber;

    // Overloaded constructors for phone number initialization
    public PhoneNumber(String phoneNumber) {
        if (phoneNumber.length() > 10) {
            this.countryCode = Integer.parseInt(phoneNumber
                    .substring(0, phoneNumber.length() - 10));
            this.phoneNumber = phoneNumber
                    .substring(phoneNumber.length() - 10);
        } else {
            this.countryCode = 1;
            this.phoneNumber = phoneNumber;
        }
    }

    public PhoneNumber(int countryCode, String phoneNumber) {
        this.countryCode = countryCode;
        this.phoneNumber = phoneNumber;
    }

    // Overridden toString() method for displaying phone number details
    @Override
    public String toString() {
        return "PhoneNumber{" +
                "countryCode=" + countryCode +
                ", phoneNumber='" + phoneNumber + '\'' +
                '}';
    }
}
```

Figure 25:Example of polymorphism 1



```java
public class Contact {
    private String name;
    private  int phoneNumber;
    private String emailAddress;

    // Overloaded constructors for various contact combinations
    public Contact(String name, int phoneNumber, String emailAddress) {
        this.name = name;
        this.phoneNumber = phoneNumber;
        this.emailAddress = emailAddress;
    }

    public Contact(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
    }

    public Contact(String name, String emailAddress) {
        this.name = name;
        this.emailAddress = emailAddress;
    }

    // Overridden toString() method for displaying contact details
    @Override
    public String toString() {
        return "Contact{" +
                "name='" + name + '\'' +
                ", phoneNumber=" + phoneNumber +
                ", emailAddress='" + emailAddress + '\'' +
                '}';
    }
}
```

Figure 26:Example of polymorphism 2

In the course, polymorphism was implemented in the Contact and PhoneNumber class. Contact is demonstrated as constructor overloading (method overloading) as polymorphism. The class can handle multiple combinations of attributes. Method Overriding is also used in the exercise. The method overriding toString () in both Contact and PhoneNumber. It handles multiple phone number format using the overloading method in the phoneNumber class.

5.A. Discuss how exception handling is implemented in Java. What are the benefits of using exception handling in OOP?

Ans: In java, the process of getting the runtime error or the process of detection of error and ensuring the program distributed is known as exceptional handling. It provides well organized exceptional handling though five keywords: throw, throws, try, catch, and finally. Well mostly try to catch keywords is used for exceptional handling.

1.try block: The code inside the try block check if the error is enclosed or not.

2.catch block: If an error occurs in try block the catch block catch the error and display the appropriate error message.

3.finally block: The finally block is used to release the files, database connections and other network sockets. The finally block automatically executed whether try catch or exception is thrown or not.

4.throw statement: Explicitly throw the exception in the code.

5.throws keyword: The method signatures declare that a method might throw one or more exceptions.

The benefits of the exception handling in Object Oriented Programming (OOP) are as follows:

1.Enhanced program robustness.

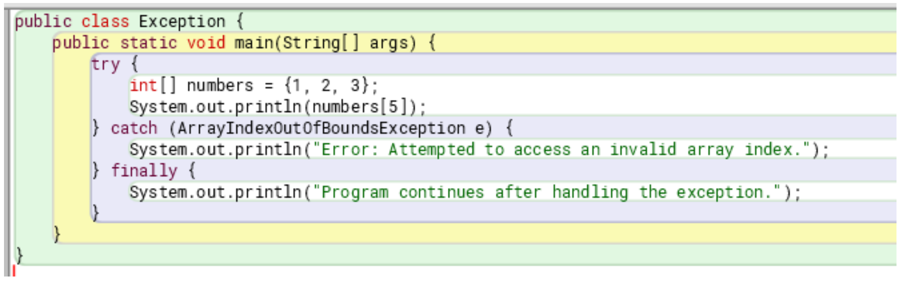
2. separation of error handling code for main logic

3.Clear error information

4.object-oriented design

5.Program Maintainability

6.Assits Debugging and Problem Resolution



```
public class Exception {
    public static void main(String[] args) {
        try {
            int[] numbers = {1, 2, 3};
            System.out.println(numbers[5]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Error: Attempted to access an invalid array index.");
        } finally {
            System.out.println("Program continues after handling the exception.");
        }
    }
}
```

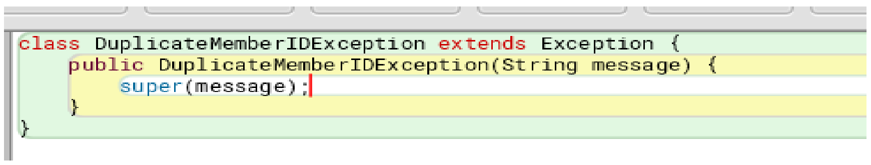
Figure 27:Exception Example

5.B. Provide an example of a custom-except class you might create based on a scenario from the course. How would you implement this exception in your code?

Ans: In the real-world environment applications, error may occur in the program or not. To handle the different kinds of error more effectively we create custom exception classes. It is easier to understand, more organized, and gives helpful output to the user for further processing.

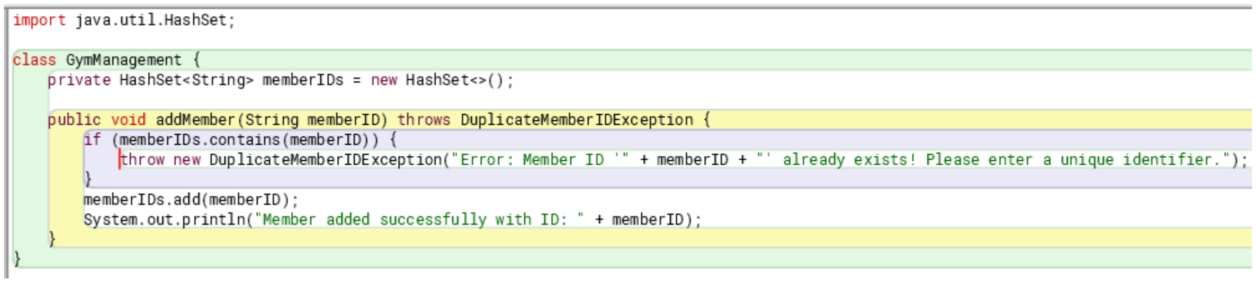
For example, based on the previous course work, the Gym Management system project we have worked on with the exceptional handling. There is a situation to prevent duplicate member id, In the Gui the user should input the unique values to procced or he/she accidentally input the same value then the system will show the exception.

To show the exception when user gives invalid input, we have created a class called duplicateMemberIDException. This exception is only for handling the duplicate id. I have used built-in exception function for the custom exception.



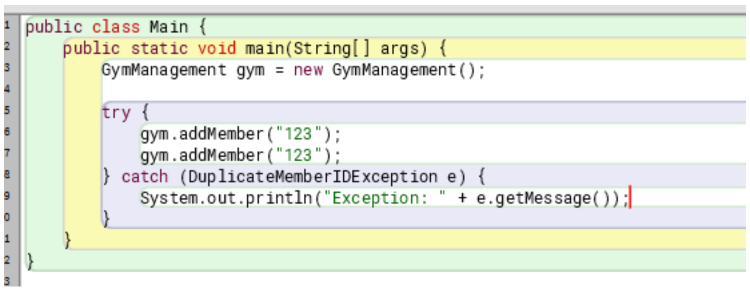
```java
class DuplicateMemberIDException extends Exception {
    public DuplicateMemberIDException(String message) {
        super(message);
    }
}
```

Figure 28:Example of custom exception



