

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



SECOND-CYCLE DEPARTMENT

COURSE HANDOUT

Object Oriented Programming

By

Dr Akkacha BEKADDOUR

Preface

Object-Oriented Programming (OOP) is not merely a programming technique; it is a way of perceiving and organizing the world through object logic. This publication is designed to guide the student through the transition from writing linear scripts to engineering complex, scalable systems.

The primary question students often ask is why move away from simple procedural logic? The answer lies in the scale of modern software. In an era where applications consist of millions of lines of code, procedural programming often leads to "spaghetti code" which are unmanageable, fragile, and difficult to debug.

Students need OOP because it provides a mental framework for managing complexity. By treating data and behavior as a single unit (the object), students learn to build software that mirrors real-world entities, making the logic more intuitive and the system more resilient to change.

However, writing code is only one part of the equation. This course places equal emphasis on Object-Oriented Modeling. For a future engineer, the ability to model a system by abstracting, categorizing, and defining the interactions between components before any code is written is a critical skill. Whether the task involves designing a software architecture or an industrial production line, modeling allows for the validation of logic and ensures the overall integrity of the system.

Our aim is to introduce students to Object-Oriented Programming and Modeling for the design of robust systems within the modern industrial landscape.

Content

Introduction	1
Chapter 1 : Introduction to JAVA	3
JAVA Environment	3
JAVA Definition	4
JAVA Technology	5
JAVA platform	6
JAVA Program Type	7
Chapter 2 : Core Concepts of OOP	9
Object	9
Communication via Messages	10
The Concept of Classes	12
The Concept of Inheritance	13
The Concept of Interface	14
Polymorphism	16
Abstract Classes	18
Encapsulation	19
From Principle to Implementation	21
Chapter 3 : JAVA Syntax	30
The Global Structure of JAVA Program	30
Variables in JAVA	31
Constants	34
Operators in JAVA	34
Data Types and Advanced Syntax in JAVA	36
Input and Output in JAVA	38
Control Structures in JAVA	40
Chapter 4 : Introduction to OOP Modeling	45
Principles of Modeling	45
Object Oriented Modeling	46
UML Overview	47
Chapter 5 : Behavioral Modeling : Use Case Diagrams	63

Use Case Definition	63
Steps for Creating Use Case Diagram	66
Elements of Use Case Diagram	67
Chapter 6 : Structural Modeling : Class Diagrams	71
Definition	71
Modeling Classes	72
Modeling Relationships	79
Chapter 7 : Behavioral Modeling : Sequence Diagrams	89
Definition	89
Frame	90
Interaction	90
Examples of Sequence Diagrams	94
Advanced Interaction Fragments	97
Conclusion	100
Bibliography and Webography	101

Introduction

In the contemporary engineering landscape, the analysis and design of complex systems have become sophisticated endeavors, necessitating high-level methodologies and specialized languages to manage increasing technical intricacies. The evolution of modern infrastructure and digital integration requires a shift from linear problem-solving to holistic system architecture. Central to this modern approach is the paradigm of Object-Oriented Programming (OOP), which has emerged as a cornerstone for architecting robust and scalable systems in an interconnected world.

The effective realization of these systems requires the mastery of dedicated design tools and programming environments tailored specifically to this object-centric philosophy. In the industrial sphere, where the integration of hardware and software is paramount, the ability to mirror real-world entities through abstract code structures is no longer optional. This technical transition is marked by the move toward modularity, where complex functions are decomposed into manageable, reusable components to ensure long-term stability and ease of maintenance.

As systems become more complex, combining smart design with solid coding has become essential. This need has led engineers to adopt standard modeling tools and modern programming practices. By connecting abstract ideas to real-world execution, these methods help teams manage difficult requirements and work together more clearly.

This course, designed for students navigating these advanced technical requirements, aims to provide a comprehensive introduction to system design and development, with a focus on essential concepts and methods. It specifically centers on the mastery of the Unified Modeling Language (UML) for architectural design and the JAVA language for professional programming. Through this dual approach, the curriculum ensures that future engineers are equipped with both the conceptual map and the technical engine required to build the next generation of industrial systems.

This handbook is structured to provide a comprehensive path through the world of Object-Oriented Programming, catering to both beginners and seasoned developers:

Chapter 1: Introduction to JAVA

We lay the foundation by introducing the JAVA platform and its core architecture. Students will gain a fundamental understanding of the JAVA ecosystem and the framework that enables cross-platform development.

Chapter 2: Core Concepts of OOP

This chapter explores the Object-Oriented Programming paradigm. Students will transition from theory to practice, mastering essential building blocks like inheritance, interfaces, encapsulation, and constructors.

Chapter 3: JAVA Syntax and Logic

We focus on the mechanics of the JAVA language. Students will gain the technical proficiency to utilize variables, operators, and control structures to build functional program logic.

Chapter 4: Introduction to UML

Students will delve into the Unified Modeling Language (UML) for system blueprinting. The course covers the basic building blocks and mechanisms necessary for professional architectural modeling.

Chapter 5: Behavioral Modeling: Use Case Diagrams

This chapter introduces Use Case diagrams for capturing system requirements. Students will learn to model actors and define a system's functional scope from an external perspective.

Chapter 6: Structural Modeling: Class Diagrams

We examine the Class Diagram, the most critical structural tool in UML. Students will learn to visualize the static relationships, attributes, and behaviors that form the backbone of an OOP system.

Chapter 7: Behavioral Modeling: Sequence Diagrams

The final chapter introduces Sequence Diagrams for dynamic modeling. It explores how objects communicate over time to fulfill specific system tasks and workflows.

Chapter 1 : Introduction to JAVA

The JAVA platform encompasses a robust suite of technologies designed to build and deploy secure, portable, and high-performance applications. Whether powering enterprise-level servers, mobile devices, or complex embedded systems, JAVA plays a pivotal role in bridging the gap between hardware-specific constraints and cross-platform flexibility. This equips developers with the capacity to write code once and execute it anywhere, addressing the challenges of modern, fragmented computing environments.

In this chapter, we will introduce the fundamental architecture and core principles of the JAVA ecosystem. We will explore the mechanics of the JAVA Virtual Machine (JVM) and the runtime environment that forms the essential foundation for professional software development.

1. JAVA environment

While you might already be familiar with languages like C, JAVA is a bit different. It isn't just a **programming language** used to write code; it is also a **platform**. This means it provides its own special environment that allows programs to run smoothly on almost any computer or device.

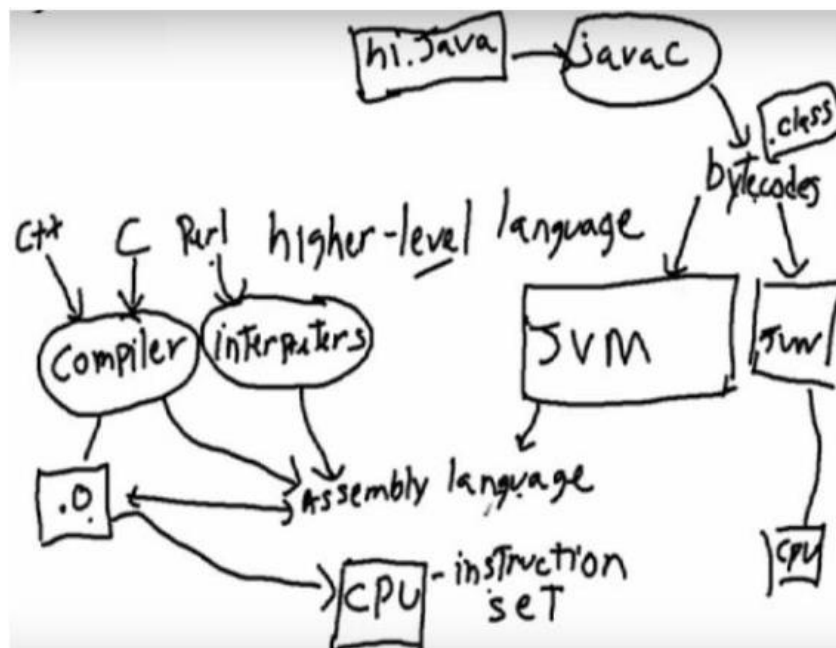


Figure 1. JAVA as both a Programming Language and a Platform

While a **compiler** translates source code into machine-readable object code (such as **.o** files in C) for the CPU to execute later, an **interpreter** processes the code directly. Instead of generating a permanent file, the interpreter reads, verifies, and executes the instructions on the CPU in real-time. JAVA is a hybrid language that utilizes both a **compiler** to transform code into bytecode and an **interpreter** to execute that bytecode.

2. JAVA Definition

JAVA was created in the early **1990s** by **James Gosling** and his team at **Sun Microsystems**. Originally named **Oak**, it was first meant for digital devices like remote controls. However, as the internet grew, it was renamed JAVA and released in **1995** as a tool to build powerful, flexible web applications. Today, it is managed by Oracle.

JAVA is a high-level programming language defined by these key features:

- **Simple & Object-Oriented:** It is designed to be easy to learn and focuses on "objects" to organize code.
- **Robust & Secure:** It has strong checks to prevent errors and built-in protection against security threats.
- **Portable & Neutral Architecture:** It doesn't care what kind of computer you use; the code is neutral, so it runs the same way everywhere.
- **High Performance & Multithreaded:** It is fast and can perform many tasks at the same time.
- **Distributed & Dynamic:** It is built to work across networks and can adapt to new information while running.

The JAVA language utilizes a dual process of **compilation** and **interpretation** to execute programs. During the first stage, source code is transformed into an intermediate format known as **bytecode**. Subsequently, an interpreter decodes and executes this bytecode instruction by instruction, allowing the program to run on the host computer.

