

PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH
HIGHER SCHOOL OF APPLIED SCIENCES OF TLEMCEM



SECOND-CYCLE DEPARTMENT

Tutorial

Object Oriented Programming

By

Dr Akkacha BEKADDOUR

Preface

This tutorial manual serves as a practical bridge between the abstract principles of Object-Oriented Programming (OOP) and the rigorous demands of technical implementation. While the primary course materials establish the conceptual framework of objects and classes, these exercises are designed to transform that understanding into functional, resilient software.

These exercises are structured to move the student back and forth between the conceptual and the concrete. By engaging with both modeling diagrams and code implementation, students will learn that a class is not just a block of syntax, but a representation of a real-world entity.

The goal is to develop an engineering mindset where design and coding happen together. By practicing both, you will learn to spot logical errors in your models before they become bugs in your code. This dual approach ensures you can build organized, professional systems that are easy to maintain and scale. To support this journey, detailed solutions are provided for every exercise, offering a clear reference to validate your logic and refine your implementation.

Content

Introduction	1
Tutorial 1: Introduction to OOP Concepts	3
Tutorial 2: Introduction to JAVA	5
Tutorial 3: Variable Scope in JAVA	7
Tutorial 4: Arrays and Strings in JAVA	11
Tutorial 5: Introduction to OOP Programming using JAVA	13
Tutorial 6: Inheritance and Abstract Classes in JAVA	16
Tutorial 7: Introduction to UML	19
Tutorial 8: Behavioral Modeling: Use Case Diagrams	21
Tutorial 9: Structural Modeling: Class Diagrams	23
Tutorial 10: Behavioral Modeling: Sequence Diagrams	26
Tutorials Solution	28
Bibliography and Webography	65

Introduction

This manual is the practical companion to the course handout. While the main text provides the theoretical and conceptual foundations of Object-Oriented Programming (OOP), this book is designed for action. It serves as a dedicated laboratory guide where students transition from understanding concepts to engineering functional solutions.

The exercises within this manual are varied to reflect the diverse tasks of a modern engineer. Students will engage in architectural design using UML to create precise blueprints, technical implementation using JAVA to write and optimize functional code, or system integration to bridge the two by translating models into executable software and refining existing logic.

By dedicating this manual to practical application, we emphasize that mastering OOP requires more than just reading theory. Whether a specific task focuses on a visual model or a block of code, the goal remains the same: to develop the technical discipline needed to build robust, scalable systems for the industrial landscape.

This tutorial manual is organized into ten progressive sessions designed to translate theoretical knowledge into practical engineering skill:

Tutorial 1: Introduction to OOP Concepts

We explore the foundational mindset of object logic, learning to perceive systems as collections of interacting objects rather than linear scripts.

Tutorial 2: Introduction to JAVA

This is a hands-on orientation to the JAVA development environment, focusing on the platform's architecture and the execution of basic programs.

Tutorial 3: Variable Scope in JAVA

This session examines the lifecycle and visibility of data within a program, ensuring students understand how memory and access are managed within blocks and methods.

Tutorial 4: Arrays and Strings in JAVA

We study the manipulation of essential data structures, learning to handle sequences of data and text, which are the building blocks of most industrial applications.

Tutorial 5: Introduction to OOP Programming using JAVA

This session focuses on the core transition to implementation, where data and behavior are integrated through the creation of functional classes and objects.

Tutorial 6: Inheritance and Abstract Classes in JAVA

This tutorial focuses on system hierarchy and reusability, explaining how to extend and define abstract classes for complex systems.

Tutorial 7: Introduction to UML

An entry into standardized visual communication, where the objective is to learn the syntax and symbols required for professional architectural modeling.

Tutorial 8: Behavioral Modeling: Use Case Diagrams

We practice capturing functional requirements by modeling the interactions between external actors and the system's internal boundaries.

Tutorial 9: Structural Modeling: Class Diagrams

The focus shifts to the system's backbone by visualizing static structures, defining the detailed relationships and attributes that form a stable architecture.

Tutorial 10: Behavioral Modeling: Sequence Diagrams

The final session covers dynamic logic, where we model the chronological exchange of messages between objects to fulfill specific system workflows.

Tutorial 1 : Introduction to OOP concepts

Exercise 1: Solutions → p. 28

Fill in the gaps with the appropriate technical terms to complete the following statements regarding computer history and architecture.

1. The company that popularized personal computing was _____.
2. The computer that made personal computing legitimate in businesses and industry was the _____.
3. Computers process data through a set of instructions called _____.
4. The main logical units of a computer are the _____, _____, _____, and the _____.
5. The three main types of programming languages are _____, _____, and _____.
6. The program that converts a program written in a high-level language into a machine language program is called a _____.

Exercise 2: Solutions → p. 28

Fill in the gaps with the correct technical commands and file extensions required for the JAVA development lifecycle.

1. The JDK command used to execute a compiled JAVA application is: _____.
2. The JDK command responsible for invoking the JAVA compiler is: _____.
3. Every JAVA source code file must be saved with the following file extension: _____.
4. Upon successful compilation, the JAVA compiler generates a bytecode file ending with the extension: _____.
5. The file produced by the JAVA compiler contains _____, which will be executed by the JAVA Virtual Machine.

Exercise 3: Solutions → p. 28

In the modern industrial landscape, we are surrounded by complex systems. A common example is the wrist watch. Analyze a watch through the lens of the Object-Oriented paradigm by defining how the following technical concepts apply to it:

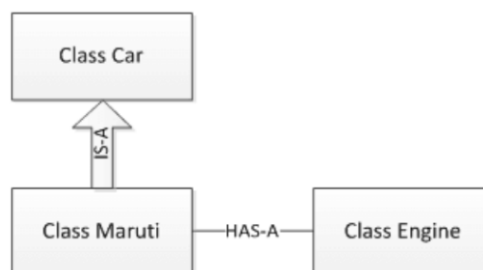
- **Object & Class:** Differentiate between a specific watch on a wrist (the instance) and the general blueprint used by a manufacturer (the template).
- **Attributes & Behaviors:** Identify the static data (e.g., brand, power level) and the dynamic actions (e.g., ticking, resetting the time) of the system.
- **Inheritance:** Using an **alarm clock** as a specialized case, explain how a child class can extend the basic functionality of a standard time-piece.

- **Abstraction:** Describe how a user interacts with the system without needing to understand the internal gears or electronic oscillators.
- **Modeling:** Explain the importance of defining these interactions and structures before the physical assembly or programming begins.
- **Encapsulation & Information Protection:** Discuss how the internal mechanism is shielded from the user and why access to sensitive "data" (the internal gear alignment) is restricted.
- **Messages & Interface:** Define the external controls (the crown, the buttons) that act as the communication bridge between the user and the object's internal logic.

Exercise 4: Solutions → p. 29

Fill in the gaps with the appropriate technical terms to complete the following statements regarding advanced OOP concepts and relationships.

1. The size, shape, color, and weight of an object are considered _____ of the object's class.
2. Objects have the property of _____: even though objects may know how to communicate with each other, they are generally not allowed to know how other objects are implemented.
3. Sending a _____ to an object means asking the object to execute or invoke one of its methods.
4. A _____ is a model or prototype from which objects are created.
5. An _____ is a class that contains only constants and unimplemented methods.
6. With _____, new classes of objects are derived by using the characteristics of existing classes and then adding unique features.
7. The _____ attributes of a superclass can be accessed by that class and all its subclasses.
8. **Relationship Analysis:** Distinguish between an **IS-A** and a **HAS-A** relationship. Using the provided diagram, explain the structural difference between how Maruti relates to Car versus how it relates to Engine.



9. If a class contains an abstract method, then it must be defined as an _____ class.
10. Classes that are fully implemented and allow for the instantiation of objects are called _____ classes.
11. If a subclass extends an abstract class, it must provide a concrete _____ for any inherited abstract methods.
12. Methods that define a signature but do not provide an implementation are known as _____ methods.

Tutorial 2 : Introduction to JAVA

Exercise 1: Solutions → p. 31

Develop a JAVA program that encapsulates a "Identity" or "Profile" concept to display your name. The implementation should utilize a structured class approach rather than a simple procedural script.

Exercise 2: Solutions → p. 31

Write a JAVA program that initializes four distinct variables with the values 1, 2, 3, and 4. Display these values on a single line, separated by spaces, by implementing and comparing three different output methods: `System.out.print()`, `System.out.println()`, and `System.out.printf()`.

Exercise 3: Solutions → p. 31

Create a program designed to compute the product of three user-defined integers. The system should process the input and output the final mathematical result to the console.

Exercise 4: Solutions → p. 32

Implement a JAVA Applet that renders your name within a graphical window. *Note: Ensure the use of the `paint()` method to handle the string rendering.*

Exercise 5: Solutions → p. 32

Develop a program that prompts the user for two real numbers (floating-point). The system must perform a comparative analysis to identify and display the greater value. If the values are identical, the program should output the string: **"EQUALITY"**.

Exercise 6: Solutions → p. 32

Write a JAVA program that accepts three integers as input. The program must calculate and output the following metrics for the dataset:

- The total Sum.
- The Arithmetic Mean (Average).
- The Product.

- The Minimum value (Smallest).
- The Maximum value (Largest).

Exercise 7: Solutions → p. 33

Construct a JAVA program that utilizes iterative loops or formatted strings to generate a mathematical scale table. The output should follow this specific structural alignment:

N	10*N	100*N	1000*N
1	10	100	1000
2	20	200	2000
3	30	300	3000
4	40	400	4000
5	50	500	5000

Exercise 8: Solutions → p. 33

Write a program that allows a user to input a sequence of 10 integers. The program must monitor the input stream in real-time to identify and display the highest value recorded by the end of the session.

Tutorial 3 : Variable Scope in JAVA

Exercise 1: Solutions → p.34

Determine the results of the following JAVA code execution:

```
class Portee {
    public static void main(String[] args) {
        int x = 11, y = 22;
        System.out.println("1. x = " + x);
        System.out.println("2. y = " + y);
        System.out.println();
        calcul1(x, y);
        System.out.println("5. x = " + x);
        System.out.println("6. y = " + y);
        System.out.println();
        calcul2(x, y);
        System.out.println("11. x = " + x);
        System.out.println("12. y = " + y);
        System.out.println();
        int[] object = {5};
        System.out.println("13. object[0] = " + object[0]);
        calcul3(object);
        System.out.println("15. object[0] = " + object[0]);
    }
    static void calcul1(int x, int y) {
        x = 878;          y++;
        System.out.println("3. x = " + x);
        System.out.println("4. y = " + y);
    }
    static void calcul2(int a, int b) {
        int x = 33;        int y = 44;
        a = 2 * x;          b = 2 * y;
        System.out.println("7. x = " + x);
        System.out.println("8. y = " + y);
        System.out.println("9. a = " + a);
        System.out.println("10. b = " + b);
    }
    static void calcul3(int[] object) {
        object[0] = 37;
        System.out.println("14. object[0] = " + object[0]);
    }
}
```

Exercise 2: Solutions → p.34

Determine the results of the following JAVA code execution:

```
class Test {
    static int x = 11;
    private int y = 33;
    public void method1(int x) {
        Test t = new Test();
        this.x = 22;          y = 44;
        System.out.println("Test.x: " + Test.x);
        System.out.println("t.x: " + t.x);
        System.out.println("t.y: " + t.y);
        System.out.println("y: " + y);
    }
    public static void main(String args[]) {
        Test t = new Test();
        t.method1(5);
    }
}
```

Exercise 3: Solutions → p.35

Determine the results of the following main JAVA classes execution:

```
class Test {
    public static void main(String args[]) {
        for (int x = 0; x < 4; x++) {
            System.out.println(x);
        }
        // Will produce error
        System.out.println(x);
    }
}

class Test2 {
    public static void main(String args[]) {
        int a = 5;
        for (int a = 0; a < 5; a++) {
            System.out.println(a);
        }
    }
}
```

Exercise 4: Solutions → p.35

Determine the results of the following JAVA code execution:

```
class Student { // Static keyword usage
    String name;           int rollNo;
    static String cllgName; // static variable
    static int counter = 0; // static counter for unique roll no
    public Student(String name) {
        this.name = name;
        this.rollNo = setRollNo(); // unique rollNo assigned
    }
    // Getting unique rollNo through static counter
    static int setRollNo() {
        counter++;
        return counter;
    }
    // Static method to set college name
    static void setCllg(String name) {
        cllgName = name;
    }
    // Instance method to print student info
    void getStudentInfo() {
        System.out.println("name : " + this.name);
        System.out.println("rollNo : " + this.rollNo);
        System.out.println("cllgName : " + cllgName);
    }
}

class StaticDemo { // main class
    public static void main(String[] args) {
        // calling static method without instantiating Student class
        Student.setCllg("XYZ");
        Student s1 = new Student("Alice");
        Student s2 = new Student("Bob");
        s1.getStudentInfo();
        s2.getStudentInfo();
    }
}
```

Exercise 5: Solutions → p.35

Determine the results of the following JAVA code execution:

```
class Main {
    public static void main(String args[]) {
        final int i;    i = 20;
        System.out.println(i);
    }
}
```

Exercise 6: Solutions → p. 35

Determine the results of the following JAVA code execution:

```
class Base {
    public final void show() {
        System.out.println("Base::show() called");
    }
}
class Derived extends Base {
    public void show() {
        System.out.println("Derived::show() called");
    }
}
```

Exercise 7: Solutions → p. 36

Provide a detailed technical analysis of the final keyword in JAVA. Your response should explicitly define its behavior and constraints when applied to **variables** (including primitives and references), **methods**, and **classes**.