```java
import java.util.HashSet;

class GymManagement {
    private HashSet<String> memberIDs = new HashSet<>();

    public void addMember(String memberID) throws DuplicateMemberIDException {
        if (memberIDs.contains(memberID)) {
            throw new DuplicateMemberIDException("Error: Member ID '" + memberID + "' already exists! Please enter a unique identifier.");
        }
        memberIDs.add(memberID);
        System.out.println("Member added successfully with ID: " + memberID);
    }
}
```

Figure 29: Class to check exception



```java
public class Main {
    public static void main(String[] args) {
        GymManagement gym = new GymManagement();

        try {
            gym.addMember("123");
            gym.addMember("123");
        } catch (DuplicateMemberIDException e) {
            System.out.println("Exception: " + e.getMessage());
        }
    }
}
```

Figure 30:Main method to run the exception

6. How would you approach designing a Java application for a library management system using OOP principles? Outline the classes, objects, and interactions you would consider. (Identify key components, explain interactions and relationships among classes, consider additional features, provide sample code snippets).

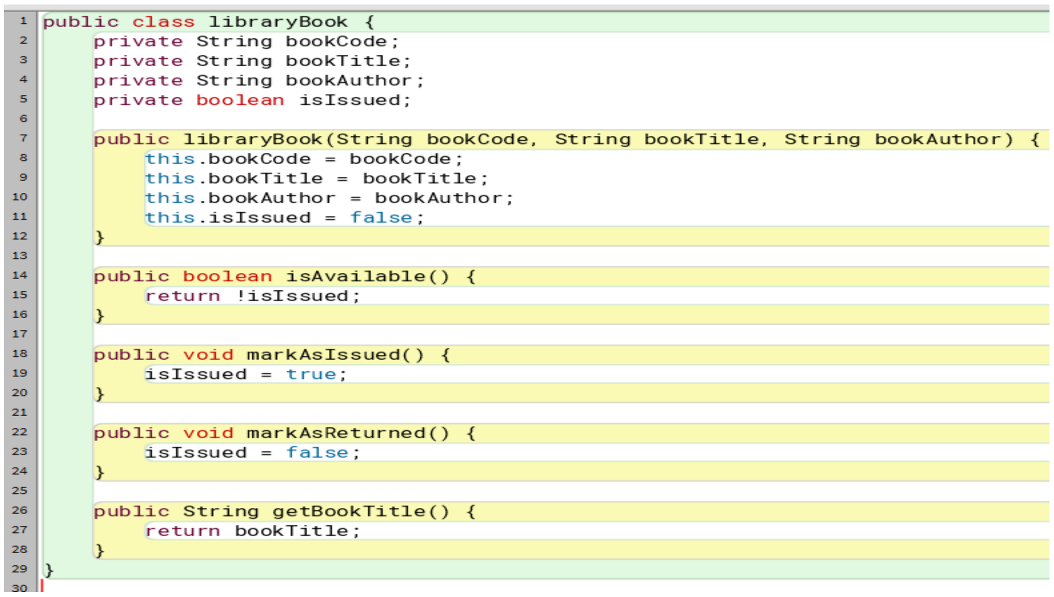
Ans: Java is a classic and real-world application of object-oriented programming languages. Designing a java application for a library management system using OOP principles like Encapsulation, Abstraction, Polymorphism, and inheritance. The main aim is to build software to manage books, manage members, and borrow transactions.

1. Classes and Objects

1. libraryBook class:
    - It represents books in the library.
    - Key attributes: bookcode, bookTitle, bookAuthor, isIssued.
    - Purpose: Track the books information and availability.
2. libraryUser Class:
    - show general user
    - Key attributes: userId, userName
    - Purpose: User can borrow and return books

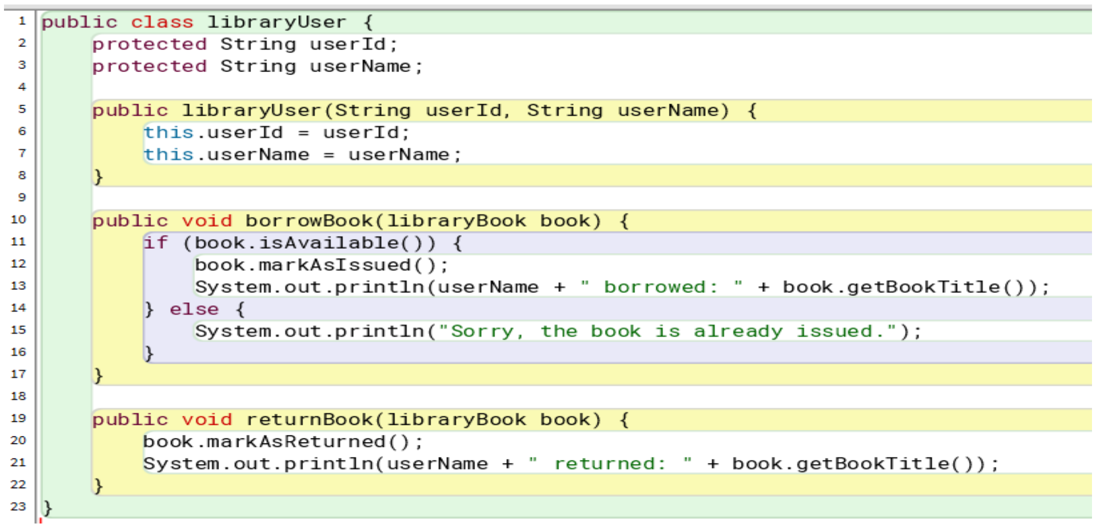
3. libraryAdmin Class (inherits from libraryUser):
    - Add books to the library
    - Inherits normal features but have extra permissions.
4. LibrarySystem Class:
    - Admin of the Library
    - Key Attributes: List of books and users
    - Purpose: Add users, add books, manage book borrow and returning.

1. Interactions and Relationships
   - A user can borrow and return books from the library
   - The library class manages books and user records
   - Only the librarian manages books inventory.
   - Manages logs of transactions.



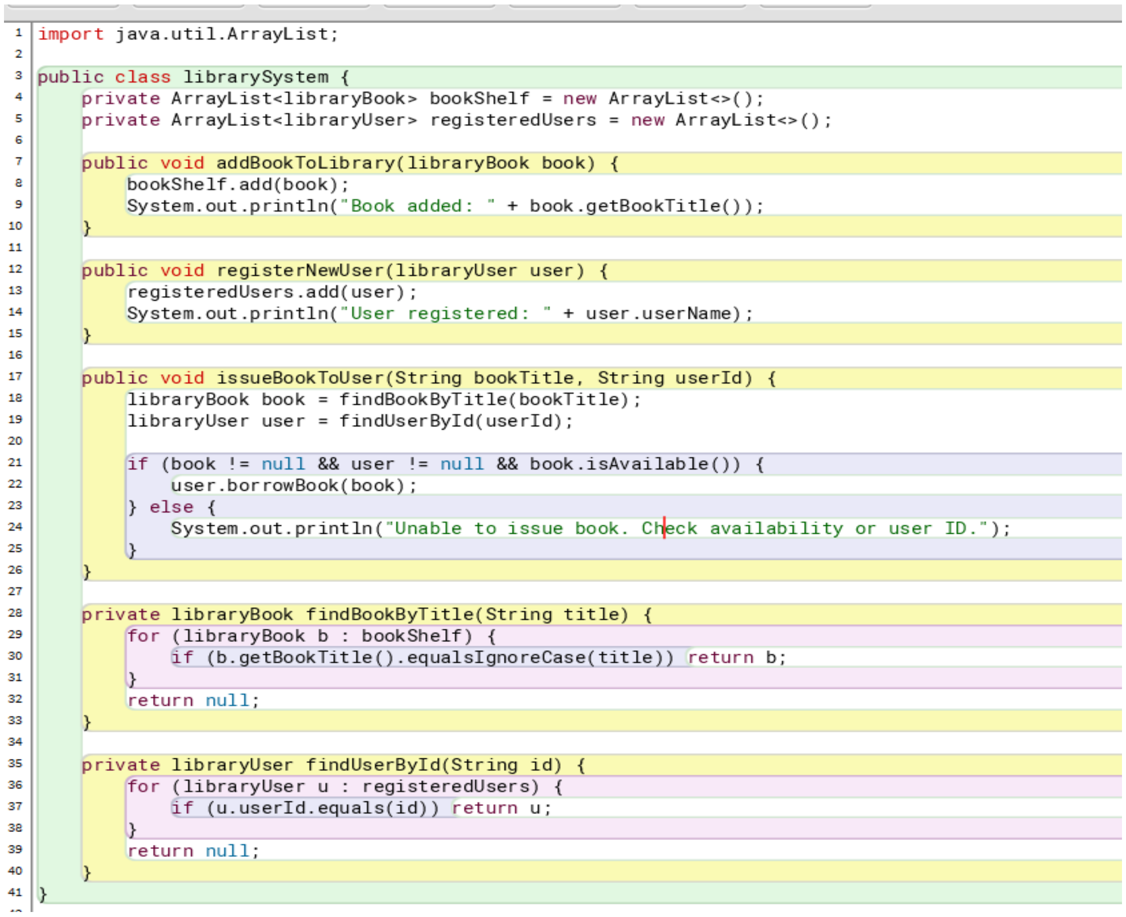
```java
public class libraryBook {
	private String bookCode;
	private String bookTitle;
	private String bookAuthor;
	private boolean isIssued;