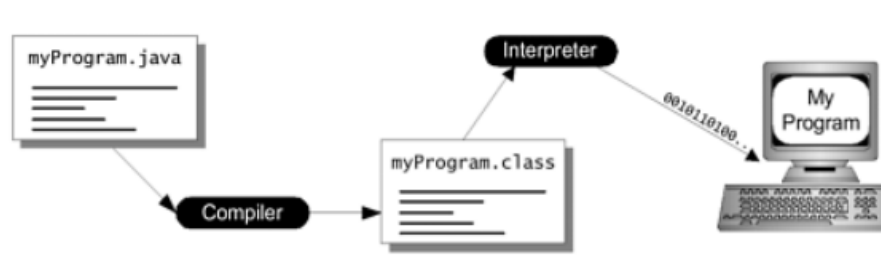


Figure 2. The Dual Process of JAVA Compilation and Interpretation

The principle of combining compilation and interpretation enables the "**Write Once, Run Anywhere**" capability. A JAVA program can be executed on any device as long as the appropriate interpreter, known as the **JAVA Virtual Machine (JVM)**, is installed to process the bytecode.

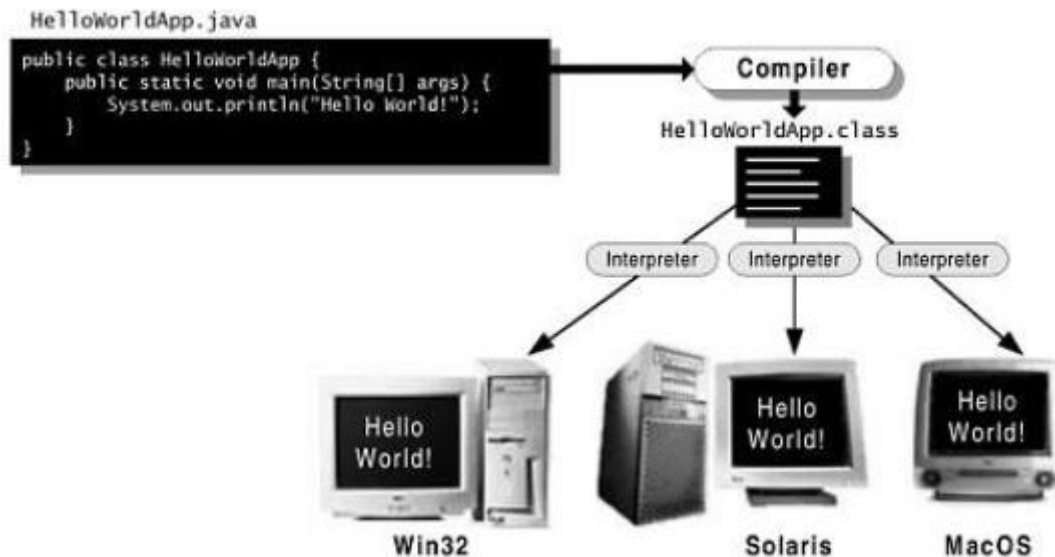


Figure 3. The "Write Once, Run Anywhere" (WORA) Principle via the JVM

A JAVA program undergoes a five-stage lifecycle before execution:

- **Editing Stage:** The source code is created using a text editor and saved to the disk as a **.JAVA** file.
- **Compilation Stage:** The JAVA compiler (**JAVAc**) translates the source code into bytecode, which is stored on the disk as a **.class** file.
- **Loading Stage:** The **Class Loader** reads the bytecode from the disk and transfers it into the system's memory (RAM).
- **Verification Stage:** The bytecode is checked to ensure it is valid, secure, and follows all JAVA safety rules and restrictions.
- **Execution Stage:** Finally, the **JAVA Virtual Machine (JVM)** translates the bytecode into machine language and executes the instructions on the computer.

3. JAVA Technology

JAVA technology provides a comprehensive ecosystem consisting of three essential layers to develop and execute applications:

- **The JDK (JAVA Development Kit):** This is the full development environment. It includes everything needed to **create** and **run** JAVA programs, such as the compiler (**JAVAc**), documentation tools, and the JRE.
- **The JRE (JAVA Runtime Environment):** This is the standalone package designed purely for **running** JAVA applications. It contains the necessary libraries and the JVM, but lacks development tools like the compiler.
- **The JVM (JAVA Virtual Machine):** This is the core engine that **executes** JAVA bytecode. It acts as the platform-independent bridge that translates bytecode into instructions the computer's hardware can understand.

Note on Modern Versions: Since JAVA 11 and continuing into the latest releases, the **JDK** has become the primary way to distribute JAVA. For most developers and users today, the JDK is the all-in-one package that provides the entire environment.

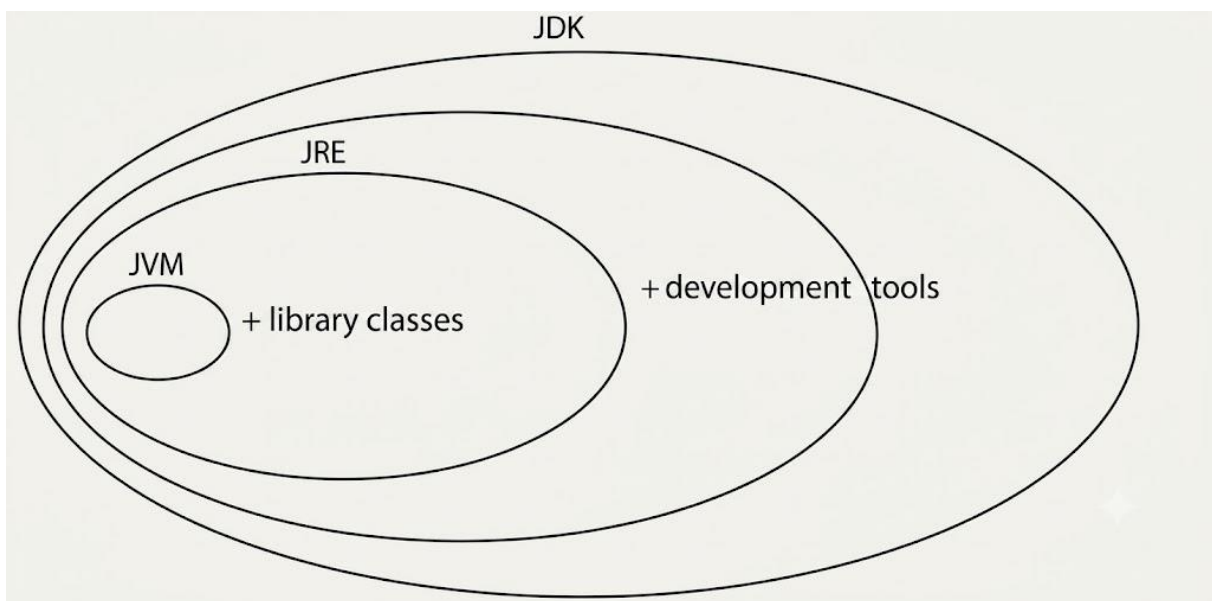


Figure 4. The Hierarchy of JAVA Environments: JDK, JRE, and JVM

4. JAVA Platform

A **platform** is typically the combination of hardware and the operating system (such as Windows, Linux, or macOS) where a program runs. The **JAVA platform** is unique because it is a software-only platform that sits on top of other hardware-based platforms. It consists of two primary components:

- **The JAVA Virtual Machine (JVM):** This is the heart of the platform. It is responsible for loading, verifying, and executing JAVA bytecode, ensuring the program runs smoothly regardless of the underlying hardware.
- **The JAVA Application Programming Interface (API):** This is a vast collection of ready-made software components that provide a wide range of capabilities, including:
 - **Essentials:** Handling objects, strings, numbers, threads, and input/output.
 - **Networking:** Managing URLs, TCP/UDP sockets, and IP addresses.
 - **Internationalization:** Allowing programs to automatically adapt to different languages and regions.
 - **Security:** Managing electronic signatures, keys, and access control.
 - **Database Connectivity (JDBC):** Providing uniform access to relational databases.

By combining the JVM and the API, JAVA frees programs from being dependent on specific hardware.

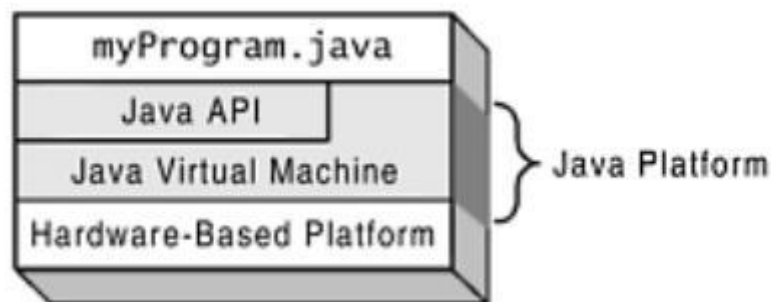


Figure 5. Hardware Independence through the JAVA API and JVM

5. JAVA Program Types

JAVA is incredibly versatile, allowing developers to create software for everything from tiny chips to massive server clusters. Depending on where the code lives and how it is used, JAVA programs generally fall into these modern categories:

- **Standalone Applications** These are programs that run directly on a computer's operating system (like Windows or macOS) via the JVM. They do not require a web browser.

- **Mobile Applications (Android)** JAVA has been the primary language for Android development for years. These programs are specially compiled to run on mobile devices, utilizing the hardware features of smartphones like GPS and cameras. Most classic apps found on the Google Play Store are an example of mobile applications.
- **Enterprise & Web Applications** These are "invisible" programs that run on powerful servers. They handle the complex logic behind big websites, banking systems, and online shopping carts.
- **Embedded & IoT Applications** JAVA is used in "smart" technology. Because the platform is so lightweight, it can run on small chips inside everyday objects like Smartwatches, SIM cards, television set-top boxes, and industrial robots.
- **JAVA Applets (Legacy)** Historically, these were small programs embedded in websites to provide interactivity. However, because modern web browsers no longer support the necessary plugins, applets are now considered obsolete and are rarely used in new development.

Chapter 2: Core concepts of OOP

The Object-Oriented Programming (OOP) paradigm shifts focus from linear logic to modular, real-world modeling. By organizing code into self-contained entities, OOP enables the creation of scalable and maintainable software that can adapt to complex requirements.

In this chapter, we explore the essential principles of this approach. We will define the relationship between **Classes** and **Objects**, examine how data and behavior are encapsulated, and study the mechanisms through which objects communicate via **Messages**. This conceptual shift is the first step toward mastering professional JAVA development and architectural design.