Tutorial 4 : Arrays and Strings in JAVA

Exercise 1: Solutions → p. 38

Using the `JAVA.util.Arrays` class, write a JAVA program that sorts two arrays:

- T1: an array of integers initialized with the values (1789, 2035, 1899, 1456, 2013, 1458, 2458, 1254, 1472, 2365, 1456, 2165, 1457, 2456)
- T2: an array of strings initialized with the values ("JAVA", "Python", "PHP", "C#", "C Programming", "C++")

Exercise 2: Solutions → p. 39

Implement the sorting logic for the integer and string arrays from Exercise N1 manually.

Do not use the `JAVA.util.Arrays` utility class; instead, develop your own sorting algorithm (such as Bubble Sort or Selection Sort) to organize the elements in ascending and alphabetical order.

Exercise 3: Solutions → p. 39

Develop a JAVA program that calculates the arithmetic sum of all elements stored within a predefined integer array. The array should be initialized with the sequence of integers from 1 through 10.

Exercise 4: Solutions → p. 40

Create a JAVA program that retrieves and displays a specific character from a string literal. Initialize the string with the value "**exercice N4**" and output the character located at a user-defined or pre-specified index **x** using string manipulation methods.

Exercise N5: Solutions → p. 40

Determine the results of the following JAVA code execution:

```
class Testimmutablestring {  
    public static void main(String args[]) {  
        String s = "I am";        s.concat(" Hungry");  
        System.out.println(s);  
    }  
}
```

Exercise N6: Solutions → p. 41

Determine the results of the following JAVA code execution:

```
public class Exercise6 {
    public static void main(String[] args) {
        String str = "The quick brown fox jumps over the lazy dog.";
        String temp = "The";
        String new_str1 = str.replace('d', 'f');
        String new_str2 = str.replaceAll("fox", "cat");
        String new_str3 = str.trim();
        boolean A = str.startsWith(temp);

        System.out.println("Original string: " + str);
        System.out.println("New String: " + new_str1);
        System.out.println("New String: " + new_str2);
        System.out.println("New String: " + new_str3);
        System.out.println(str + " starts with " + temp +
            " is " + A);
    }
}
```

Tutorial 5 : Introduction to OOP Programming - JAVA

Exercise 1: Solutions → p. 42

Given the following class Circle:

```
class Circle {
    private double radius;
    private String color;
    public Circle() {
        radius = 1.0;
        color = "red";
    }
    public Circle(double r) {
        radius = r;
        color = "red";
    }
    public double getRadius() {
        return radius;
    }
    public double getArea() {
        return radius * radius * Math.PI;
    }
}
```

In addition, the following class TestCircle:

```
class TestCircle {
    public static void main(String[] args) {
        Circle c1 = new Circle();
        System.out.println("The circle has radius of " +
            c1.getRadius() + " and area of " + c1.getArea());

        Circle c2 = new Circle(2.0);
        System.out.println("The circle has radius of " +
            c2.getRadius() + " and area of " + c2.getArea());
    }
}
```

Give the execution result of the TestCircle class.

Exercise 2: Solutions → p. 42

Write a class OPERATION having two attributes (a character representing one of the four operations and three real variables representing respectively the two operands and the result of the operation). This class has five behaviors: void calcul() allowing to calculate the result of the operation and to store it in the result attribute, the method double get_result() allowing to return the value of the result, the two methods void set_op1(double a) and void set_op2(double b) allowing to change the value of the first operand – respectively the second operand – to a – respectively to b, and the method void set_op(char c) allowing to change the operation to the one contained in c.

Your JAVA program must allow testing the class OPERATION (using the five methods in the first case and using only the first two methods in the second case).

Exercise 3: Solutions → p. 44

Given the following interface:

```
public interface MyInterface {  
    public void sayHello();  
}
```

Propose two implementations — man and cow — of this interface.

Exercise 4: Solutions → p. 44

Provide the code for the MyOtherInterface interface, given the following class:

```
public class MyInterfaceImpl implements MyInterface,  
MyOtherInterface {  
  
    public void sayHello() {  
        System.out.println("Hello");  
    }  
  
    public void sayGoodbye() {  
        System.out.println("Goodbye");  
    }  
}
```

Exercise 5: Solutions → p. 44

What is the difference between the man from Exercise N3 and the following man2?

```
public class man2 extends MyInterface {  
    public void sayHello() {  
        System.out.println(MyInterface.hello);  
    }  
}
```

Exercise 6: Solutions → p. 44

Define a JAVA interface named Command containing two methods, turnOn() and turnOff(), to represent the basic power functions of a controller. Subsequently, implement this interface in two distinct classes: TVRemote, which represents a television controller, and ProjectorRemote, which represents a datashow controller. Both classes should feature a private String attribute called style to represent their specific design state, provide concrete console-print implementations for the interface methods, and include a toString() method that returns a string describing the object's current state.

Tutorial 6 : Inheritance and Abstract Classes in JAVA

Exercise 1: Solutions → p. 46

Let the following two classes:

```
class Car {  
    public void m1() {  
        System.out.println("car 1");  
    }  
    public void m2() {  
        System.out.println("car 2");  
    }  
    public String toString() {  
        return "vrooom";  
    }  
}
```

```
class Truck extends Car {  
    public void m1() {  
        System.out.println("truck 1 " );  
    }  
}
```

1- What is the result of the following instructions:

```
Truck mycar = new Truck();  
System.out.println(mycar);  
mycar.m1();  
mycar.m2();
```

2- Assuming the Truck class has changed to the following:

```
class Truck extends Car {  
    public void m1() {  
        System.out.println("truck 1");  
    }  
    public void m2() {  
        super.m1();  
    }  
    public String toString() {  
        return super.toString() + super.toString();  
    }  
}
```

What will be the result of the following instructions:

```
Truck mycar = new Truck();  
System.out.println(mycar);  
mycar.m1();  
mycar.m2();
```

Exercise 2: Solutions → p. 46

Assuming still the two classes Car and Truck from the first exercise, write a class MonsterTruck that inherits from the Truck class and produces the following output:

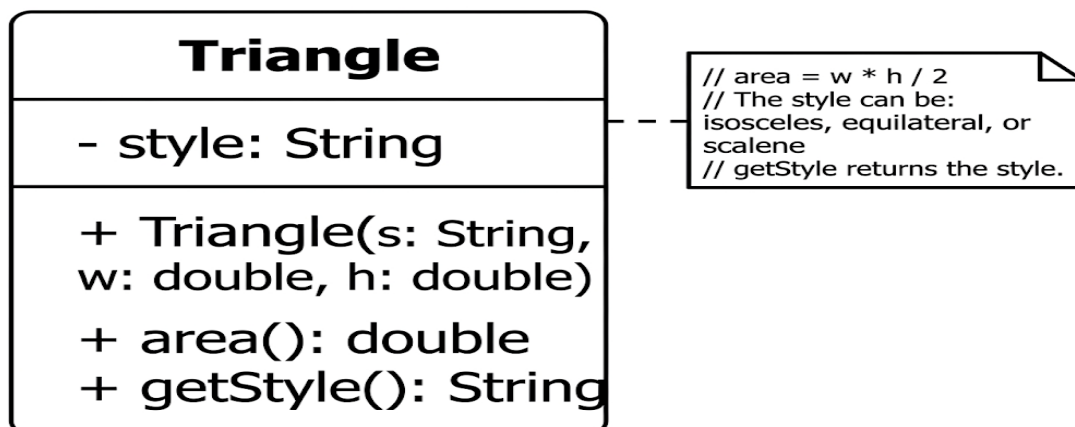
```
MonsterTruck bigfoot = new MonsterTruck();
bigfoot.m1(); // monster 1
bigfoot.m2(); // truck 1 / car 1
System.out.println(bigfoot); // monster vroomvroom
```

Exercise 3: Solutions → p. 46

Let the following class TwoDShape (2D geometric shape) be:

```
class TwoDShape {
    private double width;
    private double height;
    public TwoDShape(double width, double height) {
        this.width = width;
        this.height = height;
    }
    public double getWidth() {
        return width;
    }
    public double getHeight() {
        return height;
    }
    public void print() {
        System.out.println("Width is " + width + " Height is "
+ height);
    }
}
```

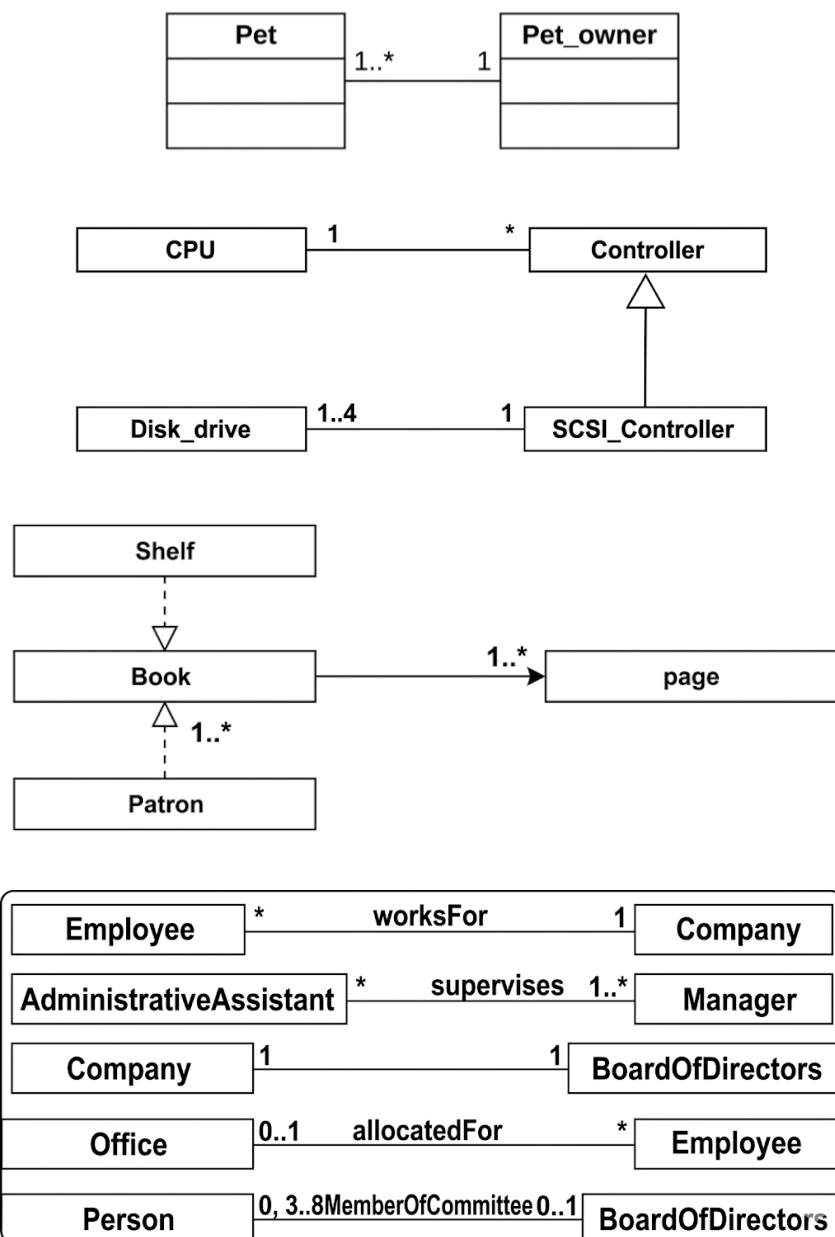
Additionally, let the class Triangle inherit from the class TwoDShape.



- 1- Provide the corresponding JAVA code for the Triangle class.
- 2- Rewrite the TwoDShape class with an abstract method called area, then implement this method in the Triangle class that inherits from TwoDShape. Additionally, both classes should use the toString method to display their dimensions (the toString method of the Triangle class should also display the style of the triangle).

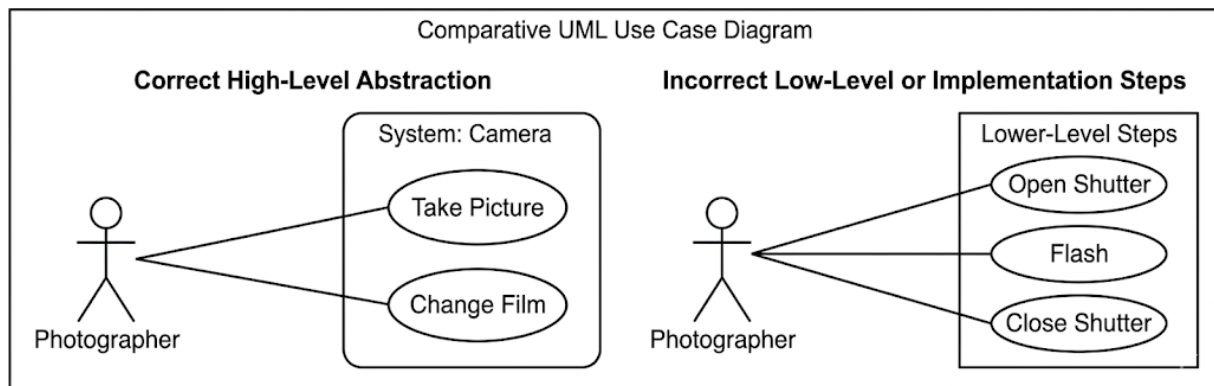
Tutorial 7 : Introduction to UML Solutions → p. 49

- 1- UML is the result of a combination of three modeling methods, which are:, ... and
- 2- UML is a modeling language that allows for, ..., ... and the of elements of a software system.
- 3- Read and explain the following UML models:

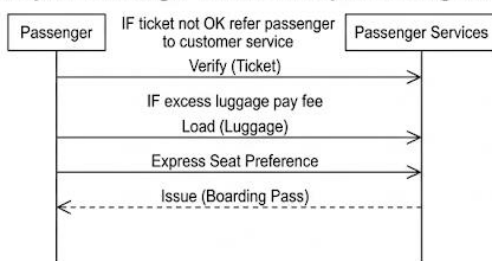


4- The diagram allows capturing the behavior of a system in terms of requirements.