	public libraryBook(String bookCode, String bookTitle, String bookAuthor) {
		this.bookCode = bookCode;
		this.bookTitle = bookTitle;
		this.bookAuthor = bookAuthor;
		this.isIssued = false;
	}

	public boolean isAvailable() {
		return !isIssued;
	}

	public void markAsIssued() {
		isIssued = true;
	}

	public void markAsReturned() {
		isIssued = false;
	}

	public String getBookTitle() {
		return bookTitle;
	}
}
```

Figure 31:libraryBook Class



```java
public class libraryUser {
	protected String userId;
	protected String userName;

	public libraryUser(String userId, String userName) {
		this.userId = userId;
		this.userName = userName;
	}

	public void borrowBook(libraryBook book) {
		if (book.isAvailable()) {
			book.markAsIssued();
			System.out.println(userName + " borrowed: " + book.getBookTitle());
		} else {
			System.out.println("Sorry, the book is already issued.");
		}
	}

	public void returnBook(libraryBook book) {
		book.markAsReturned();
		System.out.println(userName + " returned: " + book.getBookTitle());
	}
}
```

Figure 32: libraryUser Class



```java
import java.util.ArrayList;

public class librarySystem {
    private ArrayList<libraryBook> bookShelf = new ArrayList<>();
    private ArrayList<libraryUser> registeredUsers = new ArrayList<>();

    public void addBookToLibrary(libraryBook book) {
        bookShelf.add(book);
        System.out.println("Book added: " + book.getBookTitle());
    }

    public void registerNewUser(libraryUser user) {
        registeredUsers.add(user);
        System.out.println("User registered: " + user.userName);
    }

    public void issueBookToUser(String bookTitle, String userId) {
        libraryBook book = findBookByTitle(bookTitle);
        libraryUser user = findUserById(userId);

        if (book != null && user != null && book.isAvailable()) {
            user.borrowBook(book);
        } else {
            System.out.println("Unable to issue book. Check availability or user ID.");
        }
    }

    private libraryBook findBookByTitle(String title) {
        for (libraryBook b : bookShelf) {
            if (b.getBookTitle().equalsIgnoreCase(title)) return b;
        }
        return null;
    }