1. Object

In the physical world, objects are everywhere: a cat, a desk, or a bicycle. Every real-world object is defined by two fundamental characteristics: **State** and **Behavior**.

- **State:** The attributes or data an object possesses (e.g., a cat has a name, color, and breed).
- **Behavior:** The actions an object can perform (e.g., a cat can meow, hunt, or stretch).

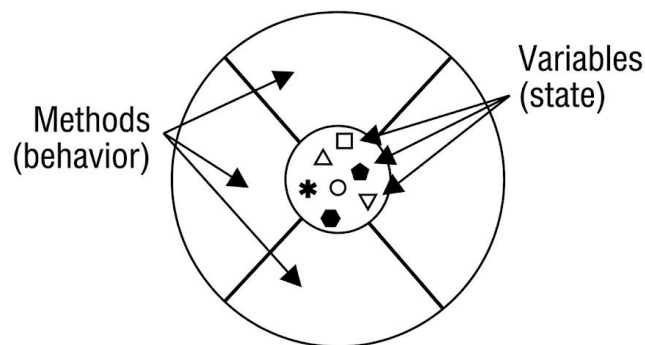


Figure 6. Anatomy of a Software Object

Software objects are modeled after these real-world counterparts. In a JAVA program, an object is a self-contained unit that bundles data and functionality together:

- **State (Variables):** The data maintained by the object. For a bicycle object, variables might store the current speed (10 mph) or the current gear (5th).
- **Behavior (Methods):** The functions or subroutines that define what the object can do. A bicycle object would include methods to brake, accelerate, or change gears.

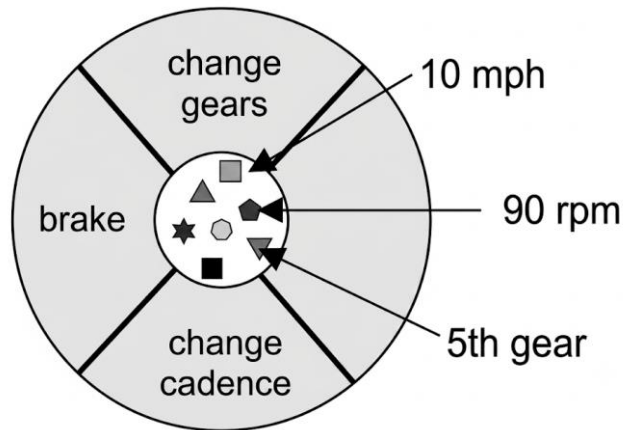


Figure 7. The real object bicycle modeled by a code object

By structuring programs around objects, JAVA provides two critical software engineering benefits:

- **Modularity:** Objects are independent units. You can modify the source code of one object without affecting others, making the system easier to build, test, and reuse.
- **Information Protection:** Objects control what information they reveal. By hiding complex internal logic and exposing only necessary methods, an object protects its data and simplifies how other parts of the program interact with it. For example, you can shift gears on a bicycle without needing to understand the mechanical complexity of the chain and derailleurs.

2. Communication via messages

In a real-world scenario, a single object is rarely useful in isolation. For example, a bicycle hanging in a garage is just a collection of metal and rubber; it only fulfills its purpose when another object (a rider) interacts with it. Similarly, a JAVA program consists of multiple objects that interact to achieve complex functionality.

Software objects communicate by sending **messages**. When Object A needs Object B to perform a specific action, Object A invokes one of Object B's methods. This request is known as "sending a message."

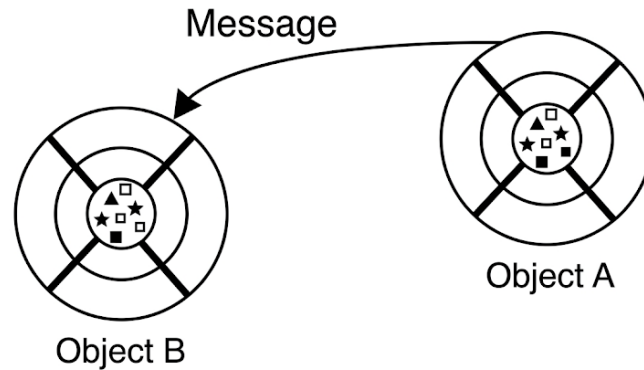


Figure 8. Object interaction using messages.

Often, the receiving object requires specific details to execute a task correctly. For instance, telling a bicycle to "change gears" is not enough; we must specify *which* gear. This extra data is passed along with the message as **parameters**.

A standard message consists of three essential components:

- **The Recipient:** The specific object to which the message is sent (e.g., myBicycle).
- **The Method:** The name of the action to be performed (e.g., changeGears).
- **The Parameters:** Any additional information required by the method (e.g., lowerGear).

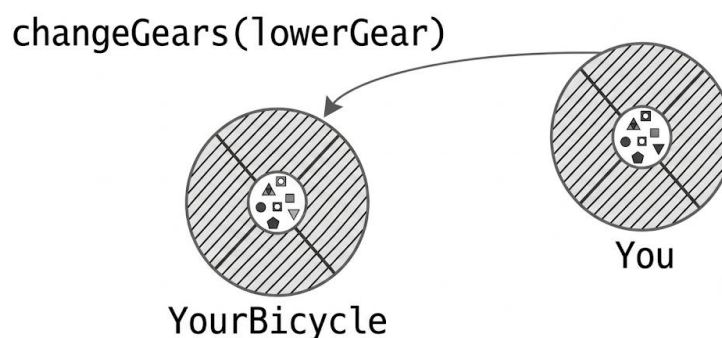


Figure 9. Example of a message in JAVA.

By utilizing these message components, objects can work together seamlessly while remaining distinct, modular entities.

3. The concept of classes

In the real world, we often encounter many objects of the same kind. Your bicycle is not unique in its basic design; it is one of millions that share the same fundamental structure. In Object-Oriented Programming, we say that your bicycle is an **instance** of a general category known as a **Class**.

We can see a class as a **prototype** or a **blueprint**. It would be highly inefficient for a factory to design a new set of plans for every single bicycle they build. Instead, they create one master blueprint and use it to manufacture thousands of identical bikes.

- **The Class:** The blueprint that defines what variables and methods all objects of that type will have.
- **The Object:** The actual, physical bicycle created from that blueprint.

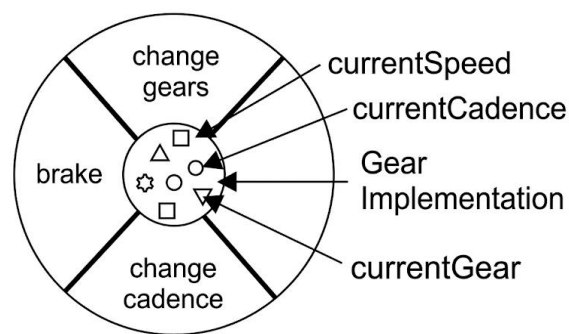


Figure 10. The Bicycle class acting as a blueprint for multiple objects.

When we create objects from a class, they all possess the same variables, but their internal data can differ.

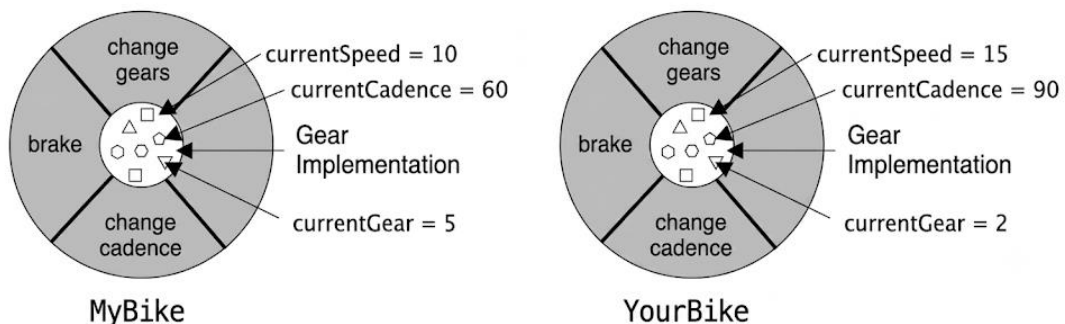


Figure 11. Two bicycle object instances sharing a class variable.

To distinguish between individual data and shared data, JAVA categorizes variables into two types:

- **Instance Variables:** These belong to the specific object. Your bicycle might be in **3rd gear** while your friend's bicycle is in **5th gear**. Each "instance" maintains its own state.
- **Class Variables:** These contain information shared by *every* object created from that class. For example, if every bicycle model produced by a factory is designed with exactly **18 gears**, that value is stored in a class variable. Every individual bike instance then references this shared value.

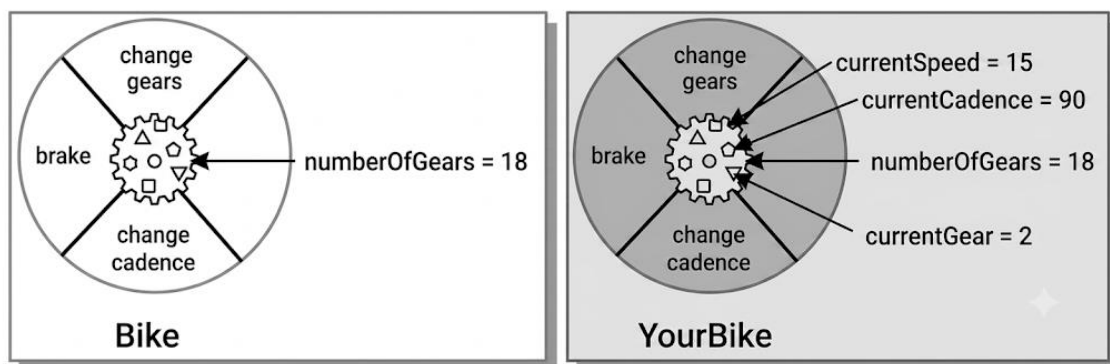


Figure 12. Relationship between a shared Class Variable and specific Instance Variables

While objects provide **modularity** and **protection**, classes provide **reusability**. Just as a manufacturer reuses a blueprint to save time and resources, a programmer reuses a class to create thousands of objects without rewriting the code. By defining the logic once in a class, you can deploy it infinitely across your application.