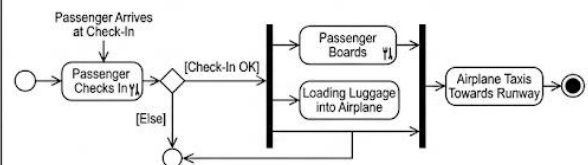
5- Discuss the following diagrams:



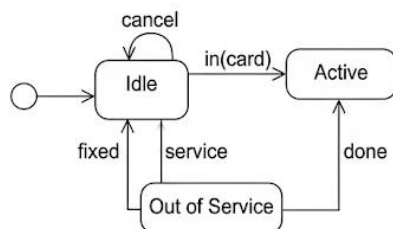
Airport Passenger Check-In Sequence Diagram



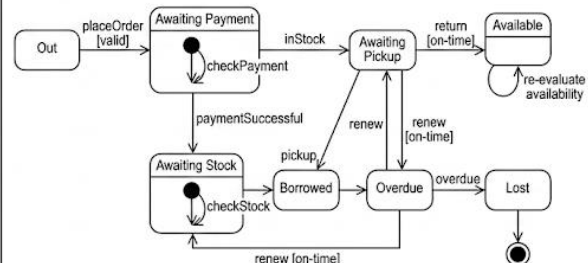
Passenger Boarding Activity Diagram



State Machine: Bank ATM



UML 2.x State Machine: Book State



Tutorial 8 : Behavioral Modeling: Use Case Diagrams

Exercise 1: Solutions → p. 52

Consider a gas station that distributes fuel. Customers pump the fuel themselves, and the gas station attendant fills the tanks.

1. The customer pumps fuel as follows: they take a nozzle attached to a pump and press the trigger to get fuel. Who is the actor in the system: the customer, the nozzle, the pump, the fuel, or the trigger?
2. Ali, whose job is a gas station attendant, wants to pump fuel for his car. To model Ali's activity, should we define a new actor? Draw the corresponding use case diagram.
3. When Ali comes with his tanker truck to fill the pump tanks, is he considered a new actor? Enrich the previous use case diagram.
4. Among the gas station attendants, some are also qualified to perform maintenance operations in addition to their usual duties such as filling tanks. They remain attendants but are called technicians. Further enrich the use case diagram.

Exercise 2: Solutions → p. 53

Consider the following use cases:

Order shipment Complete shipment Partial shipment Order management
Place order Place urgent order User identification

Order management refers to the entire process, from placing an order to shipment. However, it can happen that a placed order is not shipped. Placing an urgent order is a special case of placing an order. To place an order, the user must be validated.

- Propose a use case diagram **without representing the actors**.

Exercise 3: Solutions → p. 54

We want to model an automatic teller machine (ATM). This machine will be used by clients who want to choose an operation among cash withdrawal (quick or normal) and checking the balance of their account. For each operation, the user must be identified. The

ATM should allow printing receipts for each operation if the user wishes. An external central system will verify account balances in the case of a large withdrawal and provide the necessary information in the case of a balance inquiry.

- Propose a use case diagram to model the expected functionalities of the ATM system.

Exercise 4: Solutions → p. 55

We want to design a cash register system. The use of this cash register proceeds as follows:

1. A customer arrives at the register with items.
2. The cashier records the identification number of each item, as well as the quantity if it is greater than 1.
3. The register displays the price and description of each item.
4. When all items have been recorded, the cashier signals the end of the sale.
5. The register displays the total amount of the purchase.
6. The customer chooses their payment method:
 - **Cash:** the cashier collects the money and the register shows the change to be returned to the customer (if any).
 - **Check:** the cashier notes the customer's identity and the register records the amount on the check.
 - **Credit card:** a banking terminal is part of the register and transmits the request to a multi-bank authorization center.
7. The register records the sale and prints a receipt.
8. The cashier hands the printed receipt to the customer.
9. A customer can present discount coupons before payment. When payment is completed, the register sends information about the sold items to the stock management system. Every morning, the store manager initializes the registers for the day.

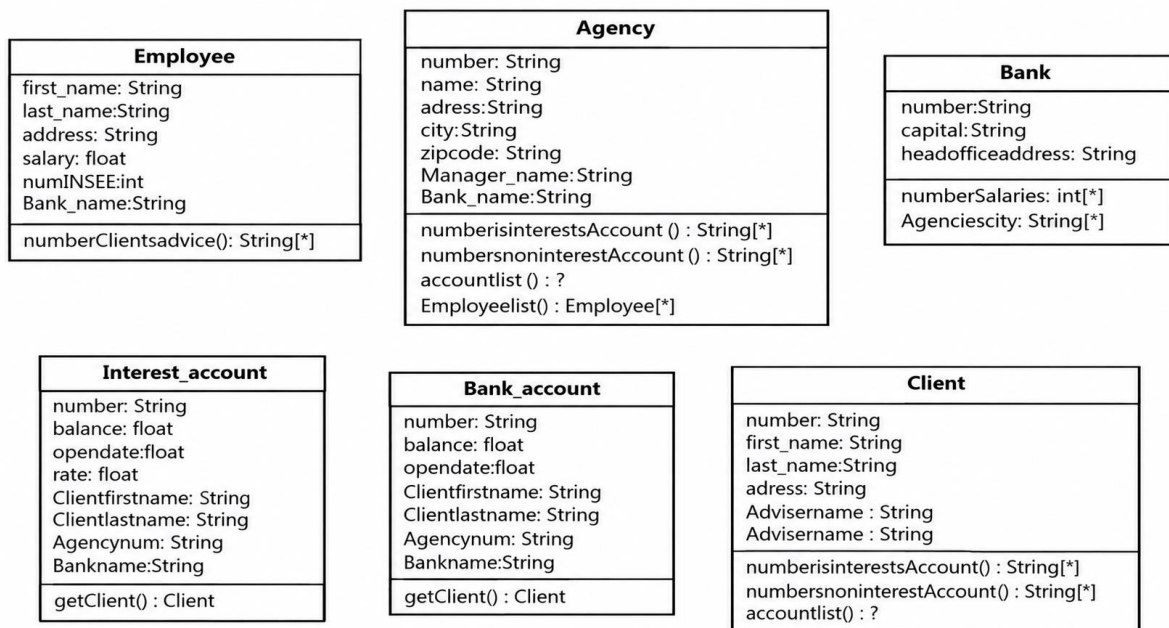
Some of this information is relevant for creating a use case diagram, and some less so.

- Propose a use case diagram to model the expected functionalities of the cash register system.

Tutorial 9 : Structural Modeling: Class Diagrams

Exercise 1: Solutions → p. 56

An initial analysis led to the creation of the following classes for organizing the banks in a consortium. This is only a first draft, which should be carefully reviewed and corrected: some attributes are redundant.



1- Introduce associations between classes to limit the redundancy of information. It will be assumed that the introduced associations give rise to accessors (getters, setters) without the need to include all of them in the diagram.

2- Use inheritance to optimize the model. You may introduce a new class if necessary.

Exercise 2: Solutions → p. 58

For each of the following statements, provide a class diagram. Unless otherwise specified, do not represent attributes or operations.

1. Every writer has written at least one work.
2. A rectangle can be defined by two vertices which are points (in the case where the sides are perfectly vertical and horizontal). A rectangle is constructed from the

coordinates of two points. It is possible to calculate its area and perimeter, or to translate it. Model the attributes and operations.

3. People can be associated with universities both as students and as professors.
4. Cinemas are composed of several screening rooms. Films are shown in rooms. The corresponding screenings each take place at a specific time (you may use an association class).

Exercise 3: Solutions → p. 59

The preliminary study of billing management in a company has made it possible to identify the following business rules:

- Invoices are numbered and dated.
- Each invoice includes a certain number of products. A quantity is associated with each product on an invoice.
- A product is identified by a code, and a label provides a clearer identification.
- The pre-tax price of products must be known. The total price including tax (TTC) is calculated.
- The VAT applicable to each product depends on its type. For each product type, a label and the associated VAT rate are known.
- Invoices must not concern more than one customer. Each one with a name and an address.
- It must be possible to calculate the total amount including tax (TTC) and the total amount excluding tax (HT) of the invoices.
- It must be possible to add and remove products from an invoice. These are the only two ways to modify the contents of an invoice. Moreover, if a product already present on an invoice is added again, the quantity of that product is simply increased.

Provide a class diagram to represent this information.

Exercise 4: Solutions → p. 60

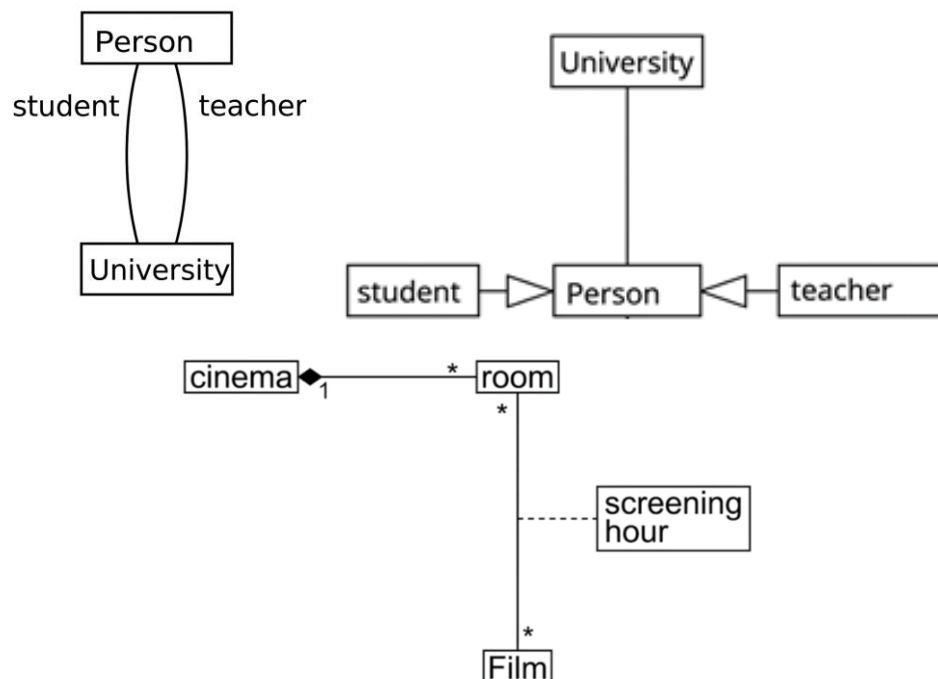
Provide the class diagram corresponding to the following **PointPlan** class:

```
import JAVA.util.Scanner;
class PointPlan {
    private float abscisse;    private float ordonnee;
    public PointPlan() {
        this.abscisse = 0;
        this.ordonnee = 0;
    }
    public PointPlan(float x, float y) {
        this.abscisse = x;
        this.ordonnee = y;
    }
    public PointPlan(PointPlan p) {
        this.abscisse = p.getAbscisse();
        this.ordonnee = p.getOrdonnee();
    }
    private float getOrdonnee() {
        return this.ordonnee;
    }
    private float getAbscisse() {
        return this.abscisse;
    }
    public boolean egaleA(PointPlan p) {
        return (this.getAbscisse() == p.getAbscisse() &&
            this.getOrdonnee() == p.getOrdonnee());
    }
    public void init() {
        Scanner input = new Scanner(System.in);
        System.out.print("Abscisse ? ");
        this.abscisse = input.nextFloat();
        System.out.print("Ordonnée ? ");
        this.ordonnee = input.nextFloat();
    }
    public void translate(float dx, float dy) {
        this.abscisse += dx;    this.ordonnee += dy;
    }
    public double distance() {
        return Math.sqrt(this.abscisse * this.abscisse +
            this.ordonnee * this.ordonnee);
    }
    public String toString() {
        return "(" + this.getAbscisse() + ", " +
            this.getOrdonnee() + ")";
    }
    public void affiche() {
        System.out.println(this.toString());
    }
}
```

Tutorial 10 : Behavioral Modeling: Sequence Diagrams

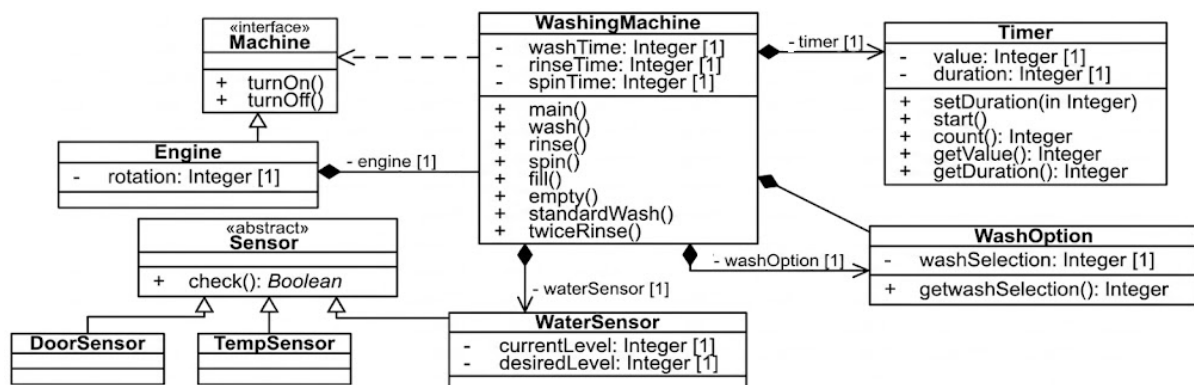
Exercise 1: Solutions → p. 60

Provide the JAVA code corresponding to the following UML class diagrams.