    private libraryUser findUserById(String id) {
        for (libraryUser u : registeredUsers) {
            if (u.userId.equals(id)) return u;
        }
        return null;
    }
}
```

Figure 33:librarySystem Class

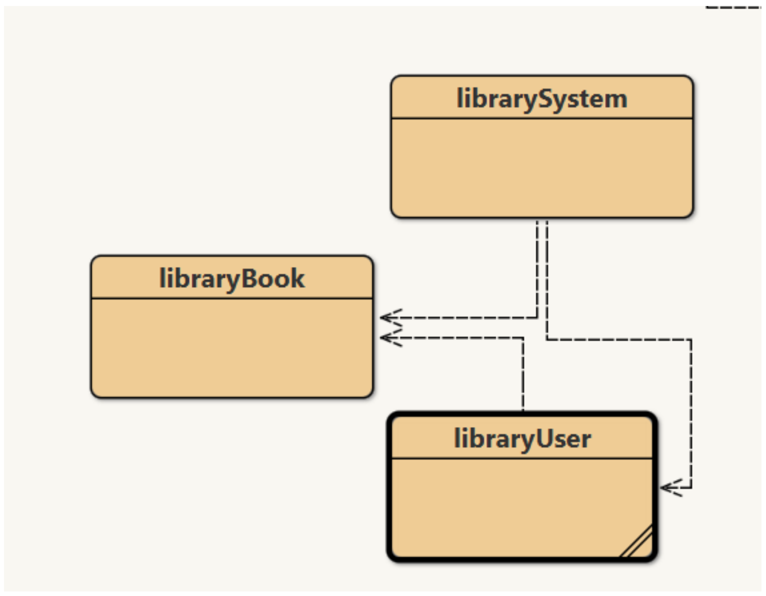
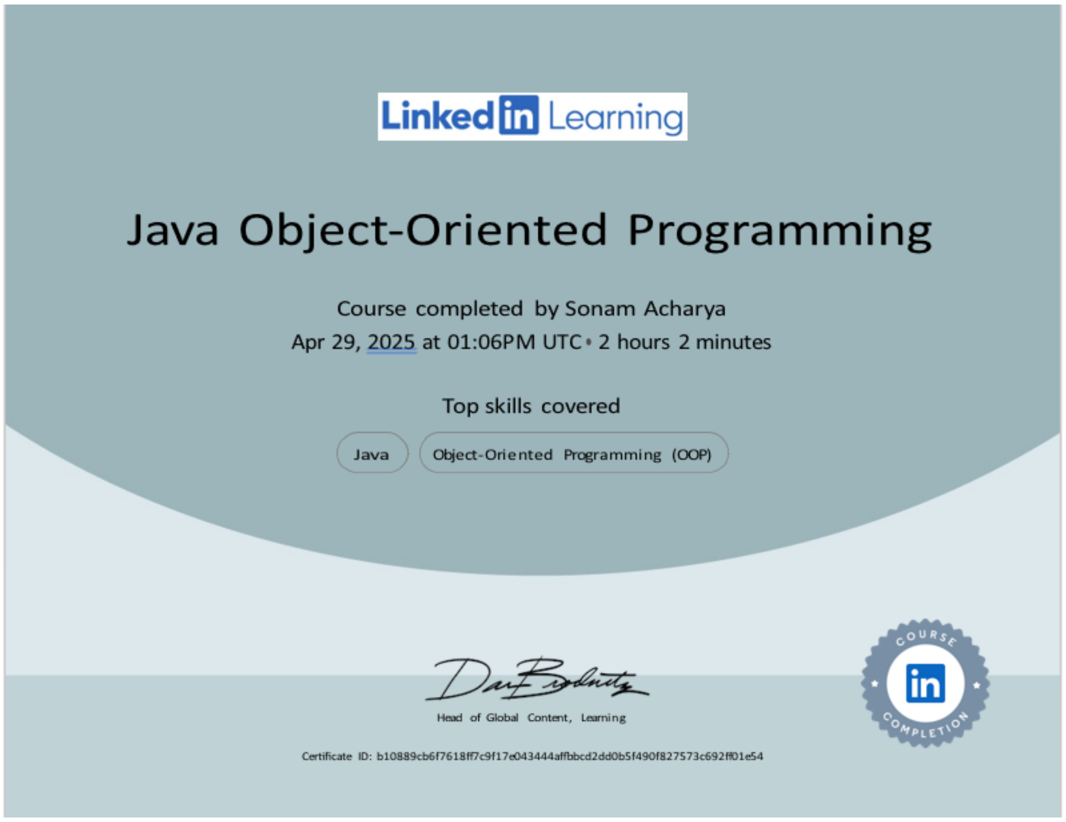


Figure 34:Class Diagram Of librarysystem and librarybook & libraryUser

7. Upon completing the LinkedIn Learning course on Java Object-Oriented Programming, you should have earned a course completion badge. Attach a screenshot or digital copy of your course completion badge that displays your name and the date of completion. In addition, briefly describe a key concept or project from the course that you found most insightful. How did this course help you enhance your understanding of Object-Oriented Programming? Provide specific examples or code snippets to illustrate your point.



Linked in Learning

Java Object-Oriented Programming

Course completed by Sonam Acharya

Apr 29, 2025 at 01:06PM UTC • 2 hours 2 minutes