4. The concept of inheritance

In Object-Oriented Programming, classes can be organized into hierarchies. Often, different types of objects share many characteristics but remain distinct in their specific functions. For example, **Mountain Bikes**, **Racing Bikes**, and **Tandems** are all different, yet they are all fundamentally **Bicycles**.

Inheritance allows one class to derive its state and behavior from another.

- **Superclass (Parent):** The general class that provides common features (e.g., Bicycle).
- **Subclass (Child):** The specialized class that inherits from the superclass (e.g., MountainBike).

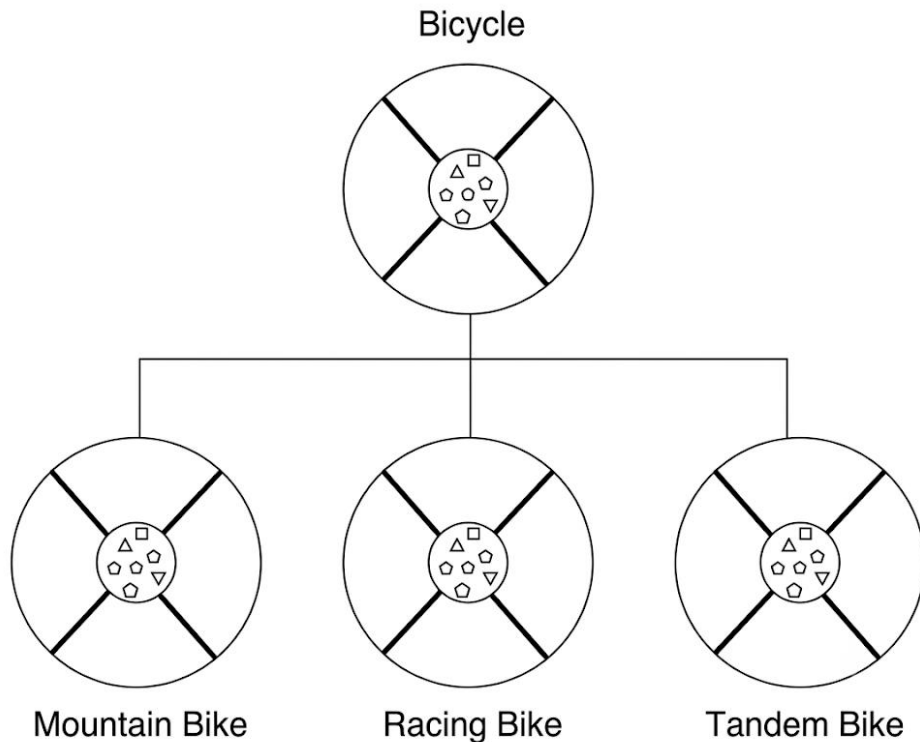


Figure 7: Hierarchy of the Bicycle superclass and its specialized subclasses.

Through inheritance, the relationship between a parent and child class is defined by three primary mechanisms:

- **Shared State and Behavior:** Every subclass automatically inherits the variables (cadence, speed) and methods (braking, shifting) of the superclass. This avoids redundant code.
- **Specialization:** Subclasses can define their own unique features. For instance, a TandemBike might add a variable for secondSeat, while a MountainBike might include an additional suspension method.
- **Method Overriding:** A subclass can modify an inherited behavior to fit its specific needs. If a MountainBike has an extra set of gears, it can **override** the standard changeGears method to include logic for the lower gear ratio.

Inheritance promotes **reusability** and **extensibility**. Instead of writing the same "braking" logic for every type of bike, a programmer writes it once in the Bicycle superclass. Subclasses then simply inherit that logic and add only what makes them unique, creating an efficient and organized "inheritance tree."

5. The concept of interface

In everyday life, an **interface** is a bridge that allows unrelated entities to interact. For example, a remote control serves as the interface between a human and a television, and a common language acts as an interface between two people.

In JAVA, an interface is a functional contract that allows unrelated objects (objects from different class hierarchies) to communicate. While inheritance links related objects (like different types of bikes), an interface links objects that might have nothing in common except for a specific capability.

It is important to note that an interface is **not a class**. It follows a distinct set of rules:

- **No State:** Unlike a class, an interface does not contain instance variables to store data.
- **Method Signatures:** It only contains constants and "generic" methods (declarations without the actual code).
- **Implementation:** The classes that use the interface are responsible for providing the actual implementation of those methods.

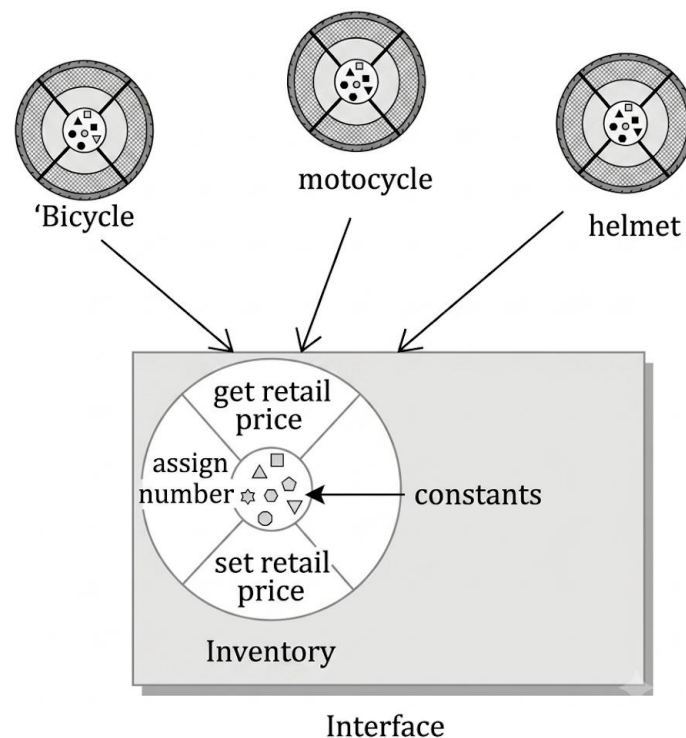


Figure 13: Unrelated objects (Bicycle, Motorcycle, Helmet) utilizing a common interface to manage pricing and tracking.

Imagine a store that sells bicycles, protective helmets, and motorcycles. the **Inventory Interface** acts as a professional contract. It doesn't care *what* an object is (a vehicle or a piece of clothing); it only cares that the object can perform specific inventory tasks. The two methods shown are "requirements" that every object must fulfill:

- **get retail price / set retail price:** This allows the system to handle money. The **Bicycle** might calculate its price based on parts, while the **Motorcycle** includes taxes and registration. Because of the interface, the cash register doesn't need to know these details—it just asks for the "Price," and the object provides it.
- **assign number:** This is the "Tracking" requirement. Every item in a store needs a unique ID (like a barcode). By implementing this method, the **Helmet**, **Bicycle**, and **Motorcycle** all agree to accept a tracking number so they can be stored in the same database.

6. Polymorphism

In biology, polymorphism refers to the ability of an organism or species to exist in many different forms. In Object-Oriented Programming, **polymorphism** is the principle that objects belonging to the same class hierarchy can exhibit different behaviors. Essentially, it means "one interface, many implementations."

Polymorphism allows objects to perform the same action in different ways. For example, consider a general Bicycle class that includes a brake method. Even though two objects are both "bicycles," how they execute that brake command can differ:

- **MyBike:** Uses a simple braking system. When the brake method is called, it requires no extra information—it just slows down.
- **YourBike:** Features a more complex double-brake system. When its brake method is called, it requires a **parameter** (such as "front" or "rear") to know which specific brake to apply.

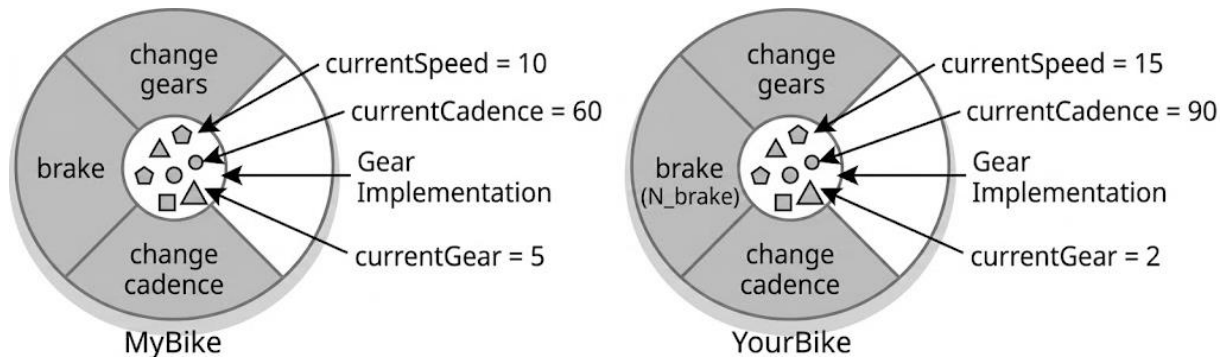


Figure 14. Two object instances (MyBike and YourBike) of the same Bicycle class exhibiting different behaviors.

Polymorphism allows a programmer to treat different objects as if they were the same type while letting each object handle its own specific logic.

Imagine you have a list of various bikes (Mountain Bikes, Racing Bikes, etc.). You can tell the entire group to `brake()`. Because of polymorphism:

- The **Mountain Bike** might use heavy-duty disc brakes.
- The **Racing Bike** might use lightweight rim brakes.
- The **Tandem** might engage brakes on both sets of wheels.