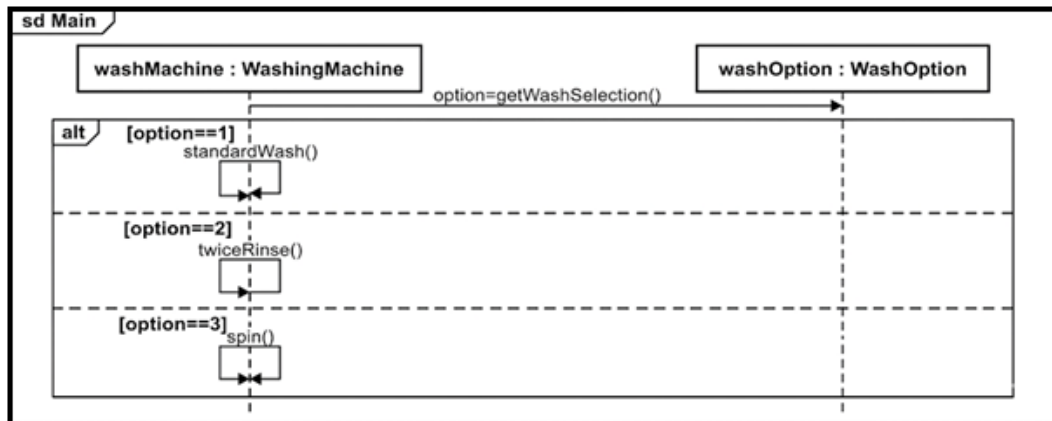
Exercise 2: Solutions → p. 61

Provide the JAVA code corresponding to the following UML class diagram.

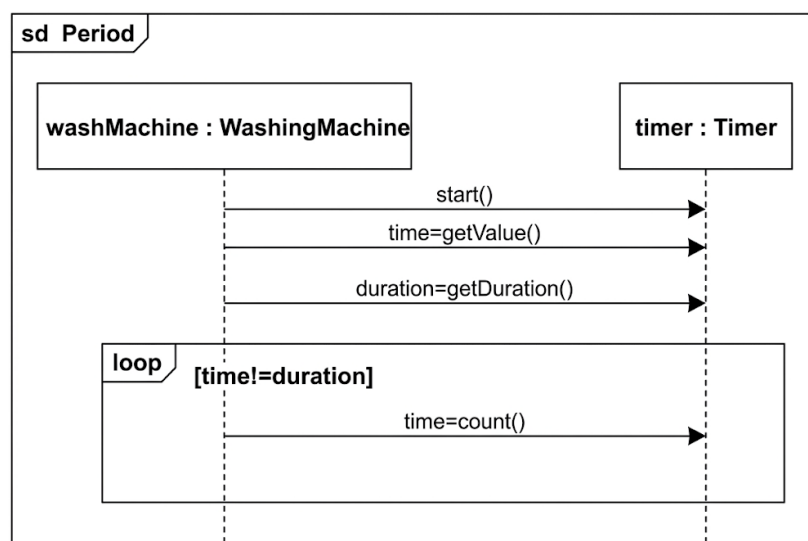
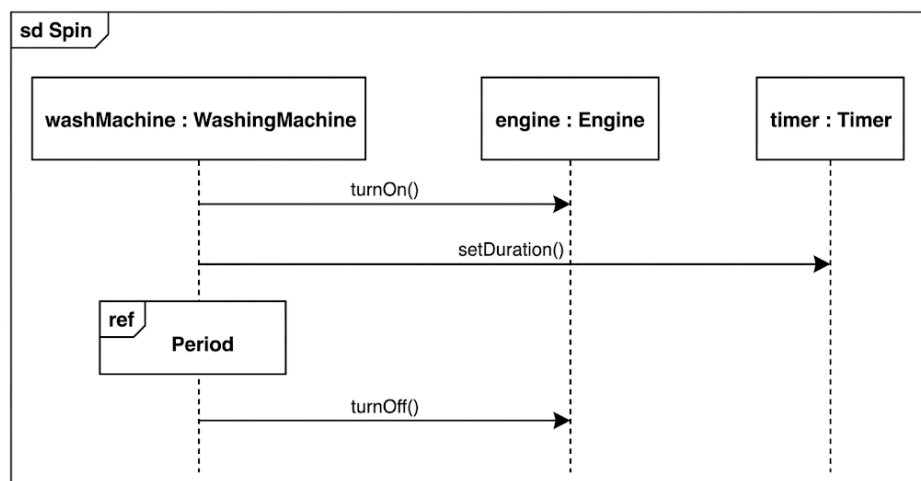


Exercise 3: Solutions → p. 61

Given the sequence diagram corresponding to the main function of the WashingMachine class from Exercise N 2.



Similarly, for the spin method of the WashingMachine class.



Tutorials Solution

Tutorial 1 : Introduction to OOP concepts

Exercise 1:

1. Apple
2. IBM PC
3. Programs (or Software)
4. Input unit, Output unit, Memory unit, Arithmetic and logic unit (ALU), Central processing unit (CPU), and Secondary storage unit.
5. Machine languages, Assembly languages, and High-level languages.
6. Compiler

Exercise 2:

1. JAVA
2. JAVAc
3. .JAVA
4. .class
5. Bytecode

Exercise 3:

- **Object & Class:** The **Class** is the technical drawing or blueprint for a specific model (e.g., Casio F-91W). An **Object** is the actual physical watch you bought and are wearing.

- **Attributes & Behaviors:** **Attributes** are state data like `currentTime`, `batteryLevel`, or `strapColor`. **Behaviors** are functions the watch performs, such as `updateDisplay()`, `startChime()`, or `illuminate()`.
- **Inheritance:** An **Alarm Clock** inherits the time-keeping attributes of a **Watch** but extends the class by adding a new attribute (`alarmTime`) and a new behavior (`soundAlarm()`).
- **Abstraction:** The watch face provides a simple view of the time. The user does not need to know if the watch uses a quartz crystal or a mechanical spring to track seconds.
- **Modeling:** Before building the watch, engineers use diagrams to ensure the interaction between the hands, gears, and battery is logical and efficient.
- **Encapsulation & Information Protection:** The watch case (the "wrapper") protects the delicate internal components. The user cannot touch the gears directly; this prevents the internal state from being corrupted or broken.
- **Messages & Interface:** The **Interface** consists of the buttons and the crown. When you press a button, you are sending a **Message** to the watch object to execute a specific method, like `toggleBacklight()`.

Exercise 4:

1. Attributes (or properties/fields)
2. Information Hiding (or encapsulation)
3. Message
4. Class
5. Interface
6. Inheritance
7. Protected

8. Relationship Analysis:

- **IS-A:** Represents inheritance. Maruti is-a Car, meaning it inherits all characteristics of the Car class.
- **HAS-A:** Represents composition. Maruti **has-an** Engine, meaning the Engine is a separate object used as a component within the car.

9. Abstract

10. Concrete

11. Implementation (or override)

12. Abstract (or virtual/unimplemented)

Tutorial 2: Introduction to JAVA**Exercise 1:**

```
public class Identity {
    private String name;

    public Identity(String name) {
        this.name = name;
    }

    public void display() {
        System.out.println("Name: " + this.name);
    }

    public static void main(String[] args) {
        Identity myProfile = new Identity("Your Name Here");
        myProfile.display();
    }
}
```

Exercise 2:

```
public class OutputComparison {
    public static void main(String[] args) {
        int a = 1, b = 2, c = 3, d = 4;

        // Using print
        System.out.print(a + " " + b + " " + c + " " + d + "
");

        // Using println (adds a newline at the end)
        System.out.println("");
        System.out.println(a + " " + b + " " + c + " " + d);

        // Using printf (formatted)
        System.out.printf("%d %d %d %d\n", a, b, c, d);
    }
}
```

Exercise 3:

```
import JAVA.util.Scanner;

public class ProductCalculator {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter three integers: ");
        int n1 = input.nextInt();
        int n2 = input.nextInt();
    }
}
```

```
        int n3 = input.nextInt();

        int result = n1 * n2 * n3;
        System.out.println("The product is: " + result);
    }
}
```

Exercise 4:

```
import JAVA.applet.Applet;
import JAVA.awt.Graphics;

public class NameApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("Your Name Here", 50, 25);
    }
}
```

Exercise 5:

```
import JAVA.util.Scanner;

public class Comparison {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter two real numbers: ");
        double num1 = input.nextDouble();
        double num2 = input.nextDouble();

        if (num1 == num2) {
            System.out.println("EQUALITY");
        } else {
            System.out.println("Greater number: " +
Math.max(num1, num2));
        }
    }
}
```

Exercise 6:

```
import JAVA.util.Scanner;

public class Analysis {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter three integers: ");
        int x = input.nextInt();
        int y = input.nextInt();
        int z = input.nextInt();
    }
}
```

```
int sum = x + y + z;
int product = x * y * z;
double average = sum / 3.0;
int max = Math.max(x, Math.max(y, z));
int min = Math.min(x, Math.min(y, z));

System.out.printf("Sum: %d\nAverage: %.2f\nProduct:
%d\nSmallest: %d\nLargest: %d\n",
sum, average, product, min, max);
}
}
```

Exercise 7:

```
public class ScaleTable {
    public static void main(String[] args) {
        System.out.println("\n\t10*N\t100*N\t1000*N");
        for (int n = 1; n <= 5; n++) {
            System.out.printf("%d\t%d\t%d\t%d\n", n, n * 10, n
* 100, n * 1000);
        }
    }
}
```

Exercise 8:

```
import JAVA.util.Scanner;

public class MaxSearch {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        int max = Integer.MIN_VALUE;

        for (int i = 1; i <= 10; i++) {
            System.out.print("Enter integer " + i + ": ");
            int current = input.nextInt();
            if (current > max) {
                max = current;
            }
        }
        System.out.println("The largest number is: " + max);
    }
}
```

Tutorial 3: Variable scope in JAVA**Exercise 1:**

1. 1. x = 11
2. 2. y = 22
3. 3. x = 878 (Inside calcul1, the local copy is changed)
4. 4. y = 23 (Inside calcul1, the local copy is incremented)
5. 5. x = 11 (Back in main, the original x remains unchanged)
6. 6. y = 22 (Back in main, the original y remains unchanged)
7. 7. x = 33 (Inside calcul2, local variable x)
8. 8. y = 44 (Inside calcul2, local variable y)
9. 9. a = 66 (Parameter a was 11, now 2×33)
- 10.10. b = 88 (Parameter b was 22, now 2×44)
- 11.11. x = 11 (Original x unchanged)
- 12.12. y = 22 (Original y unchanged)
- 13.13. object[0] = 5
- 14.14. object[0] = 37 (Inside calcul3, the actual array content is modified)
- 15.15. object[0] = 37 (The change persists because the reference was used)

Exercise 2:

- Test.x: 22 (The static x was changed via this.x)
- t.x: 22 (The new object t shares the same static x)
- t.y: 33 (The new object t has its own fresh copy of y)
- y: 44 (The y of the object that called method1 was updated)

Exercise 3:**Class Test**

- **Result: Compilation Error.**
- **Reason:** The variable x is declared inside the for loop block. It does not exist outside that loop. The final `System.out.println(x);` will fail.

Class Test2

- **Result: Compilation Error.**
- **Reason:** JAVA does not allow you to declare a variable (int a in the loop) that has the same name as another variable in the same scope (int a = 5). This is known as a variable name clash.

Exercise 4:

- name : Alice
- rollNo : 1
- clgName : XYZ
- name : Bob
- rollNo : 2
- clgName : XYZ

Exercise 5:

- 20
- **Note:** This works because a final variable can be initialized **once**. Since i was declared but not assigned, the line `i = 20;` is its first (and only) valid assignment.

Exercise 6:**Result: Compilation Error.**

- **Reason:** In class Base, the method `show()` is marked as final. When class Derived attempts to override it (`public void show()`), the compiler will throw an error because final methods cannot be overridden.