Top skills covered

Java | Object-Oriented Programming (OOP)

[illegible]

Head of Global Content, Learning

Certificate ID: b10889cb6f7618ff7c9f17e043444affbbcd2dd0b5f490f827573c692ff01e54

COURSE COMPLETION

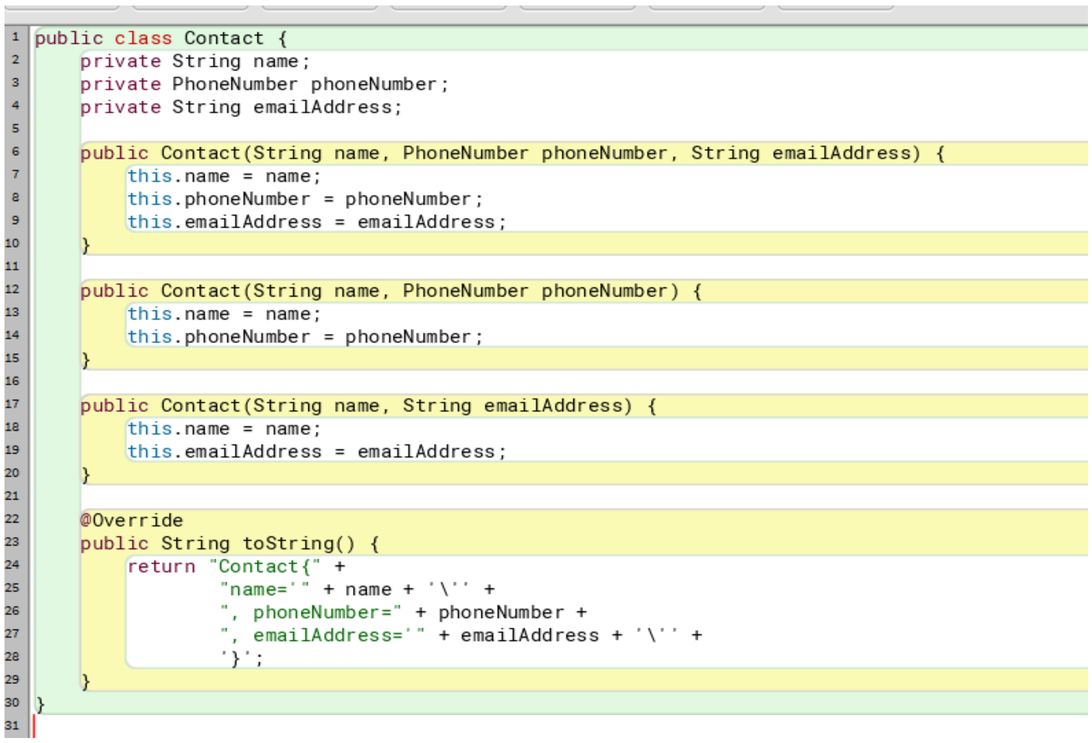
Figure 35: Course Completion Certificate

Ans:

Well, I have completed my java object-oriented course from LinkedIn learning, earned a course completion badge that shows my achievement above.

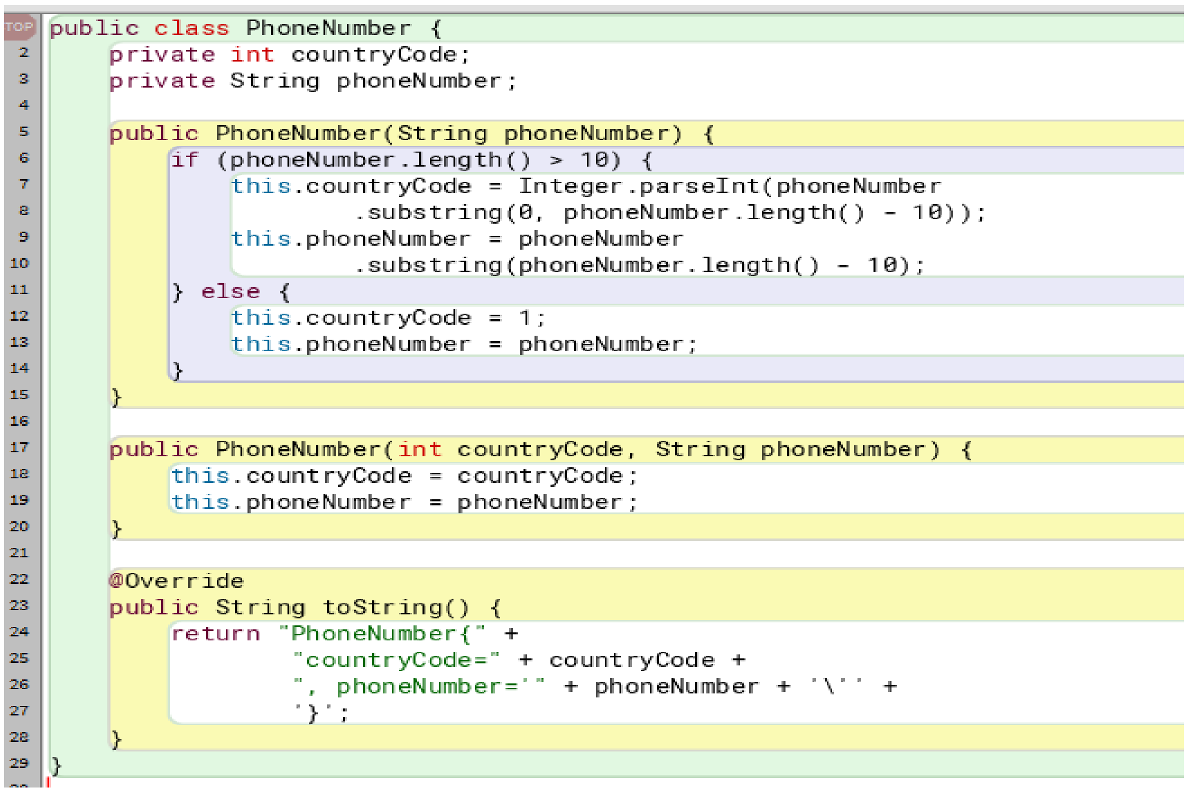
One of the most insightful concepts I get from the course is about the use of polymorphism and inheritance to minimize the code or simply code reuse and extend functionality. During linked, clear example of parent and child classes can work together to create a maintainable system.

- Key concept learned: Inheritance and Method Overriding



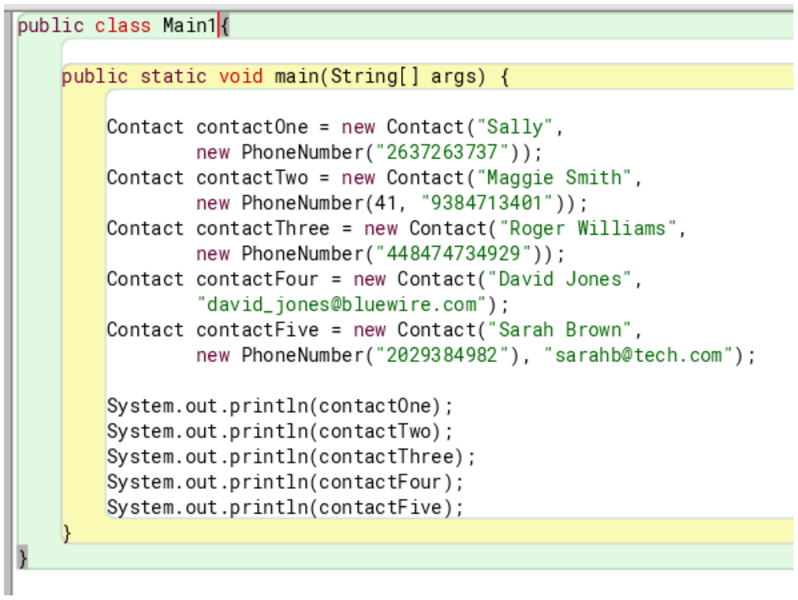
```java
public class Contact {
    private String name;
    private PhoneNumber phoneNumber;
    private String emailAddress;