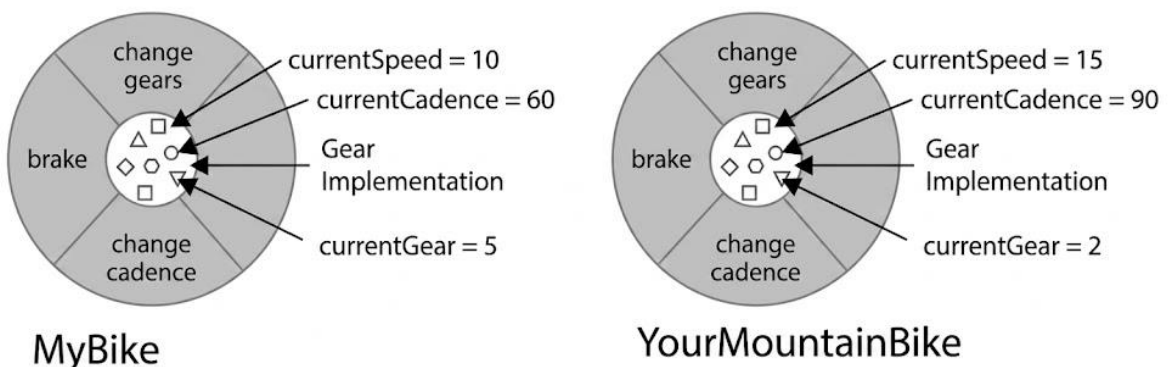


Figure 15: Polymorphism in action across different bicycle types in the same hierarchy.

Benefits of Polymorphism

- **Flexibility:** You can add new types of objects (like an `ElectricBike`) to your program without changing the code that manages the bikes. You simply define how that new bike "brakes."

- **Simplicity:** It reduces complexity by allowing you to use the same method name (brake) for different actions, rather than having to remember unique names like `mountainBikeBrake()` or `racingBikeBrake()`.

7. Abstract classes

In some cases, you may want to design a class that represents a general concept rather than a specific, finished product. This is where **Abstract Classes** come into play.

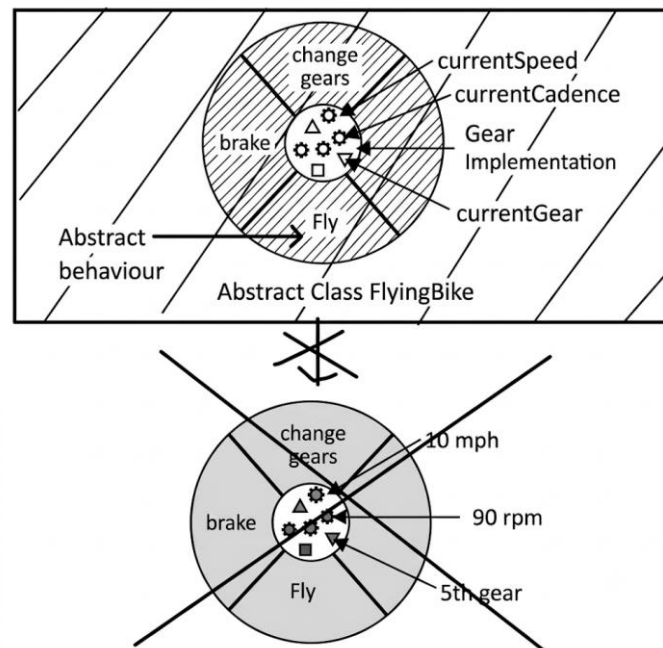


Figure 16. Attempted instantiation of the Abstract Class FlyingBike and why it is prohibited.

Imagine you want to invent a futuristic "Flying Bicycle." You know it should have the ability to fly, but at the design stage, you haven't yet figured out the mechanical details of *how* it stays airborne (e.g., propellers vs. jet engines).

In programming, you would declare the `fly()` behavior without writing the code for it. This is called an **abstract method**. A class that contains at least one abstract method is an **Abstract Class**.

To ensure these classes are used correctly within a program, they are governed by three fundamental rules:

- **No Direct Objects:** You cannot create an object from an abstract class. Since the class is "incomplete" (like a blueprint missing the engine schematics), the computer wouldn't know what to do if you tried to use it.
- **The Concrete Requirement:** To actually use an abstract class, you must first create a **subclass** that inherits from it. This subclass must provide the "concrete" implementation (the actual code) for all the abstract methods.
- **Purpose:** Abstract classes act as a strict template. they ensure that all future specialized versions of the class follow a specific structure while allowing each version to handle the "how" differently.

To visualize how this concept moves from an idea to a functional object, consider this step-by-step progression:

- **Abstract Superclass:** FlyingBicycle (defines that it *must* fly, but not how).
- **Concrete Subclass:** PropellerBike (inherits from FlyingBicycle and implements the fly() method using propeller logic).
- **Result:** You can now create an object of PropellerBike because it is a finished, "concrete" version of the abstract idea.

8. Encapsulation

Encapsulation is the principle of shielding an object's internal state by wrapping its variables within protective methods. In simpler terms, it is the practice of "hiding" the data so that it can only be accessed or modified through authorized behaviors.

Imagine the currentSpeed variable of a bicycle. In a well-encapsulated system, you cannot simply reach into the object and manually type a new number for the speed. Instead, the speed changes only as a result of a specific action, such as invoking the changeGears or applyBrakes methods.

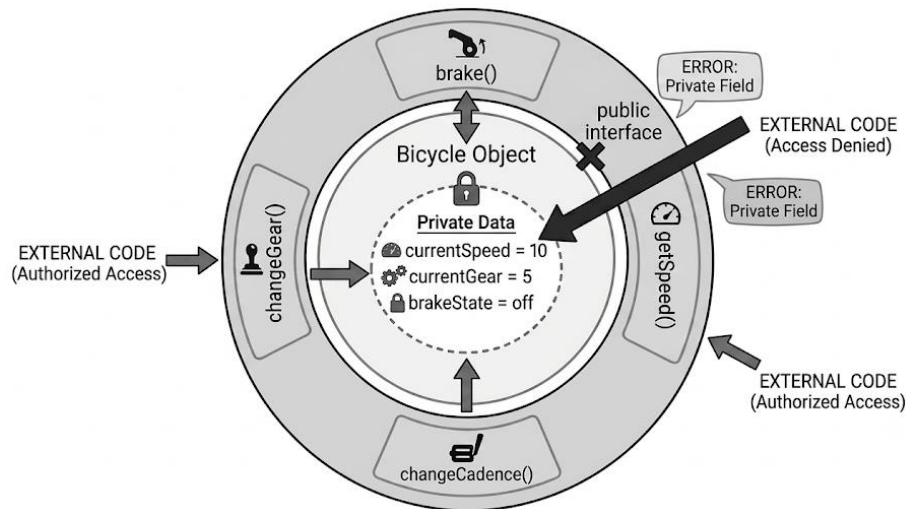


Figure 17: Encapsulation acting as a protective barrier around an object's data.

By implementing these protective boundaries, developers gain several critical technical advantages:

- **Information Protection:** This keeps the internal "secrets" of how an object works hidden. By only showing what is necessary, you prevent outside parts of the program from messing with the object's internal logic.
- **Data Integrity:** This acts like a quality control check. It ensures that the information inside an object always "makes sense" because it can only be changed through methods that follow the rules of the program.
- **Prevents Corruption:** By restricting direct access, you prevent external code from accidentally setting a variable to an impossible value (like setting a bicycle's speed to -50 mph).
- **Simplified Interaction:** Other parts of the program don't need to know exactly how a value is calculated; they just need to know which method to call to get the result they want.
- **Ease of Maintenance:** If you want to change how the speed is calculated later, you only have to fix the code inside that one method instead of searching through your entire project for every time the speed variable was mentioned.

9. From principle to implementation

After understanding the core theories of Object-Oriented Programming (OOP), we will now implement them using **JAVA**. In this course, JAVA serves as our primary tool to bring these concepts to life.

➤ Class Definition and Visibility

To define a class in JAVA, we use the class keyword along with **Access Modifiers** to control data security:

- **public**: Visible to all classes.
- **protected**: Visible to the class and its subclasses.
- **private**: Visible only within the class (The basis of **Encapsulation**).

```
class Bicycle {  
  
    public int cadence = 0;    // Public attribute  
  
    protected int gear = 1;    // Protected attribute  
  
    private int speed = 0;    // Private attribute (Encapsulated)  
  
    void changeCadence(int newValue) { cadence = newValue; }  
  
    void changeGear(int newValue) { gear = newValue; }  
  
    void speedUp(int increment) {  
  
        speed = speed + increment;  
  
    }  
  
    void applyBrakes(int decrement) {  
  
        speed = speed - decrement;  
  
    }  
  
    void printStates() {  
  
        // System.out.println is equivalent to printf in C  
  
        System.out.println("cadence:" + cadence + " speed:" +  
speed + " gear:" + gear);  
    }  
}
```

```
    }  
}
```

➤ Object Creation (Instantiation)

To create an object, we use the **new** keyword. This allocates a block in the **RAM (Heap)** for the object's attributes and methods.

Syntax: `Bicycle bike1 = new Bicycle();`

➤ Inheritance

Inheritance allows a **Subclass** to acquire the properties of a **Superclass** using the keyword **extends**.

```
class MountainBike extends Bicycle {  
    public int length;  
    void print() {  
        System.out.println("length:" + length);  
    }  
}
```

To instantiate a Subclass object: `MountainBike obj = new MountainBike();`

obj has access to all methods in Bicycle AND MountainBike.

To instantiate a Superclass object: `Bicycle objs = new MountainBike();`

objs can **only** access methods defined in the Bicycle class.

➤ Interfaces and Abstract Classes

An interface is a contract. Classes use the keyword **implements** and must define all listed methods.

```
interface Bicycle {  
    void speedUp(int increment);  
}
```

```
class implementedBicycle implements Bicycle {  
  
    int cadence = 0;  
  
    int speed = 0;  
  
    int gear = 1;  
  
    void speedUp(int increment) { speed = speed + increment; }  
  
}
```

A class declared with **abstract** cannot be instantiated. It can contain **abstract methods** (methods without a body) that subclasses must fill in.

```
abstract class FlyBicycle {  
  
    abstract void fly();  
  
}
```

If we want to implements the abstract method:

```
class Bicycle extends FlyBicycle {  
  
    void Fly() { // code here  
  
    }  
  
}
```

➤ Polymorphism

JAVA supports three main types of polymorphism:

- **Method Overloading:** Same method name, different parameters (within the same class or class hierarchy).

```
class Bicycle {  
  
    // ..... int speed = 0;  
  
    void applyBrakes() { speed = speed - 1;  
  
    }  
  
    applyBrakes(int decrement) {
```

```
speed = speed - decrement; }  
  
// .....  
  
}
```

- **Method Overriding:** A subclass redefines a method from the superclass to change its behavior.

```
class MountainBike extends Bicycle {  
  
// this class inherit from Bicycle class  
  
void applyBrakes() {  
  
speed = speed - 2;} // override applyBrakes() of Bicycle  
  
}  
  
}
```

- **Constructor Polymorphism:** Multiple ways to initialize an object.