Exercise N7:

The final keyword is a non-access modifier used to restrict the user in three specific contexts:

1. Final Variables

- **Primitive Variables:** Once a final variable is assigned a value, it becomes a **constant**. Any attempt to reassign it will result in a compile-time error.
- **Reference Variables:** If an object reference is final, the reference cannot be changed to point to a different object. However, the **internal state** (the data inside the object) can still be modified.

```
class VariableExample {
    public static void main(String[] args) {
        // Primitive: Value is locked
        final int MAX_SPEED = 120;
        // MAX_SPEED = 150;
        // ERROR: Cannot assign a value to final variable
        // Reference: Memory address is locked
        final int[] AGES = {10, 20};
        AGES[0] = 15; // WORKS: Internal state can change
        // AGES = new int[]{30, 40};
        // ERROR: Cannot point to a new array
    }
}
```

2. Final Methods

- A method declared as final **cannot be overridden** by subclasses.
- **Purpose:** This is used to prevent a subclass from changing the core logic of a method that is critical to the class's integrity or security.

```
class Parent {
    public final void securityCheck() {
        System.out.println("Standard security protocol.");
    }
}

class Child extends Parent {
    // ERROR: Cannot override the final method from Parent
    /* public void securityCheck() {
        System.out.println("Modified protocol.");
    } */
}
```

3. Final Classes

- A class declared as final **cannot be extended** (inherited).
- **Purpose:** It stops inheritance entirely. For example, the String class in JAVA is final so that no one can create a subclass and change how strings behave.

```
final class SecureSystem {  
    void login() {  
        System.out.println("Logging in...");  
    }  
}  
  
// ERROR: The Hacker cannot inherit the class SecureSystem  
/*  
class Hacker extends SecureSystem { }  
*/
```

Tutorial 4: Strings and arrays in JAVA**Exercise 1:**

```
import JAVA.util.Arrays;
class Exercisel {
    public static void main(String[] args){
        int[] my_array1 = {
            1789, 2035, 1899, 1456, 2013,
            1458, 2458, 1254, 1472, 2365,
            1456, 2165, 1457, 2456};
        String[] my_array2 = {
            "JAVA",
            "Python",
            "PHP",
            "C#",
            "C Programming",
            "C++"
        };
        System.out.println("Original numeric array:"
            +Arrays.toString(my_array1));
        Arrays.sort(my_array1);
        System.out.println("Sorted numeric array:"
            +Arrays.toString(my_array1));
        System.out.println("Original string array :"
            +Arrays.toString(my_array2));
        Arrays.sort(my_array2);
        System.out.println("Sorted string array : "
            +Arrays.toString(my_array2));
    }
}
```

▪ Program output:

Original numeric array : [1789, 2035, 1899, 1456, 2013, 1458, 2458, 1254, 1472, 2365, 1456, 2165, 1457, 2456]

Sorted numeric array : [1254, 1456, 1456, 1457, 1458, 1472, 1789, 1899, 2013, 2035, 2165, 2365, 2456, 2458]

Original string array : [JAVA, Python, PHP, C#, C Programming, C++]

Sorted string array : [C Programming, C#, C++, JAVA, PHP, Python]

Exercise 2:

```
class Exercise2 {
    public static void main(String[] args) {
        int[] T1 = {1789, 2035, 1899, 1456, 2013, 1458,
                    2458, 1254, 1472, 2365, 1456, 2165, 1457, 2456};
        String[] T2 = {"JAVA", "Python", "PHP", "C#", "C
        Programming", "C++"};

        // Manual Sort for Integers (Bubble Sort)
        for (int i = 0; i < T1.length - 1; i++) {
            for (int j = 0; j < T1.length - i - 1; j++) {
                if (T1[j] > T1[j + 1]) {
                    int temp = T1[j];
                    T1[j] = T1[j + 1];
                    T1[j + 1] = temp;
                }
            }
        }

        // Manual Sort for Strings
        for (int i = 0; i < T2.length - 1; i++) {
            for (int j = 0; j < T2.length - i - 1; j++) {
                if (T2[j].compareTo(T2[j + 1]) > 0) {
                    String temp = T2[j];
                    T2[j] = T2[j + 1];
                    T2[j + 1] = temp;
                }
            }
        }

        // Print results
        for (int n : T1) System.out.print(n + " ");
        System.out.println();
        for (String s : T2) System.out.print(s + " | ");
    }
}
```

Exercise 3:

```
class Exercise3 {
    public static void main(String[] args) {
        int my_array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        int sum = 0;
        for (int i : my_array)
            sum += i;
        System.out.println("The sum is " + sum);
    }
}
```

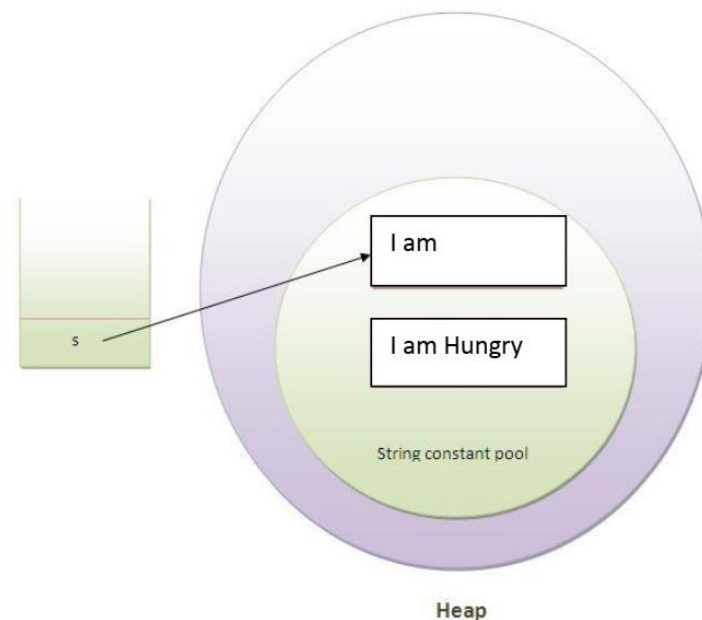
Exercise 4:

```
import JAVA.util.Scanner
class Exercise4 {
public static void main(String[] args)
{
Scanner sc= new Scanner(System.in);
String str = "JAVA Exercises!";
System.out.println("Original String = " + str);
int x=sc.nextInt();
char res = str.charAt(x);
// Print out the results.
System.out.println("The character at position " + x + " is "
+res);
}
}
```

Exercise 5:**Output:**

I am

Strings in JAVA are **immutable**. When you call `s.concat(" Hungry")`, JAVA creates a *new* String object in the String Pool, but it does not change the original variable `s`. To change `s`, you would need to write: `s = s.concat(" Hungry");`.



Exercise 6:

Output:

Original string: The quick brown fox jumps over the lazy dog.

New String: The quick brown fox jumps over the lazy fog.

New String: The quick brown cat jumps over the lazy dog.

The quick brown fox jumps overthe lazy dog.

The quick brown fox jumps over the lazy fog start with the is : TRUE

Tutorial 5: Introduction to Object Oriented Programming - JAVA**Exercise 1:****Output:**

The circle has radius of 1.0 and area of 3.141592653589793

The circle has radius of 2.0 and area of 12.566370614359172

Exercise 2:

```
class OPERATION {
    char op;
    double op1;
    double op2;
    double res;

    // Constructor with parameters
    OPERATION(char op, double op1, double op2) {
        this.op = op;
        this.op1 = op1;
        this.op2 = op2;
    }

    // Default constructor
    OPERATION() {
        op = ' ';
        op1 = 0;
        op2 = 0;
    }

    // Method to perform calculation
    void calcul() {
        switch (op) {
            case '+':
                res = op1 + op2;
                break;
            case '-':
                res = op1 - op2;
                break;
            case '*':
                res = op1 * op2;
                break;
            case '/':
                res = op1 / op2;
                break;
            default:
                System.out.println("Operation non permise");
        }
    }
}
```

```
    }

    // Getter for result
    double get_result() {
        return res;
    }

    // Setters
    void set_op1(double a) {
        op1 = a;
    }

    void set_op2(double b) {
        op2 = b;
    }

    void set_op(char c) {
        op = c;
    }
}

class Test {
    public static void main(String[] args) {
        OPERATION A = new OPERATION();
        System.out.println(A.op + " " + A.op1 + " " +
            A.op2);

        A.set_op('+');
        A.set_op1(10);
        A.set_op2(-2);
        System.out.println(A.op + " " + A.op1 + " " +
            A.op2);

        A.calcul();
        double t = A.get_result();
        System.out.println(t);

        OPERATION A1 = new OPERATION('+', 10, -2);
        A1.calcul();
        t = A1.get_result();
        System.out.println(t);
    }
}
```

Exercise 3:

```
class man implements MyInterface {
    public void sayHello() {
        System.out.println("Hello");
    }
}

class cow implements MyInterface {
    public void sayHello() {
        System.out.println("mouuuuuuuuuuh");
    }
}
```

Exercise 4:

```
public interface MyOtherInterface {
    public String hello = "Hello";
    public void sayGoodbye();
}
```

Exercise 5:

man from Exercise N5 implements the MyInterface interface, whereas **man2** incorrectly tries to extend it as a class, which is not allowed in JAVA.

Exercise 6:

```
// Interface definition
interface Command {
    public void turnOn();
    public void turnOff();
}

// Class for TV remote control
class TVRemote implements Command {
    private String style;

    public TVRemote(String style) {
        this.style = style;
    }

    public void turnOn() {
        System.out.println("The TV is ON");
    }

    public void turnOff() {
        System.out.println("The TV is OFF");
    }

    public String toString() {
```

```
        return "TV Remote style: " + style;
    }
}

// Class for Projector (Datashow) remote control
class ProjectorRemote implements Command {
    private String style;

    public ProjectorRemote(String style) {
        this.style = style;
    }

    public void turnOn() {
        System.out.println("The Projector is ON");
    }

    public void turnOff() {
        System.out.println("The Projector is OFF");
    }

    public String toString() {
        return "Projector remote style: " + style;
    }
}

// Test class
class Test {
    public static void main(String[] args) {
        TVRemote tv = new TVRemote("Samsung 2016");
        ProjectorRemote projector = new
            ProjectorRemote("Acer 2017");

        tv.turnOn();
        projector.turnOn();

        tv.turnOff();
        projector.turnOff();

        System.out.println(tv);
        System.out.println(projector);
    }
}
```

Tutorial 6 : Inheritance and Abstract Classes in JAVA**Exercise 1:****Output:****1-**

vroom

Truck1

Car2

2-

vroomvroom

Truck1

Car1

Exercise 2:

```
class MonsterTruck extends Truck {  
  
    public void m1() {  
        System.out.println("monster 1");  
    }  
  
    public void m2() {  
        super.m1(); // Calls m1 from Truck  
        super.m2(); // Calls m2 from Truck  
    }  
  
    public String toString() {  
        return "monster " + super.toString();  
    }  
}
```

Exercise 3:**1-**

```
class Triangle extends TwoDShape {  
  
    private String style;
```

```
// Constructor
public Triangle(String style, double width, double height)
{
    super(width, height); // Calls TwoDShape constructor
    this.style = style;
}
// Method to calculate area
public double area() {
    return getWidth() * getHeight() / 2.0;
}
// Method to print the style of the triangle
public void printStyle() {
    System.out.println("The triangle is " + style);
}
// Getter for style
public String getStyle() {
    return style;
}
}
```

2-

```
abstract class TwoDShape {

    private double width;
    private double height;

    // Constructor
    public TwoDShape(double width, double height) {
        this.width = width;
        this.height = height;
    }

    // Getters
    public double getWidth() {
        return width;
    }

    public double getHeight() {
        return height;
    }

    // Override toString to describe the shape
    public String toString() {
        return "The shape Width is " + width + " Height is "
            + height;
    }

    // Abstract method to be implemented by subclasses
    public abstract double area();
}
```

```
}

class Triangle extends TwoDShape {

    private String style;
    // Constructor
    public Triangle(String style, double width, double
                    height) {
        super(width, height); // Calls TwoDShape constructor
        this.style = style;
    }
    // Override the abstract area method
    public double area() {
        return getWidth() * getHeight() / 2.0;
    }
    // Getter for style
    public String getStyle() {
        return style;
    }

    // Override toString to include triangle style
    public String toString() {
        return super.toString() + "\n" + "The triangle is "
            + style;
    }
}
```

Tutorial 7 : Introduction to UML

1- UML is the result of a combination of three modeling methods, which are: **OMT (Object Modeling Technique)**, **Booch**, and **OOSE (Object-Oriented Software Engineering)**.

2- UML is a modeling language that allows for **Visualization**, **specification**, **construction**, and the **documentation** of elements of a software system.

3- Read and explain the following UML models:

- Pets and PetOwners

One or more Pets are associated with exactly one PetOwner.

Each Pet has exactly one PetOwner.

- CPU and Controllers

One CPU is associated with zero or more Controllers.

One to four DiskDrives are associated with one SCSIController.

SCSIController is a specialized type of Controller.

- Books and Pages

One or more Books are associated with one or more Pages.

Patron and Shelf depend on (use) Books.

- Employees and Company

Each Employee works for one Company (which can have zero employees).

Each AdministrativeAssistant has one or more Supervisors (who can have zero or more employees).