    public Contact(String name, PhoneNumber phoneNumber, String emailAddress) {
        this.name = name;
        this.phoneNumber = phoneNumber;
        this.emailAddress = emailAddress;
    }

    public Contact(String name, PhoneNumber phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
    }

    public Contact(String name, String emailAddress) {
        this.name = name;
        this.emailAddress = emailAddress;
    }

    @Override
    public String toString() {
        return "Contact{" +
                "name='" + name + '\'' +
                ", phoneNumber=" + phoneNumber +
                ", emailAddress='" + emailAddress + '\'' +
                '}';
    }
}
```

Figure 36: Contact class



```java
public class PhoneNumber {
    private int countryCode;
    private String phoneNumber;

    public PhoneNumber(String phoneNumber) {
        if (phoneNumber.length() > 10) {
            this.countryCode = Integer.parseInt(phoneNumber
                    .substring(0, phoneNumber.length() - 10));
            this.phoneNumber = phoneNumber
                    .substring(phoneNumber.length() - 10);
        } else {
            this.countryCode = 1;
            this.phoneNumber = phoneNumber;
        }
    }

    public PhoneNumber(int countryCode, String phoneNumber) {
        this.countryCode = countryCode;
        this.phoneNumber = phoneNumber;
    }

    @Override
    public String toString() {
        return "PhoneNumber{" +
                "countryCode=" + countryCode +
                ", phoneNumber='" + phoneNumber + '\'' +
                '}';
    }
}
```

Figure 37: Phone number class



```java
public class Main1{

    public static void main(String[] args) {

        Contact contactOne = new Contact("Sally",
                new PhoneNumber("2637263737"));
        Contact contactTwo = new Contact("Maggie Smith",
                new PhoneNumber(41, "9384713401"));
        Contact contactThree = new Contact("Roger Williams",
                new PhoneNumber("448474734929"));
        Contact contactFour = new Contact("David Jones",
                "david_jones@bluewire.com");
        Contact contactFive = new Contact("Sarah Brown",
                new PhoneNumber("2029384982"), "sarahb@tech.com");

        System.out.println(contactOne);
        System.out.println(contactTwo);
        System.out.println(contactThree);
        System.out.println(contactFour);
        System.out.println(contactFive);
    }
}
```

Figure 38: Main class to demonstrate the inheritance and polymorphism

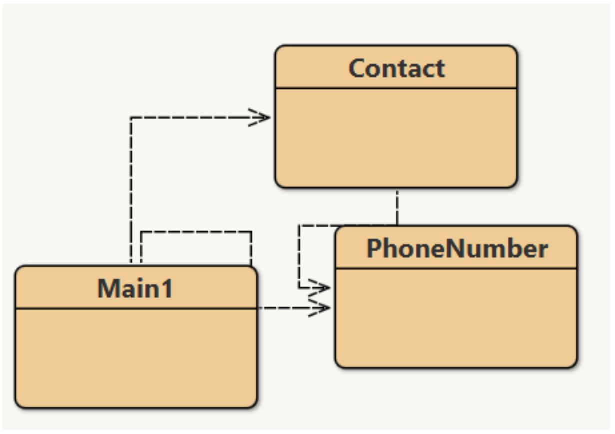


*Figure 39:Cla*SS *diagram of contact phonenumber and main*

In the above example, I have understood how the phoneNumber class could override the toString () method of the contact class to define its own behavior. This helped me to understand polymorphism, where the same method works depending on object type.

The course improved my ability to structure code logically using OOP pillars or principles, to apply real-world scenario, understand the importance of abstraction and encapsulation.

Although, the OOP learning course have enhanced and built my confidence as well as theoretical understanding and practical understanding on application of OOP in java. I feel more confident to create more maintainable java applications.

5.Appendix

5.1. Example of QN.1A.

5.1. Encapsulation example

```
public class BankAccount {
    private double balance = 0;

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

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

5.2. Inheritance example

```
public class Teacher {
    public void teach() {
        System.out.println("Teaching a class...");
    }
}

public class MathTeacher extends Teacher {
    public void solveEquations() {
        System.out.println("Solving math equations...");
    }

    public static void main(String[] args) {
        MathTeacher mTeacher = new MathTeacher();
        mTeacher.teach();
        mTeacher.solveEquations();
    }
}
```

5.3. Abstraction Example

```
public class CoffeeMachine {
    public void makeCoffee() {
        boilWater();
        grindBeans();
        System.out.println("Here is your coffee!");
    }

    private void boilWater() {
        System.out.println("Boiling water...");
    }

    private void grindBeans() {
        System.out.println("Grinding coffee beans...");
    }
}
```

5.3. Polymorphism

```
public class MessageSender {
    public void send(String message) {
        System.out.println("Sending SMS: " + message);
    }

    public void send(String message, String email) {
        System.out.println("Sending Email to " + email + ": " + message);
    }
}
```

5.2. Example of Q.N. 1B.

```
public class Product {
    String name;
    double price;

    public void displayInfo() {
        System.out.println("Product Name: " + name + ", Price: " + price);
    }
}

public class Store {
    public static void main(String[] args) {
        Product p1 = new Product();
        p1.name = "Asus Laptop";
        p1.price = 88999.99;  // Corrected formatting error from screenshot
        p1.displayInfo();
    }
}
```

5.3. Example of Q.N. 2A

```
public class Person {
    protected String name;
    protected int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}


public class Student extends Person {
    private String studentID;
    private String Course;

    // Constructor
    public Student(String name, int age, String studentID, String Course)
{
        // Call parent constructor using super
        super(name, age);
        this.studentID = studentID;
        this.Course = Course;
    }

    // Overriding the displayInfo method to include student details
    @Override
    public void displayInfo() {
        super.displayInfo(); // Call the parent's method
        System.out.println("Student ID: " + studentID);
        System.out.println("Course Name: " + Course);
    }
}
```

5.4. Example of 2B

```
public interface Event {
   long timestamp();
    void process();
}

import java.sql.Timestamp;
```

```
public class PasswordChangeEvent implements Event {
    private final long createdTimestamp;
    private final String id;

    public PasswordChangeEvent(String id) {
        this.id = id;
        this.createdTimestamp                    =                    new
Timestamp(System.currentTimeMillis()).getTime();
    }
    @Override
    public long timestamp() {
        return createdTimestamp;
    }

    @Override
    public void process() {
        System.out.println("Processing password change for user ID: " + id);
    }
}

import java.sql.Timestamp;

public class AccountTransferEvent implements Event {
    private final long createdTimestamp;
    private final String id;

    // Constructor
    public AccountTransferEvent(String id) {
        this.id = id;
        this.createdTimestamp                    =                    new
Timestamp(System.currentTimeMillis()).getTime();
    }

    // Implementing timestamp() method from Event interface
    @Override
    public long timestamp() {
        return createdTimestamp;
    }

    // Implementing process() method from Event interface
    @Override
    public void process() {
        System.out.println("Processing  account  transfer  for  user  ID:  "  +
id);
    }
}

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

        try {
            gym.addMember("123");
            gym.addMember("123");
        } catch (DuplicateMemberIDException e) {
            System.out.println("Exception: " + e.getMessage());
        }
    }
}
```