A Constructor is a special type of method used to initialize an object. It sets the initial state (attributes) of the object at the moment of creation. It must have the exact same name as the Class and it never has a return type (no void, int, etc.). It is called automatically when we use the **new** keyword.

We distinguish two type of constructor:

A. Default Constructor

If you do not write any constructor, JAVA automatically provides a "default" one. It initializes numerical variables to 0 and objects/strings to null.

```
class Bicycle {  
  
    int cadence = 0;  
  
    int speed = 0;  
  
    int gear = 1;  
  
    void printStates() {
```

```
        System.out.println("cadence:" + cadence + " speed:" +
speed + " gear:" + gear);
    }
}

class BicycleDemo {

    public static void main(String[] args) {

        // Calling the system-defined default constructor

        Bicycle bike1 = new Bicycle();

        bike1.printStates(); // Output: cadence:0 speed:0 gear:1
    }
}
```

B. User-Defined Constructors

When you want specific initial values, you define your own constructors. There are three main variations:

a. Non-parameterized Constructor

Used to set fixed default values for every new object.

```
class Bicycle {

    int cadence, speed, gear;

    public Bicycle() {

        cadence = 0;

        speed = 0;

        gear = 0;

    }

    // Note: You can have a method with the same name if it
has a return type
```

```
void Bicycle() {  
  
    System.out.println("This is a regular method, not a  
constructor!");  
  
}  
  
}
```

b. Parameterized Constructor

Allows you to pass specific values for some attributes when creating the object.

```
class Bicycle {  
  
    int cadence, speed, gear = 1;  
  
    public Bicycle(int c, int s) {  
  
        cadence = c;  
  
        speed = s;  
  
    }  
  
}
```

c. Immediate (Full) Constructor

Used when you want to initialize all attributes at once.

```
class Bicycle {  
  
    int cadence, speed, gear;  
  
    public Bicycle(int c, int s, int g) {  
  
        cadence = c;  
  
        speed = s;  
  
        gear = g;  
  
    }  
  
}
```

Note: If you define any user-defined constructor, JAVA will no longer provide the automatic default constructor. You must define a no-argument constructor manually if you still wish to use it.

In JAVA, you can have multiple constructors in the same class as long as they have different parameter lists. This is known as Constructor Overloading. JAVA decides which one to use based on the arguments you provide.

```
class Bicycle {  
    int cadence, speed, gear;  
  
    // Constructor 1: Full initialization  
    public Bicycle(int c, int s, int g) {  
        cadence = c;  
        speed = s;  
        gear = g;  
    }  
  
    // Constructor 2: Default fixed values  
    public Bicycle() {  
        cadence = 20;  
        speed = 5;  
        gear = 1;  
    }  
  
    // Constructor 3: Partial initialization  
    public Bicycle(int c, int s) {  
        cadence = c;  
        speed = s;  
        gear = 0; // Default if not provided  
    }  
}
```

```
void printStates() {  
    System.out.println("cadence:" + cadence + " speed:" +  
speed + " gear:" + gear);  
}  
  
}  
  
class BicycleDemo {  
    public static void main(String[] args) {  
        Bicycle bike1 = new Bicycle();           // Uses  
Constructor 2  
        Bicycle bike2 = new Bicycle(50, 10, 2);  // Uses  
Constructor 1  
        Bicycle bike3 = new Bicycle(63, 12);     // Uses  
Constructor 3  
        bike1.printStates(); // cadence:20 speed:5 gear:1  
        bike2.printStates(); // cadence:50 speed:10 gear:2  
        bike3.printStates(); // cadence:63 speed:12 gear:0  
    }  
}
```

Unlike languages like other language like C++, JAVA does not use destructors. Memory management is handled by the Garbage Collector. This is an automatic tool that identifies objects that are no longer being used by the program and destroys them to free up memory space automatically.

➤ **Encapsulation**

Encapsulation protects data integrity by hiding variables behind a private barrier. Users must use public methods to interact with the data.

```
class Bicycle {  
    private int cadence = 0; // Encapsulated
```

```
public int gear = 1;      // Not encapsulated

public int getcadence() { // encapsulation
    return cadence; }

}

// ....

// accessing stats

bike1.gear = 3;          // Works

bike1.cadence = 10;     // ERROR: cadence is private!

int x = bike1.getcadence(); // Works
```

Chapter 3 : JAVA syntax

The syntax of a JAVA program is built upon traditional components found in most programming languages—variables, operators, and control structures. However, JAVA's defining characteristic is its uncompromising commitment to the **Object-Oriented** paradigm.

In JAVA, your program is not a collection of loose functions; it is strictly a class or a set of classes working together. For the system to execute your code, one of these classes must contain the specific entry point: `public static void main(String[] args)`. By wrapping every piece of logic within a class structure, JAVA ensures that your code remains modular and organized from the very first line. This chapter explores the rules and components that govern how we write within this structured environment.

1. The Global Structure of a JAVA Program

Every JAVA program follows a strict hierarchical structure. It begins with **imports**, which bring in external tools. This is followed by the **class definition**, which acts as the outer container. Inside, we find the **main method**, the mandatory entry point. Within this method, we declare **variables** and use **objects**—which are initialized via **constructors**—to perform specific tasks.

```
// 1. IMPORT: Loading external utility classes

import JAVA.util.Scanner;

// 2. PRINCIPAL RUNNABLE CLASS: The main container

public class BasicsDemo {

    // 3. MAIN METHOD: The mandatory starting point

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

```
// 4. VARIABLES: Storing local data

int sum = 0;

int n;

/* 5. OBJECT & CONSTRUCTOR:

    'Scanner' is the Class.

    'scan' is the Object.

    'new Scanner(System.in)' uses the Constructor with a
Parameter. */

Scanner scan = new Scanner(System.in);

// Read input from the user

n = scan.nextInt();

// 6. CONTROL STRUCTURE: A 'for' loop for logic

for (int i = 1; i <= n; i++) {

    sum += i;

}

// 7. CLASS METHOD: Built-in behavior to display results

System.out.println("Sum = " + sum);

}

}
```

While a JAVA program can consist of dozens or even hundreds of different classes, they must all be organized around a single **Principal Runnable Class**. This class serves as the conductor of the entire application because it contains the main method. Without this specific "starting point," the program exists only as a collection of blueprints (classes) but has no way to actually begin execution.

2. Variables in JAVA

In JAVA, the **state** of an object is stored in variables. A variable is essentially a reserved memory location named by an **identifier** used to store a specific piece of data. Every variable must be declared with a **Type** and a **Name** following the syntax: `TYPE NAME;` .

JAVA provides a variety of primitive types to handle different categories of data, such as integers, real numbers (decimals), characters, and booleans. The following program demonstrates these types and their maximum possible values using JAVA's built-in constants.

```
class MaxVariablesDemo {  
    public static void main(String args[]) {  
        // --- Integers ---  
  
        byte largestByte = Byte.MAX_VALUE;           // 8-bit  
        short largestShort = Short.MAX_VALUE;        // 16-bit  
        int largestInteger = Integer.MAX_VALUE;      // 32-bit  
        long largestLong = Long.MAX_VALUE;           // 64-bit  
  
        // --- Real Numbers (Floating Point) ---  
  
        float largestFloat = Float.MAX_VALUE;  
        double largestDouble = Double.MAX_VALUE;  
  
        // --- Other Primitive Types ---  
  
        char aChar = 'S';  
        boolean aBoolean = true;  
  
        // Displaying the results  
  
        System.out.println("The largest byte value is " +  
largestByte);  
  
        System.out.println("The largest short value is " +  
largestShort);  
  
        System.out.println("The largest integer value is " +  
largestInteger);  
    }  
}
```

```
        System.out.println("The  largest  long  value  is  "  +
largestLong);

        System.out.println("The  largest  float  value  is  "  +
largestFloat);

        System.out.println("The  largest  double  value  is  "  +
largestDouble);

        // Character check

        if (Character.isUpperCase(aChar)) {

            System.out.println("The character " + aChar + " is
upper case.");

        } else {

            System.out.println("The character " + aChar + " is
lower case.");

        }

        System.out.println("The  value  of  aBoolean  is  "  +
aBoolean);

    }

}
```

The expected output of the precedent program is :

The largest byte value is 127

The largest short value is 32767

The largest integer value is 2147483647

The largest long value is 9223372036854775807

The largest float value is 3.40282e+38

The largest double value is 1.79769e+308

The character S is upper case.

The value of aBoolean is true

When naming your variables, you must adhere to the following JAVA naming conventions:

- A. **Starting Character:** It must begin with a letter.
- B. **Prohibited Words:** It cannot be a JAVA **keyword** (e.g., class, static, void), a boolean literal (true, false), or the word null.
- C. **Special Characters:** Only the underscore _ is permitted; other special characters (like @, #, or space) are invalid.

3. Constants

If you want to create a variable whose value cannot be changed once assigned, use the **final** keyword. This designates the variable as a **constant**.

```
// Syntax: final TYPE NAME = VALUE;
```

```
final double PI = 3.14159;
```

4. Operator in JAVA

JAVA provides a comprehensive set of operators to manipulate variables and perform logic. These are categorized into arithmetic, relational, logical, and bitwise operations.

Category	Operators	Description
Arithmetic	+, -, *, /, %	Standard math (Addition, Subtraction, Multiplication, Division, and Remainder/Modulo).
Relational	==, !=, >, >=, <, <=	Used for comparisons; they return a boolean (true or false).
Logical	&&, ^	
Bitwise (Binary)	&, , ^, ~	AND, OR, XOR, and NOT performed at the bit level.

Another arithmetic operator are the ++ and – operators. The position of ++ and -- determines when the variable is updated relative to the calculation. This is a common source of logic errors.

- **Prefix (++m):** Increments the value **before** it is used in the expression.
- **Postfix (n++):** Increments the value **after** the current expression is evaluated.