Each Company has exactly one BoardOfDirectors (and vice versa).

- Offices and Employees

Each Office is allocated to zero or more Employees.

An Employee can have no office or at most one office.

- Persons and BoardOfDirectors

A Person can be a board member of zero or more BoardOfDirectors.

Each BoardOfDirectors has from 3 to 8 Persons.

4- The **Use case diagram** allows capturing the behavior of a system in terms of requirements.

5- Discuss the following diagrams:

These diagrams represent different ways to visualize how a system works. In **UML (Unified Modeling Language)**, we use different types of diagrams to focus on different "views" of a system—some show the big picture, while others show step-by-step logic.

- **Use Case Diagram (The Big Picture)**

This diagram helps you understand **what** the system does and **who** uses it, without getting into the technical details.

- **Photographer (Actor):** The person interacting with the system.
- **System Boundary (The Box):** Represents the camera itself.
- **Correct Way (Abstraction):** You only show high-level goals like "Take Picture" or "Change Film."
- **Incorrect Way (Implementation):** We shouldn't list every tiny physical step like "Open Shutter." Use Cases are about *goals*, not instructions.

- **Sequence Diagram (The Conversation)**

Think of this as a script for a play. It shows how different parts of a system "talk" to each other over time.

- **Lifelines (Vertical Lines):** These represent the objects (e.g., Passenger and Passenger Services).
- **Arrows (Messages):** These show information being sent back and forth.
- **Order:** The messages go from top to bottom, showing exactly what happens first, second, and third during the check-in process.

- **Activity Diagram (The Workflow)**

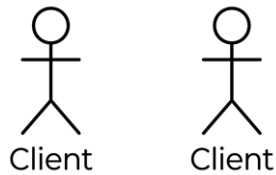
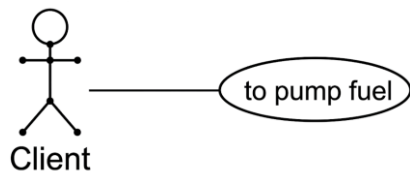
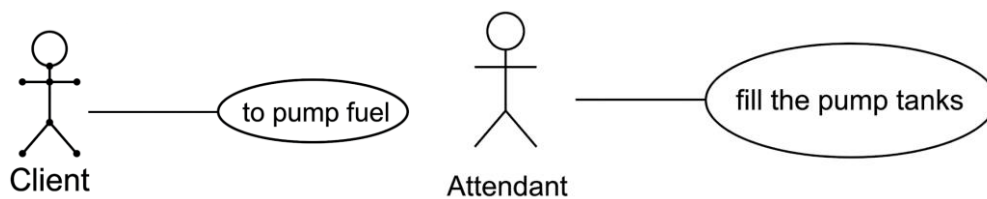
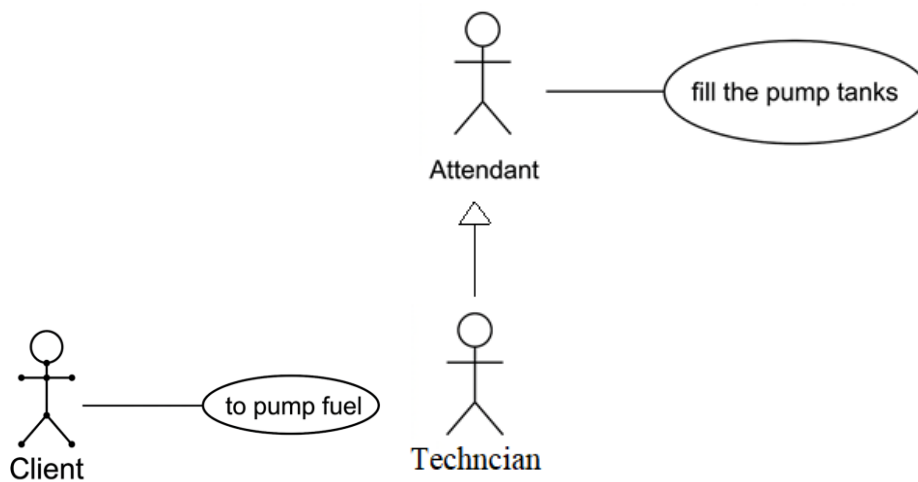
This is like an advanced flowchart. It shows the flow of actions and decisions.

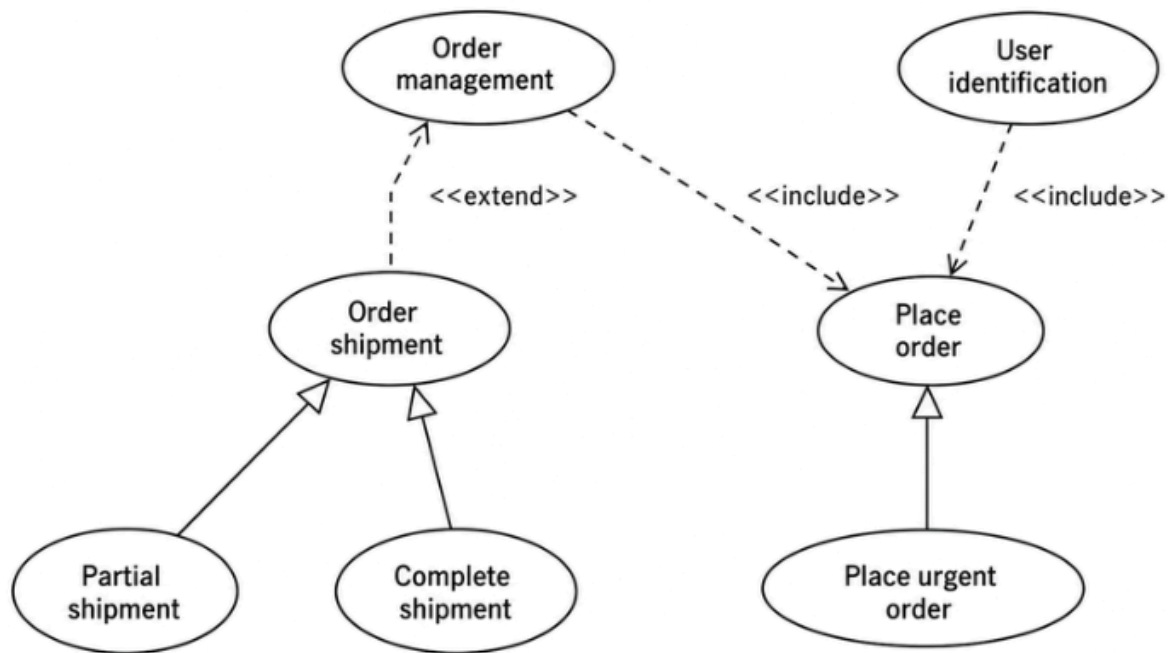
- **Decision Diamond:** The diamond shape asks a question (e.g., "Is the Check-In OK?"). If yes, the flow continues; if no, it stops.
- **Parallel Actions (Thick Horizontal Lines):** These are called "forks." They show things happening at the same time, like a passenger boarding while their luggage is being loaded.

▪ **State Machine Diagram (The Status)**

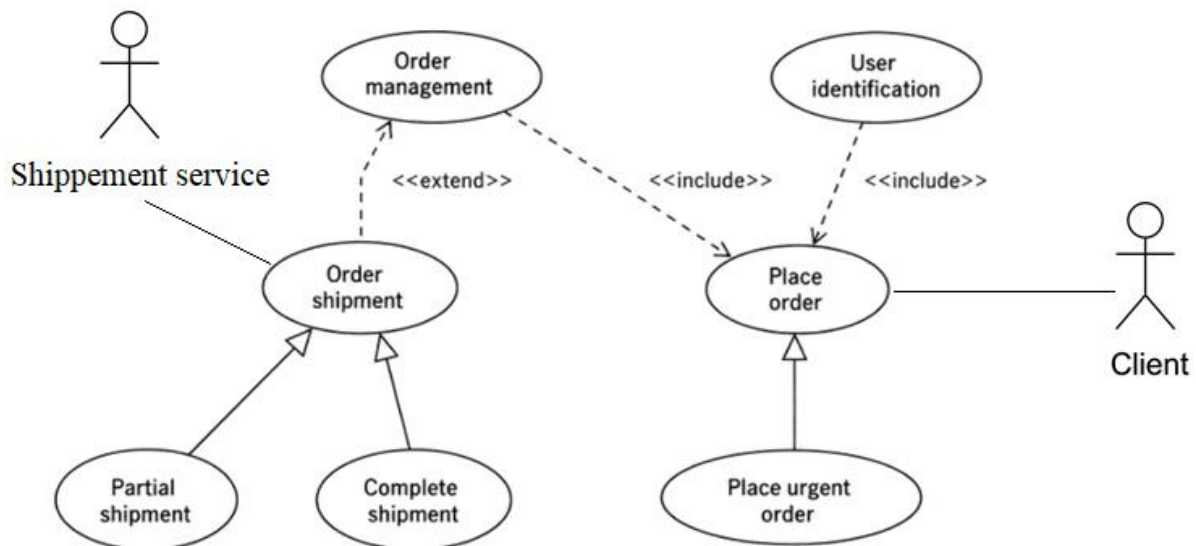
This focuses on a single object and how its "status" changes based on what happens.

- **States (Rounded Boxes):** These are the conditions an object can be in (e.g., an ATM being **Idle**, **Active**, or **Out of Service**).
- **Transitions (Arrows):** These show how you move from one state to another. For example, "inserting a card" moves the ATM from **Idle** to **Active**.

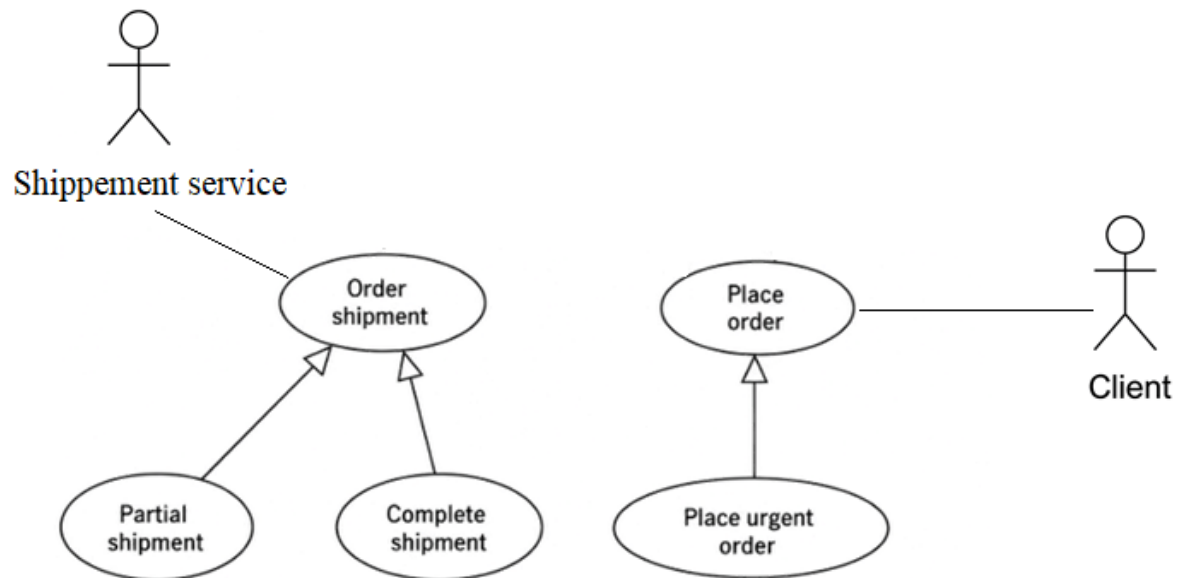
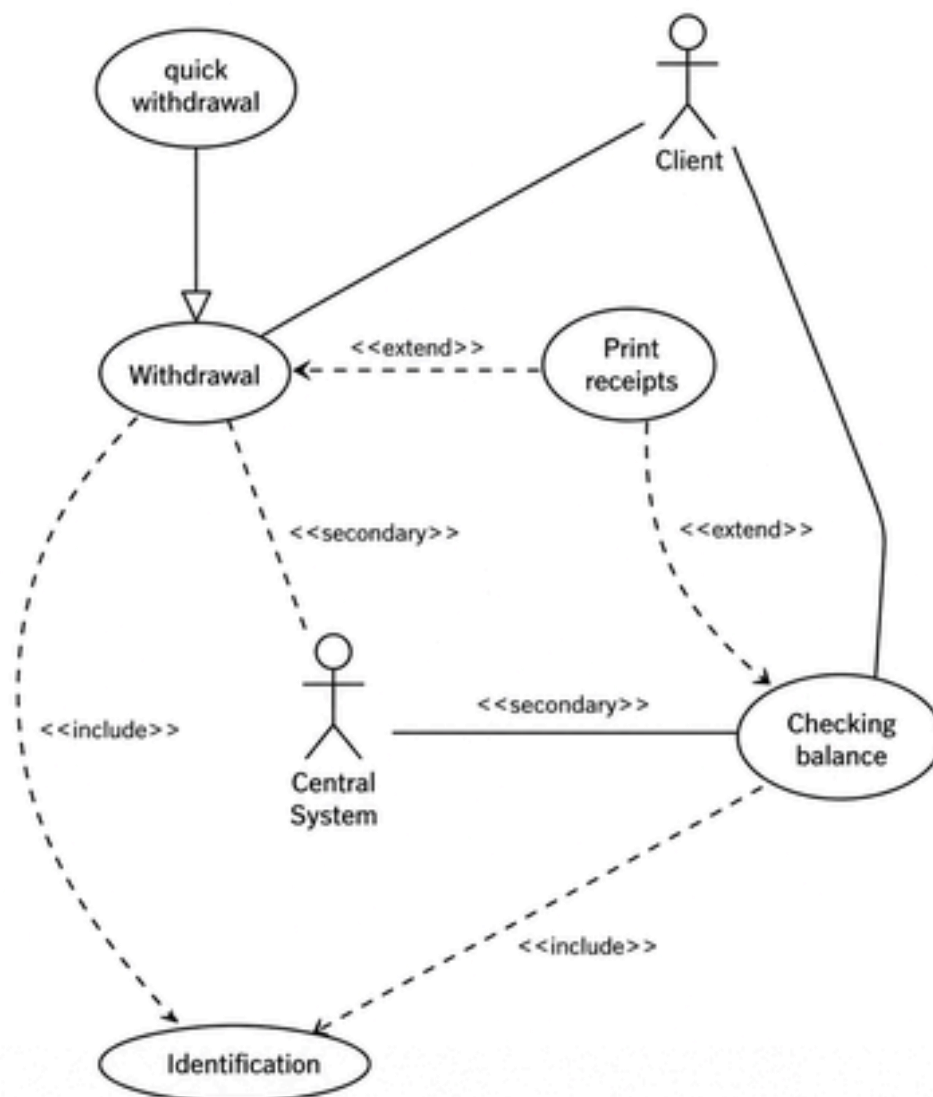
Tutorial 8: Behavioral Modeling: Use Case Diagrams**Exercise 1:****1- The actor is the client****2- Diagram****3- Diagram****4- Diagram**