5.5. Example of QN.3A.

```
public class BankAccount {
    // Private variables (cannot be accessed directly from outside)
    private String accountHolderName;
    private double balance;
    public BankAccount(String name, double initialBalance) {
        this.accountHolderName = name;
        this.balance = initialBalance;
    }

    // Public method to deposit money
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    // Public method to withdraw money
    public void withdraw(double amount) {
        if (amount > 0 && balance >= amount) {
            balance -= amount;
        }
    }

    // Public method to check balance
    public double getBalance() {
        return balance;
    }

    // Public method to get account holder's name
    public String getAccountHolderName() {
        return accountHolderName;
    }
}
```

5.6. Example of QN. 3B

```
import java.sql.Timestamp;
public class PasswordChangeEvent implements Event {
    private final long createdTimestamp;
    private final String id;

    public PasswordChangeEvent(String id) {
        this.id = id;
        this.createdTimestamp                    =                    new
Timestamp(System.currentTimeMillis()).getTime();
    }
    @Override
    public long timestamp() {
        return createdTimestamp;
    }

    @Override
    public void process() {
        System.out.println("Processing password change for user ID: " + id);
    }
}
```

5.7. Example of abstract class Qs.NO.4A.

```
/**
 * Abstract class representing a general gym member.
 * This class provides a base structure shared by both Regular and Premium
members.
 * It includes common attributes and functionality such as identification,
contact details,
 * attendance tracking, membership activation/deactivation, and display of
member information.
 *
 * Author: Sonam Acharya
 */
public abstract class GymMember {

    // Protected attributes accessible to subclasses
    protected int id;
    protected String name;
    protected String location;
    protected String phone;
    protected String email;
    protected String gender;
    protected String DOB;
    protected String membershipStartDate;
    protected int attendance;
    protected double loyaltyPoints;
    protected boolean activeStatus;

    /**
     * Constructor to initialize a GymMember with required personal and
membership details.
     * The attendance and loyalty points are set to 0, and the member is
inactive by default.
     *
     * @param id                     Unique identifier for the member
     * @param name                   Member's full name
     * @param location               City or locality of the member
     * @param phone                  Contact phone number
     * @param email                  Email address
     * @param gender                 Gender identity
     * @param DOB                    Date of birth
     * @param membershipStartDate   Membership start date
     */
    public GymMember(int id, String name, String location, String phone,
String email, String gender, String DOB, String membershipStartDate) {
        this.id = id;
        this.name = name;
        this.location = location;
        this.phone = phone;
        this.email = email;
        this.gender = gender;
        this.DOB = DOB;
        this.membershipStartDate = membershipStartDate;
        this.attendance = 0;
        this.loyaltyPoints = 0.0;
        this.activeStatus = false;
    }

    // Getter methods to access individual member attributes
    public int getId() { return id; }
    public String getName() { return name; }
    public String getLocation() { return location; }
    public String getPhone() { return phone; }
    public String getEmail() { return email; }
    public String getGender() { return gender; }
    public String getDOB() { return DOB; }
    public String getMembershipStartDate() { return membershipStartDate; }
    public int getAttendance() { return attendance; }
    public double getLoyaltyPoints() { return loyaltyPoints; }
    public boolean isActiveStatus() { return activeStatus; }

    /**
     * Abstract method that must be implemented by subclasses.
     * Used to record a member's attendance based on their type.
     */
    public abstract void markAttendance();

    /**
     * Activates the membership of a gym member if not already active.
     * Sets the activeStatus flag to true and provides feedback via the
console.
     */
    public void activateMembership() {
        if (!activeStatus) {
            activeStatus = true;
            System.out.println("Membership activated for " + name);
        } else {
            System.out.println("Membership is already active for " + name);
        }
    }

    /**
     * Deactivates the membership of a gym member if currently active.
     * Sets the activeStatus flag to false and provides confirmation via
the console.
     */
    public void deactivateMembership() {
        if (activeStatus) {
            activeStatus = false;
            System.out.println("Membership deactivated for " + name);
        } else {
            System.out.println("Membership is already inactive for " +
name);
        }
    }

    /**
     * Resets member's status by setting activeStatus to false,
     * attendance to 0, and loyalty points to 0.0. Used for reverting a
member.
     */
    public void resetMember() {
        activeStatus = false;
        attendance = 0;
        loyaltyPoints = 0.0;
        System.out.println("Member " + name + " has been reset.");
    }

    /**
     * Displays all member details to the console including ID, name,
     * contact information, membership dates, attendance, loyalty points,
     * and current active status.
     */
    public void display() {
        System.out.println("Member Details:");
        System.out.println("ID: " + id);
        System.out.println("Name: " + name);
        System.out.println("Location: " + location);
        System.out.println("Phone: " + phone);
        System.out.println("Email: " + email);
        System.out.println("Gender: " + gender);
        System.out.println("Date of Birth: " + DOB);
        System.out.println("Membership      Start      Date:      "      +
membershipStartDate);
        System.out.println("Attendance: " + attendance);
        System.out.println("Loyalty Points: " + loyaltyPoints);
        System.out.println("Active Status: " + (activeStatus ? "Active" :
"Inactive"));
    }
}
```

5.8. Example of Qs.4B.

```java
public class Contact {
    private String name;
    private  int phoneNumber;
    private String emailAddress;

    // Overloaded constructors for various contact combinations
    public Contact(String name, int phoneNumber, String emailAddress) {
        this.name = name;
        this.phoneNumber = phoneNumber;
        this.emailAddress = emailAddress;
    }

    public Contact(String name, int phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
    }

    public Contact(String name, String emailAddress) {
        this.name = name;
        this.emailAddress = emailAddress;
    }

    // Overridden toString() method for displaying contact details
    @Override
    public String toString() {
        return "Contact{" +
                "name='" + name + '\'' +
                ", phoneNumber=" + phoneNumber +
                ", emailAddress='" + emailAddress + '\'' +
                '}';
    }
}
public class PhoneNumber {
    private int countryCode;
    private String phoneNumber;

    // Overloaded constructors for phone number initialization
    public PhoneNumber(String phoneNumber) {
        if (phoneNumber.length() > 10) {
            this.countryCode = Integer.parseInt(phoneNumber
                    .substring(0, phoneNumber.length() - 10));
            this.phoneNumber = phoneNumber
                    .substring(phoneNumber.length() - 10);
        } else {
            this.countryCode = 1;
            this.phoneNumber = phoneNumber;
        }
    }

    public PhoneNumber(int countryCode, String phoneNumber) {
        this.countryCode = countryCode;
        this.phoneNumber = phoneNumber;
    }

    // Overridden toString() method for displaying phone number details
    @Override
    public String toString() {
        return "PhoneNumber{" +
                "countryCode=" + countryCode +
                ", phoneNumber='" + phoneNumber + '\'' +
                '}';
    }
}
```