```
int m = 7;

int n = 7;

// Prefix: m becomes 8, then 2 * 8 = 16

int a = 2 * ++m;

System.out.println("a = " + a + ", m = " + m); // Output: a =
16, m = 8

// Postfix: 2 * 7 = 14 is calculated first, then n becomes 8

int b = 2 * n++;

System.out.println("b = " + b + ", n = " + n); // Output: b =
14, n = 8
```

One of specific operator in JAVA is the ternary operator, which is a shorthand way of writing an if-else statement. It evaluates a condition and returns one of two values.

Syntax: condition ? expression1 : expression2;

```
// If x is less than y, a is assigned x. Otherwise, a is assigned
y.
```

```
int a = (x < y) ? x : y;
```

Another type of operators are the bitwise operators that manipulate the individual bits of an integer. This is often used for low-level flag management or extracting specific data from a binary stream.

- & (AND): 1 if both bits are 1.
- | (OR): 1 if at least one bit is 1.
- ^ (XOR): 1 if bits are different.

- ~ (NOT): Flips all bits.

```
// Extracts the 4th bit and reduces it to a 0 or 1
```

```
int fourthBitFromRight = (n & 8) / 8;
```

The next table give outputs for several value of n (in binary 8 = 1000):

Input (n)	Binary of n	Result of (n & 8)	Final Output (/8)
7	0111	0111&1000=0000 (0)	0
9	1001	1001&1000=1000 (8)	1
15	1111	1111&1000=1000 (8)	1
2	0010	0010&1000=0000 (0)	0

5. Data Types and Advanced Syntax in JAVA

JAVA is a **strongly typed language**, meaning every variable must be declared with a specific type before it can be used.

JAVA features eight primitive types, categorized by the nature and size of the data they hold:

Category	Type	Size	Description
Integers	byte	8-bit	Very small integers (-128 to 127)
	short	16-bit	Small integers
	int	32-bit	Standard integers (Most common)
	long	64-bit	Large integers
Real Numbers	float	32-bit	Single-precision floating point
	double	64-bit	Double-precision (Standard for decimals)
Other	char	16-bit	A single Unicode character
	boolean	1-bit/8-bit	Logical true or false

Often, we need to move data from one type to another. This is called type conversion. We could find two type of conversion:

- **Widening Conversion:** Automatic (e.g., int to float). However, large integers converted to float might lose some precision.
- **Typecasting (Narrowing):** Required when there is a risk of losing information.

```
double x = 9.997;
```

```
int nx = (int) x; // nx becomes 9 (decimal is truncated)
```

In JAVA, a **String is not a primitive type; it is a Class**. Therefore, every string we create is an **object**.

```
String e = "ESSAT";
```

```
String a = new String("ESSAT");
```

```
char ch[] = {'E', 'S', 'S', 'A', 'T'};
```

```
String b = new String(ch);
```

JAVA dispose of a certain number of String methods used to manipulate Strings, from which we can cite:

- `length()`: Returns number of characters.
- `substring(start, end)`: Extracts a portion of the string.
- `charAt(index)`: Returns the character at a specific position.
- `equals()`: Compares content (Use this instead of `==` for strings!).
- `toUpperCase()` / `toLowerCase()`: Changes the case of the text.

Like Strings, JAVA can handle also **arrays**, which are treated as objects in JAVA. Once an array is created, its size is fixed.

```
int[] a = new int[100]; // Creates an array of 100 integers
```

```
for (int i = 0; i < a.length; i++) {
```

```
    a[i] = i; // Accessing by index
```

```
}
```

Note: JAVA provides the **ArrayList** class as a dynamic alternative to the standard array. While a standard array has a **fixed size** determined at the time of creation, an ArrayList

belongs to the JAVA Collections Framework and can **grow or shrink** automatically as you add or remove elements.

JAVA provides also massive libraries of pre-written methods organized into **packages**:

A. Numbers and Math (Math class)

The Math class provides static methods for trigonometry and arithmetic:

- Math.sqrt(x): $\sqrt{x}$
- Math.pow(x, y): x^y
- Math.abs(x): Absolute value $|x|$
- Math.random(): Returns a decimal between 0.0 and 1.0.

B. Characters (Character class)

Useful for validating user input:

- Character.isDigit(c): Checks if it's a number.
- Character.isLetter(c): Checks if it's a letter.

C. Array Utilities (Arrays class)

Requires import `JAVA.util.Arrays;`

- Arrays.toString(arr): Converts an array to a readable string format.
- Arrays.asList(arr).contains("Value"): Checks if an element exists in the array.

In order to handle mathematics constant, JAVA use:

Constant	Type	Value
Math.PI	double	3.141592653589793
Math.E	double	2.718281828459045

6. Input and output in JAVA

To make a program interactive, JAVA uses specialized objects to handle data flowing into the system (Input) and data displayed to the user (Output).

Reading input is performed by the **Scanner** class. Because this class is not in the default JAVA package, you must import it from JAVA.util and then create a Scanner object attached to the standard input stream (System.in).

```
import JAVA.util.Scanner; // Must be at the very top of your
code

// Create the scanner object

Scanner in = new Scanner(System.in);
```

The Scanner class provides specific methods to capture different data types:

- **Strings:** in.nextLine() (reads a full line) or in.next() (reads a single word).
- **Integers:** in.nextInt(), in.nextShort(), in.nextByte(), in.nextLong().
- **Decimals:** in.nextDouble(), in.nextFloat().
- **Booleans:** in.nextBoolean().

Note: JAVA does not have a nextChar() method. To read a single character, you must read the input as a String and then extract the first character:

```
char carac = in.next().charAt(0);
```

we can also read array elements using the Scanner.

```
int[] a = new int[5];

for (int i = 0; i < a.length; i++) {

    System.out.print("Enter value " + i + ": ");

    a[i] = in.nextInt(); // Fills the array index with user input

}
```

Output is managed by the System.out object. Unlike the Scanner, System.out is automatically available; we don't need to import a library or create a new object.

We can use println method to display text or variables and then moves the cursor to a new line. Using the + operator allow to combine (concatenate) strings and variables.

```
int a = 10, b = 20;
```

```
System.out.println("a: " + a + " b: " + b);
```

Borrowed from the C language, printf allows also for precise formatting, such as limiting the number of decimal places for real numbers.

- **%d**: Placeholder for integers.
- **%.2f**: Placeholder for floating-point numbers (rounded to 2 decimal places).
- **\n**: Command to start a new line.

```
double f = 10.2555;
```

```
System.out.printf("Formatted: %.2f\n", f); // Output: 10.26
```

```
System.out.println("Actual: " + f); // Output: 10.2555
```

7. Control structures in JAVA

Control structures determine the "flow" of your program, allowing it to make decisions or repeat actions based on specific conditions. Without these, a program would simply execute line-by-line in a fixed, linear order.

JAVA uses standard if-else logic and switch cases to branch code execution based on boolean evaluations.

The if-else statement is the fundamental tool for handling logic. If the condition inside the parentheses is true, the first block executes; otherwise, the else block runs.

```
if (yourSales >= target) {  
    performance = "Satisfactory";  
    bonus = 100 + 0.01 * (yourSales - target);  
} else {  
    performance = "Unsatisfactory";  
    bonus = 0;  
}
```

For more complex scenarios, the **else if** ladder is used. The **braces { }** are always used to define the scope of your instructions clearly.

```
if (yourSales >= 2 * target) {  
    performance = "Excellent";  
    bonus = 1000;  
}  
else if (yourSales >= 1.5 * target) {  
    performance = "Fine";  
    bonus = 500;  
}  
else if (yourSales >= target) {  
    performance = "Satisfactory";  
    bonus = 100;  
}  
else {  
    System.out.println("You're fired");  
}
```

The switch statement is a cleaner alternative to multiple if statements when comparing a single variable against several constant values.

```
Scanner in = new Scanner(System.in);  
  
System.out.print("Select an option (1, 2, 3, 4): ");  
  
int choice = in.nextInt();  
  
switch (choice) {  
    case 1:  
        // Logic for option 1  
        break; // Exits the switch  
    case 2:  
        // Logic for option 2
```

```
        break;

    default:

        // Handles invalid input

        System.out.println("Bad input");

        break;

}
```

JAVA uses also Loops to repeat a block of code until a specific condition is no longer met. For this JAVA uses three kind of loops (while, do/while and for).

Thes while loop checks the condition **before** each iteration. If the condition is false initially, the code inside may never run.

```
while (balance < goal) {

    balance += payment;

    double interest = balance * interestRate / 100;

    balance += interest;

    years++;

}

System.out.println(years + " years.");
```

The do/while loop is used when the code **must execute at least once**. The condition is checked at the end of the block.

```
do {

    balance += payment;

    year++;

    System.out.printf("After year %d, your balance is %, .2f%n",
year, balance);

    System.out.print("Ready to retire? (Y/N)");

    input = in.next();

}
```

```
} while (input.equals("N"));
```

The for loop is the most concise way to repeat code when you know the number of iterations in advance. It combines initialization, condition, and update in one line.

```
for (int i = 10; i > 0; i--) {  
    System.out.println("Counting down ... " + i);  
}
```

JAVA provides also two keywords to manually alter loop behavior:

- **break:** Immediately exits the loop entirely, regardless of the condition.

```
while (years <= 100) {  
    balance += payment;  
    double interest = balance * interestRate / 100;  
    balance += interest;  
    if (balance >= goal)  
        break;  
    years++;  
}
```

- **continue:** Skips the remaining code in the current iteration and jumps back to the **start (header)** of the loop for the next cycle.