Exercise 2:**1- Diagram**

The question is not being asked, but let us note that introducing actors would lead to the following diagram:

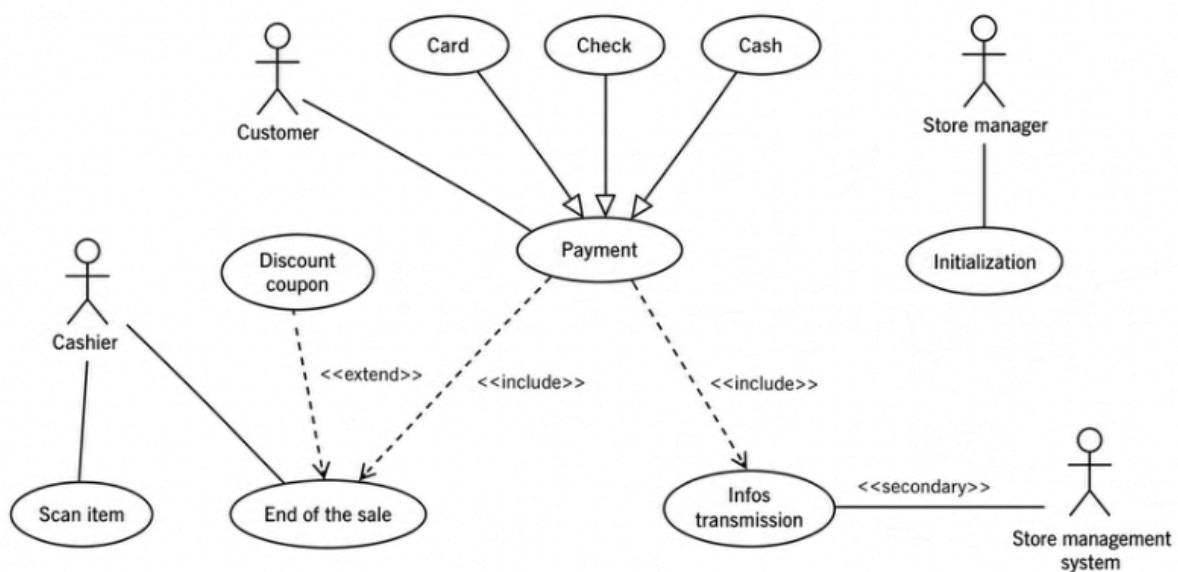


The introduction of the actors reveals that the 'Order Management' use case is not associated with any actor to trigger it. In reality, this type of overarching use case is often a design flaw. Here, the following solution is preferred:

**Exercise 3:**

An actor is considered primary for a use case when the use case provides a service to that actor. The other actors are then considered secondary. A use case has at most one primary actor. A primary actor obtains an observable result from the system, while a secondary actor is involved to provide additional information. In general, the primary actor initiates the use case through its interactions. The <<primary>> stereotype is used to label the association linking a use case to its primary actor, while the <<secondary>> stereotype is used for secondary actors.

Exercise 4:



Use cases also do **not** model sequences of actions. The **order** of actions is not represented in use case diagrams (but there will be other complementary diagrams to represent that).

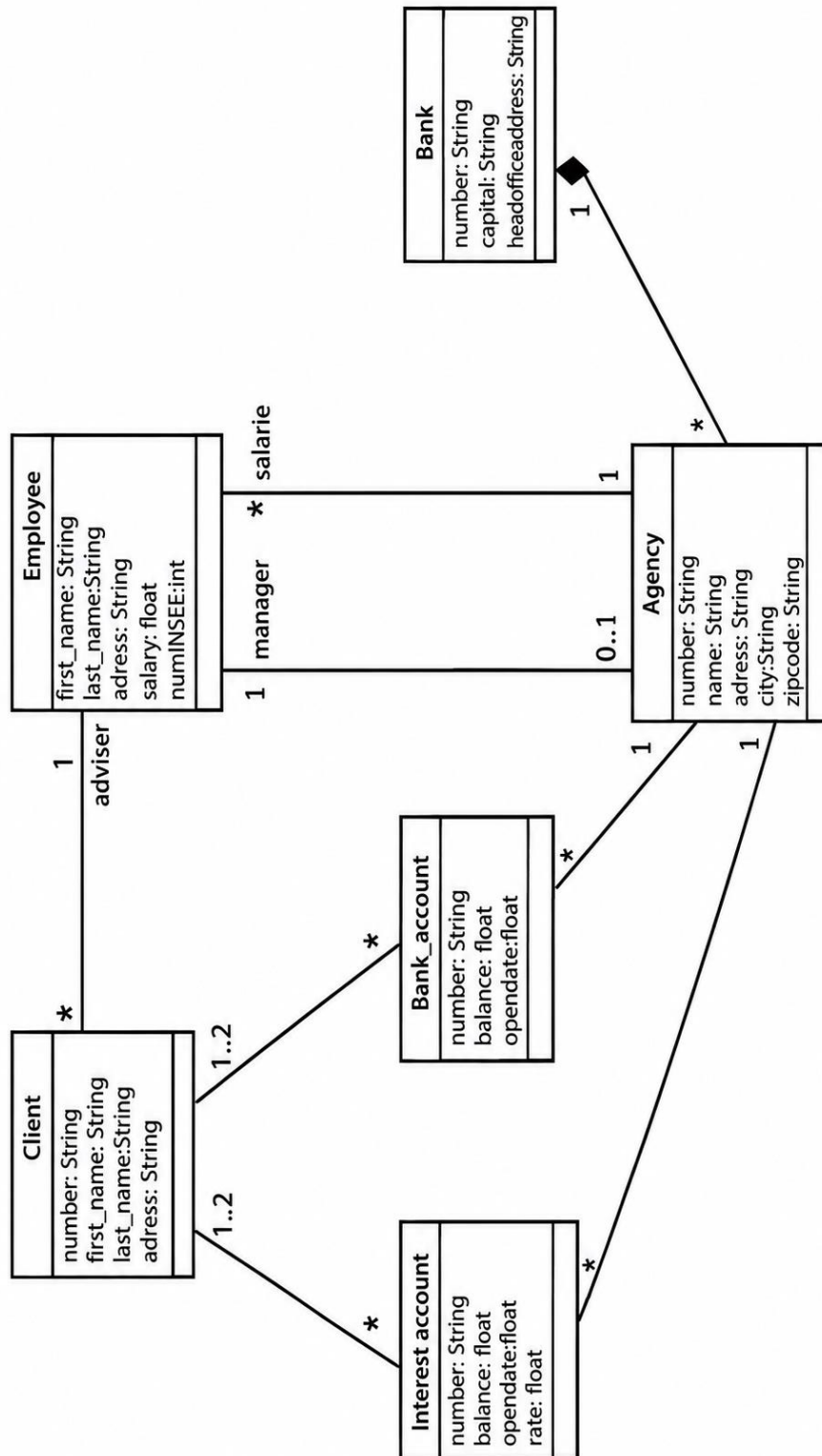
Thus, the fact that there is a loop over the items and that we exit it by signaling the end is **not** explicitly represented — and especially not through include or extend relationships. These relationships express **inclusions**, not **temporal** relationships. In this case, the item scanning is simply modeled by a use case that can be repeated, and another one that will be triggered when the time comes.

Furthermore, a use case diagram should **not** represent every single action. Here, for example, we define a use case *Scan Item* that includes entering the item number, the quantity, and displaying the label and the price. If this is not clear from the use case label alone, a textual description should be added.

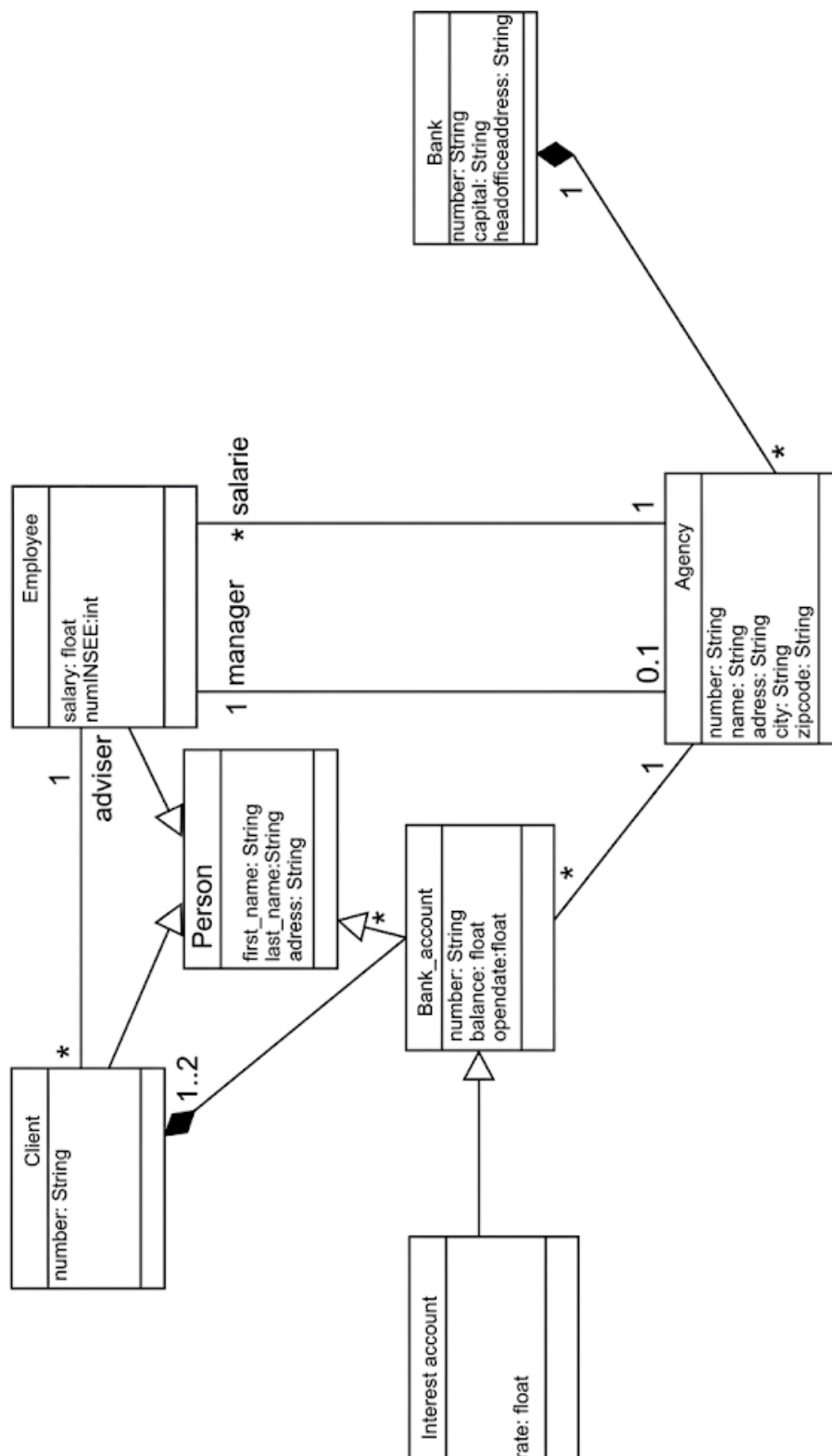
Tutorial 9: Structural Modeling: Class Diagrams

Exercise 1:

1-

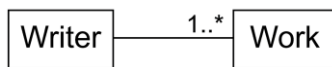


2-

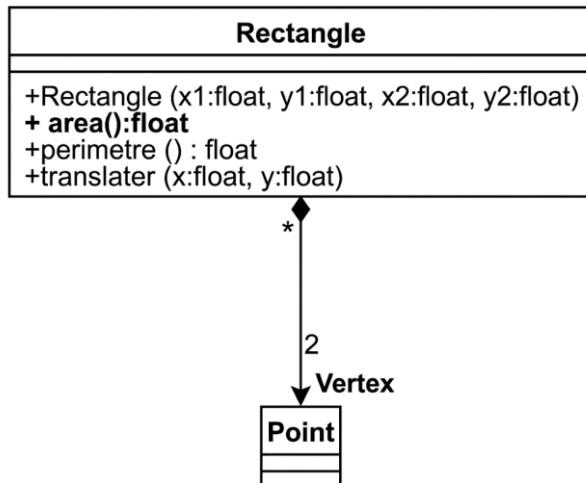


Exercise 2:

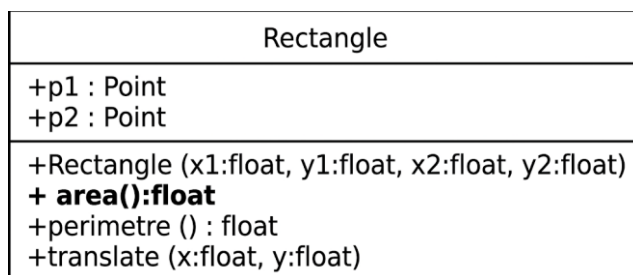
1-



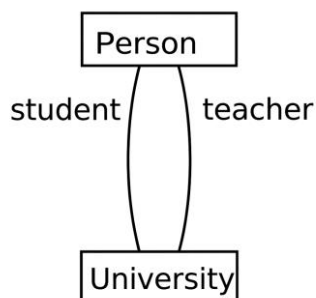
2-



The following solution is equivalent (because the associations are unidirectional compositions):

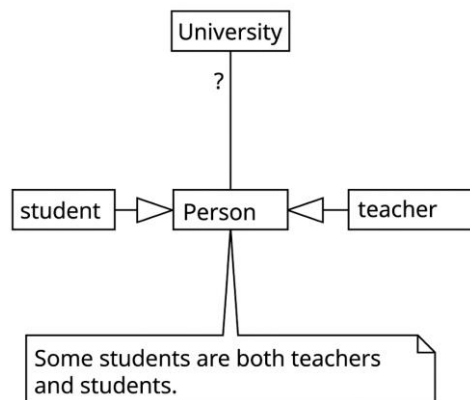


3-

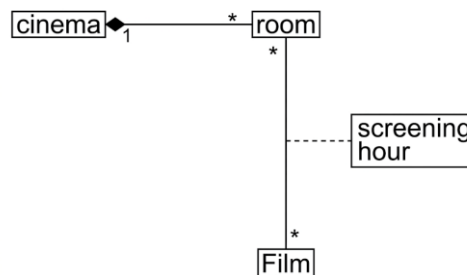


The following lesson addresses the concept of inheritance in UML. Note that using inheritance to solve this problem might have been tempting, but it is neither necessary nor desirable. Here, it is the fact of being affiliated with a university, through one association or another, that determines a person's role as a teacher or student. Generally

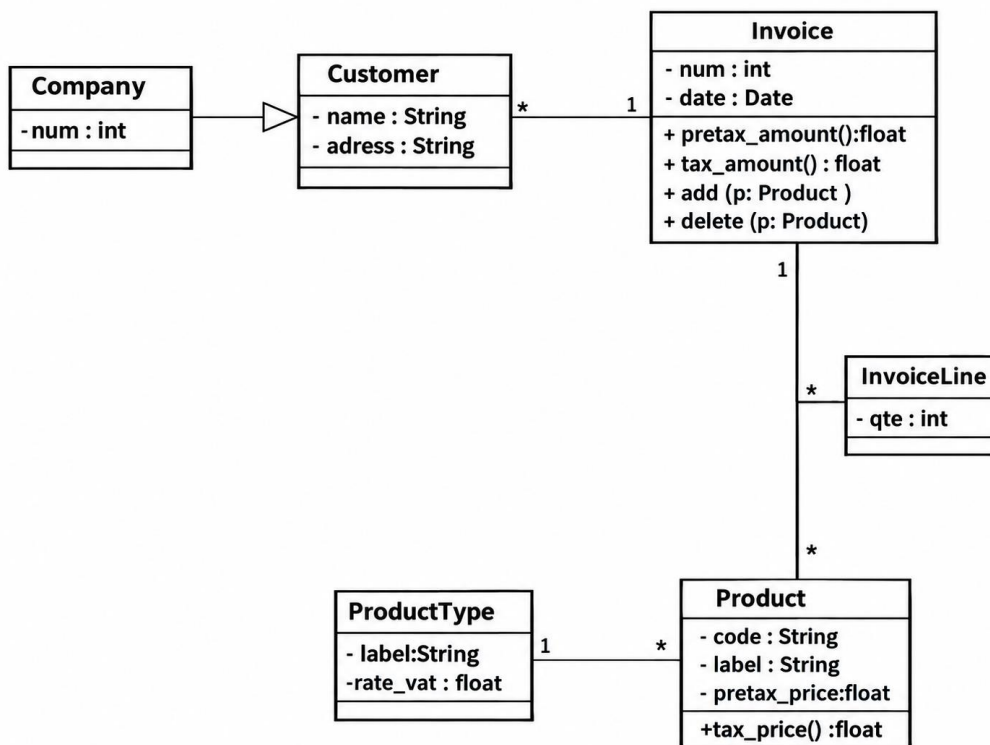
speaking, inheritance should only be used sparingly, and when truly necessary. Typically, the following solution is less than ideal:



4-



Exercise 3:



Exercise 4:

PointPlan
-abscisse:float -ordonnee:float
+PointPlan () +PointPlan (x:float,y:float) +PointPlan (p:PointPlan) -getOrdonnee () : float -getAbscisse () : float +egaleA (p:PointPlan) : boolean +init () +translate (dx:float, dy:float) +distance () : double +toString () : String +affiche ()

Tutorial 10: Behavioral Modeling: Sequence Diagrams**Exercise 1:****First diagram**

```
class Person {}
class University {
    Person student;
    Person teacher;
}
```

Second diagram

```
class Person {
    University university;
}
class Student extends Person { }
class Teacher extends Person { }
class University { }
```

Third diagram

```
class Cinema {
    // 1 to * relationship
    Room[] rooms;
}

class Room {
    // Connected to Film via ScreeningHour
}

class Film {
    // Connected to Room via ScreeningHour
}

class ScreeningHour {
    // Attributes for the Association Class
    Room[] rooms;
    Film[] films;
    int hour;
}
```

Exercise 2 and 3:

```
interface Machine { // all behaviors are public and abstract
public void  turnOn(); // abstract
public void  turnOff(); // abstract
} // -----

class Engine implements Machine { // -----
//int  x;   Integer x;
private int rotation;
public Engine(int rotation) {
this.rotation = rotation;
}
public void turnOn() {
System.out.println("the engine is ON");
}
public void turnOff() {
System.out.println("the engine is OFF");
}
} // -----

abstract class Sensor { // -----
public abstract boolean check();
} // -----

class WaterSensor extends Sensor { // -----
public int currentLevel;
public int desiredLevel;
public WaterSensor (int currentLevel, int desiredLevel) {
this.currentLevel = currentLevel;
this.desiredLevel = desiredLevel;
}
public boolean check() {
return this.currentLevel >= this.desiredLevel?true:false;
}
} // -----

class TempSensor extends Sensor { // -----
public int currentTemp;
public int desiredTemp;
public TempSensor (int currentTemp, int desiredTemp) {
this.currentTemp = currentTemp;
this.desiredTemp = desiredTemp;
}
public boolean check() {
return this.currentTemp >= this.desiredTemp?true:false;
}
} // -----

class DoorSensor extends Sensor { // -----
public boolean doorState; // true means close false means open
```

```
public DoorSensor (boolean doorState) {
    this.doorState = doorState;
}
public boolean check() {
    return this.doorState;
}
} // -----

class Timer { // -----
    private int Value;
    private int Duration;
    public Timer (int Value, int Duration) {
        this.Value = Value;
        this.Duration = Duration;
    }
    public void setDuration(int in) { this.Duration = in; } //
    setter
    public void start() { this.Value = 0; }
    public int  count() { return ++this.Value; }
    public int  getValue() { return this.Value; } // getter
    public int  getDuration() { return this.Duration; } // getter
} // -----

class WashOption { // -----
    private  int washSelection;
    public WashOption(int washSelection) {
        this.washSelection = washSelection ;
    }
    public  int  getwashSelection() { return this.washSelection; }
} // -----

class WashingMachine implements Machine { // -----
    private int washTime;
    private int rinseTime;
    private int spinTime;
    private Timer timer;
    private WashOption washOption;
    private Sensor waterSensor;
    private Engine engine;
    public WashingMachine(int washTime,int rinseTime,int spinTime,
        Timer timer,WashOption washOption,Sensor waterSensor,
        Engine engine) {
        this.washTime = washTime;
        this.rinseTime = rinseTime;
        this.spinTime = spinTime;
        this.timer = timer;
        this.washOption = washOption;
        this.waterSensor = waterSensor;
        this.engine = engine;
    }
    public void turnOn() {
```

```
System.out.println("the WashingMachine is ON");
}
public void turnOff() {
System.out.println("the WashingMachine is OFF");
}
private void wash() { System.out.println("the WM is
washing");}
private void rinse() { System.out.println("the WM is
rinsing");}
private void fill() { System.out.println("the WM is filling
");}
private void empty() { System.out.println("the WM is
emptying");}
private void standardWash() {
System.out.println("the WM is standard washing");}
private void twiceRinse() {
System.out.println("the WM is twice Rinsing");}

public static void main(String[]args) {
Timer t = new Timer(0,2400);
WashOption wo = new WashOption(3);
Sensor ws = new WaterSensor(0,4);
Engine e = new Engine(1700);
WashingMachine wm = new
WashingMachine(2400,600,2000,t,wo,ws,e);
int option = wo.getwashSelection();
if (option == 1 ) { wm.standardWash(); }
else {
    if (option == 2) { wm.twiceRinse(); }
    else {
        if (option == 3 ) { wm.spin(); }
    }
}
}

private void spin() {
Engine e = new Engine(1700);
Timer t = new Timer(0,2400);
e.turnOn();
t.setDuration(3600);
t.start();
int time = t.getValue();
int duration = t.getDuration();
while(time != duration) {
time = t.count();
System.out.println(time);
}
e.turnOff();
}

} // -----
```

Bibliography

- [1] Cay S. Horstmann. *Core JAVA: Fundamentals (Vol. 1, 12th ed.)*. Pearson, 2023.
- [2] Herbert Schildt. *JAVA: The Complete Reference (12th ed.)*. McGraw Hill, 2021.
- [3] Kathy Sierra, Bert Bates, and Trisha Gee. *Head First JAVA (3rd ed.)*. O'Reilly Media, 2022.
- [4] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs (2nd ed.)*. MIT Press, 1996.
- [5] Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Conallen, and Kelli A. Houston. *Object-Oriented Analysis and Design with Applications (3rd ed.)*. Addison-Wesley Professional, 2007.
- [6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [7] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual (2nd ed.)*. Addison-Wesley Professional, 2004.
- [8] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd ed.)*. Addison-Wesley Professional, 2004.
- [9] Russ Miles and Kim Hamilton. *Learning UML 2.0*. O'Reilly Media, 2006.
- [10] Tom Pender. *UML Bible*. Wiley Publishing, Inc., 2003.
- [11] Martina Seidl, Marion Scholz, Christian Huemer, and Gerti Kappel. *UML @ Classroom: An Introduction to Object-Oriented Modeling*. Springer, 2015.
- [12] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Professional, 2001.

Webography

- [13] Oracle Corporation. *The JAVA Tutorials: About the JAVA Technology*. Available at: <https://docs.oracle.com/JAVASE/tutorial/getStarted/intro/definition.html> (n.d.).
- [14] Baeldung. *The JVM Architecture Explained*. Available at: <https://www.baeldung.com/jvm-architecture>, 2024.

- [15] OpenJDK Project. *OpenJDK: The Place to Collaborate on an Open-source Implementation of the JAVA SE Platform*. Available at: <https://openjdk.org/> (n.d.).
- [16] Computerphile. *Object Oriented Programming*. Available at: <https://www.youtube.com/watch?v=pTB0EiLXUC8>, 2017.
- [17] Ada Computer Science. *Object-oriented programming*. Available at: https://adacomputerscience.org/topics/object_oriented_programming (n.d.).
- [18] JAVA Language Specification (JLS). *Types, Values, and Variables*. Available at: <https://docs.oracle.com/JAVASE/specs/jls/se21/html/jls-4.html>, 2024.
- [19] Exercism. *The JAVA Track: Basics*. Available at: <https://exercism.org/tracks/JAVA> (n.d.).
- [20] W3Schools. *JAVA Tutorial*. Available at: <https://www.w3schools.com/JAVA/> (n.d.).
- [21] Educative. *OOP Design Confidential*. Available at: <https://www.educative.io/blog/object-oriented-programming> (n.d.).