5.9. 5A Example of exception

```
public class Exception {
    public static void main(String[] args) {
        try {
            int[] numbers = {1, 2, 3};
            System.out.println(numbers[5]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Error: Attempted to access an invalid array
index.");
        } finally {
            System.out.println("Program continues after handling the
exception.");
        }
    }
}
```

5.10. code snippets of 5b.

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

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

        try {
            gym.addMember("123");
            gym.addMember("123");
        } catch (DuplicateMemberIDException e) {
            System.out.println("Exception: " + e.getMessage());
        }
    }
}


import java.util.HashSet;

class GymManagement {
    private HashSet<String> memberIDs = new HashSet<>();

    public void addMember(String memberID) throws
DuplicateMemberIDException {
        if (memberIDs.contains(memberID)) {
            throw new DuplicateMemberIDException("Error: Member ID '" +
memberID + "' already exists! Please enter a unique identifier.");
        }
        memberIDs.add(memberID);
        System.out.println("Member added successfully with ID: " +
memberID);
    }
}
```

5.11. Code of 6 Library Management system.

```
import java.util.ArrayList;

public class librarySystem {
    private ArrayList<libraryBook> bookShelf = new ArrayList<>();
    private ArrayList<libraryUser> registeredUsers = new ArrayList<>();

    public void addBookToLibrary(libraryBook book) {
        bookShelf.add(book);
        System.out.println("Book added: " + book.getBookTitle());
    }

    public void registerNewUser(libraryUser user) {
        registeredUsers.add(user);
        System.out.println("User registered: " + user.userName);
    }

    public void issueBookToUser(String bookTitle, String userId) {
        libraryBook book = findBookByTitle(bookTitle);
        libraryUser user = findUserById(userId);

        if (book != null && user != null && book.isAvailable()) {
            user.borrowBook(book);
        } else {
            System.out.println("Unable to issue book. Check availability or
user ID.");
        }
    }

    private libraryBook findBookByTitle(String title) {
        for (libraryBook b : bookShelf) {
            if (b.getBookTitle().equalsIgnoreCase(title)) return b;
        }
        return null;
    }

    private libraryUser findUserById(String id) {
        for (libraryUser u : registeredUsers) {
            if (u.userId.equals(id)) return u;
        }
        return null;
    }
}

public class libraryBook {
    private String bookCode;
    private String bookTitle;
    private String bookAuthor;
    private boolean isIssued;

    public   libraryBook(String   bookCode,   String   bookTitle,   String
bookAuthor) {
        this.bookCode = bookCode;
        this.bookTitle = bookTitle;
        this.bookAuthor = bookAuthor;
        this.isIssued = false;
    }

    public boolean isAvailable() {
        return !isIssued;
    }

    public void markAsIssued() {
        isIssued = true;
    }

    public void markAsReturned() {
        isIssued = false;
    }

    public String getBookTitle() {
        return bookTitle;
    }
}

public class libraryUser {
    protected String userId;
    protected String userName;
    public libraryUser(String userId, String userName) {
        this.userId = userId;
        this.userName = userName;
    }

    public void borrowBook(libraryBook book) {
        if (book.isAvailable()) {
            book.markAsIssued();
            System.out.println(userName     +     "     borrowed:     "     +
book.getBookTitle());
        } else {
            System.out.println("Sorry, the book is already issued.");
        }
    }

    public void returnBook(libraryBook book) {
        book.markAsReturned();
        System.out.println(userName + " returned: " + book.getBookTitle());
    }
}
```

5.12. No.7 example code

```
public class Contact {
    private String name;
    private PhoneNumber phoneNumber;
    private String emailAddress;

    public   Contact(String   name,   PhoneNumber   phoneNumber,   String
emailAddress) {
        this.name = name;
        this.phoneNumber = phoneNumber;
        this.emailAddress = emailAddress;
    }

    public Contact(String name, PhoneNumber phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
    }

    public Contact(String name, String emailAddress) {
        this.name = name;
        this.emailAddress = emailAddress;
    }

    @Override
    public String toString() {
        return "Contact{" +
                "name='" + name + '\'' +
                ", phoneNumber=" + phoneNumber +
                ", emailAddress='" + emailAddress + '\'' +
                '}';
    }
}

public class PhoneNumber {
    private int countryCode;
    private String phoneNumber;

    public PhoneNumber(String phoneNumber) {
        if (phoneNumber.length() > 10) {
            this.countryCode = Integer.parseInt(phoneNumber
                    .substring(0, phoneNumber.length() - 10));
            this.phoneNumber = phoneNumber
                    .substring(phoneNumber.length() - 10);
        } else {
            this.countryCode = 1;
            this.phoneNumber = phoneNumber;
        }
    }

    public PhoneNumber(int countryCode, String phoneNumber) {
        this.countryCode = countryCode;
        this.phoneNumber = phoneNumber;
    }

    @Override
    public String toString() {
        return "PhoneNumber{" +
                "countryCode=" + countryCode +
                ", phoneNumber='" + phoneNumber + '\'' +
                '}';
    }
}

public class Main1{

    public static void main(String[] args) {

        Contact contactOne = new Contact("Sally",
                new PhoneNumber("2637263737"));
        Contact contactTwo = new Contact("Maggie Smith",
                new PhoneNumber(41, "9384713401"));
        Contact contactThree = new Contact("Roger Williams",
                new PhoneNumber("448474734929"));
        Contact contactFour = new Contact("David Jones",
                "david_jones@bluewire.com");
        Contact contactFive = new Contact("Sarah Brown",
                new PhoneNumber("2029384982"), "sarahb@tech.com");

        System.out.println(contactOne);
        System.out.println(contactTwo);
        System.out.println(contactThree);
        System.out.println(contactFour);
        System.out.println(contactFive);
    }
```

```
}
```