```
while (sum < goal) {  
    System.out.print("Enter a number: "); n = in.nextInt();  
    if (n < 0)  
        continue;  
    sum += n;  
    // not executed when n < 0  
}
```

Note : JAVA code must be translated into bytecode before it can be executed by the JAVA Virtual Machine (JVM).

Compile: Use the JAVAc tool on your source file.

```
JAVAc yourprogram.JAVA (This creates a yourprogram.class file)
```

Run: Use the JAVA tool on the class containing the main method.

```
JAVA yourmainclass
```

Chapter 4 : Introduction to OOP modeling

In every professional field, modeling serves as a vital bridge between a concept and its creation. Just as an architect drafts blueprints for a home or an engineer simulates an aircraft's aerodynamics, software developers use models to simplify reality. Modeling allows us to strip away unnecessary details and focus on the essential structure of a system before a single line of code is written.

Today, modeling is a cornerstone of successful IT projects. It addresses the reality that complex systems are impossible to understand in their entirety at once. By creating a model, we achieve four critical goals: we **visualize** the system's current or intended state, **specify** its precise structure and behavior, provide a **template** to guide its physical construction, and establish a permanent **documentation** of our design decisions. Ultimately, modeling is the art of managing complexity, ensuring that our software remains scalable, maintainable, and aligned with real-world requirements.

1. Principles of modeling

To build effective software, modeling must follow a set of fundamental principles that ensure the design is both practical and accurate.

- **Influence on Approach:** The choice of which models to create significantly shapes how we perceive a problem and, ultimately, how we formulate its solution. A well-chosen model guides the developer toward the most efficient architectural path.
- **Varying Levels of Precision:** Models are not "one size fits all." They can be expressed at different levels of detail depending on who is using them (a high-level overview for stakeholders or a high-precision blueprint for developers).
- **Real-World Grounding:** The most successful models are those directly connected to reality. A model should accurately reflect the actual requirements and constraints of the physical or business environment it represents.
- **Multiple Viewpoints:** No single model can capture the entire scope of a complex, non-trivial system. Instead, the best approach involves a small set of nearly independent

models that provide different perspectives, ensuring all aspects of the system's behavior and structure are covered.

2. Object oriented modeling

The evolution of software development has shifted from a functional focus to a structural one, prioritizing objects that mirror the real world. Historically, software was built around **procedures and functions**. While effective for small tasks, this approach struggles with scaling. As systems grow and requirements change, algorithmic code often becomes tangled and difficult to maintain.

Today, the fundamental building blocks are **objects and classes** where a **class** is a blueprint or description of a set of similar objects and an **object** is a concrete "thing" derived from the problem or solution.

Every object possesses three critical characteristics:

- **Identity:** A way to distinguish it from other objects (e.g., a unique name).
- **State:** The data or properties associated with it.
- **Behavior:** The actions it can perform or how it communicates with others.

For example, in a billing system, objects appear at every level. The **User Interface** contains buttons and menus; the **Business Layer** handles transactions and rules; the **Database** manages tables representing customers and products.

In the early days of OOP, developers used competing methods (OMT by Rumbaugh, OOSE by Jacobson, and Booch by Grady Booch). These three pioneers eventually joined forces to create the **Unified Modeling Language (UML)**.

UML is not a "method" (a step-by-step process), but a **modeling language** (a standardized vocabulary and set of rules used to represent software plans). Because it is independent of any specific programming language, it has become the global standard for architects and designers.

UML is defined by its ability to handle four key tasks in software development:

- **Visualization:** It provides a graphical representation of the system, making it easier for developers to understand and maintain complex logic.

- **Specification:** It allows for the creation of precise, complete, and unambiguous models that leave no room for guesswork.
- **Implementation:** UML models are designed to be easily "translated" into code for object-oriented languages like **JAVA, C++, or C#**.
- **Documentation:** It serves as a detailed record of the system's architecture, requirements, and testing procedures.

While primarily intended for software, UML's versatility has made it essential in various complex fields like **Finance** (Banking systems and financial services), **Infrastructure:** (Telecommunications, transportation, and web services), **Critical Systems:** (Aerospace, defense, and medical electronics) and **Research:** (Science and large-scale business information systems).

3. UML overview

To master UML, a conceptual model of the language must be formed. This process involves learning three fundamental pillars: the **basic building blocks**, the **rules** for their arrangement, and the **common mechanisms** applied throughout the system.

A. The Basic Building Blocks of UML

The UML vocabulary is organized into three primary categories: **Things, Relationships,** and **Diagrams**.

a. Things: The Core Abstractions

"Things" represent the fundamental elements within a model. These are categorized into four types based on their specific role:

▪ Structural Things (The Nouns)

These represent the static components of a model, describing physical or conceptual parts. Collectively referred to as **Classifiers**, they define the "what" of the system. An example of these things are Class, Interface, Collaboration, Use Case, Active Class, Component, Artifact, and Node.

▪ Behavioral Things (The Verbs)

These constitute the dynamic portions of a UML model. They represent behavior over time and space, acting as the operational "verbs." Interactions (sequences of messages), State Machines (lifecycle transitions), and Activities (flow of tasks) are example of these things.

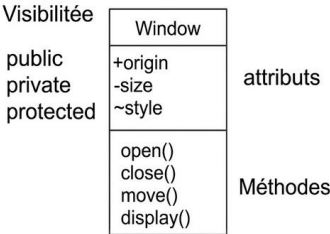
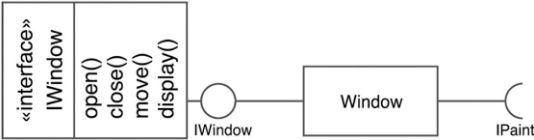
▪ Grouping Things (The Organizational Folders)

Grouping things serve as the organizational framework for a model. They allow complex systems to be decomposed into smaller, manageable parts. The **Package** is the primary grouping mechanism used to organize other elements into folders or modules.

▪ Annotational Things (The Explanatory Notes)

These are the descriptive parts of the UML model. They provide space for comments, clarifications, or highlights regarding any other element in the design. The **Note** is the only annotational thing. It is represented as a rectangle with a folded corner.

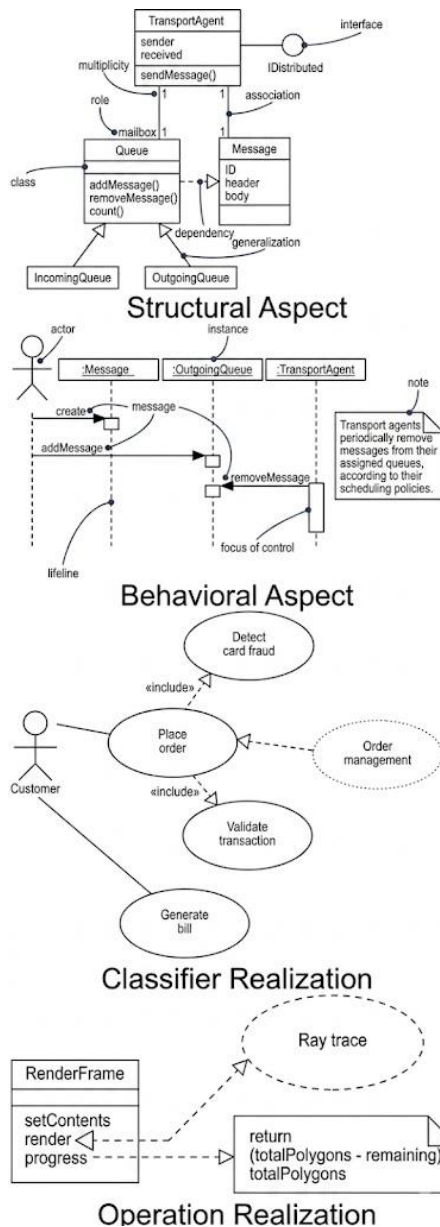
The following table summarizes the symbols with a brief explanation of the four types of things.

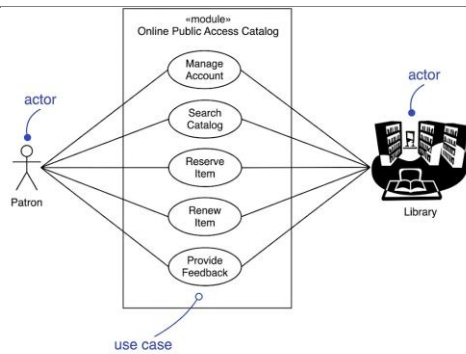
Figure (Symbol)	Description
 The diagram shows a class named 'Window'. To its left, a list of visibility modifiers: 'Visibilitée', 'public', 'private', and 'protected'. The class box is divided into three sections: the top section contains the class name 'Window'; the middle section, labeled 'attributs' to its right, contains '+origin', '-size', and '~style'; the bottom section, labeled 'Méthodes' to its right, contains 'open()', 'close()', 'move()', and 'display()'.	A class is a set of objects sharing attributes and operations. Represented by a rectangle with three sections (Name, Attributes, Methods).
 The diagram shows an interface 'IWindow' and a class 'Window'. The interface is represented by a rectangle with '«interface»' in the top-left corner, 'IWindow' in the top-right corner, and methods 'open()', 'close()', 'move()', and 'display()' in the bottom section. The class 'Window' is a simple rectangle. A solid line connects a small circle on the interface to a semi-circle on the class, representing an implementation relationship. The interface is also labeled 'IWindow' and the class is labeled 'IPaint'.	An interface is a set of operations that other elements must implement. Represented by a circle (lollipop) or a class with the «interface» keyword.



A collaboration is a combination of elements working together to provide cooperative behavior. Represented by a **dotted ellipse**.

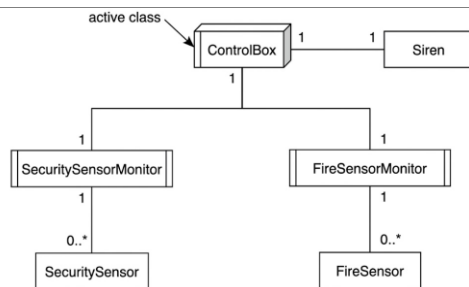
A collaboration is also the specification of how an element, such as a classifier (including a class, interface, component, node, or use case) or an operation, is realized by a set of classifiers and associations. A collaboration has a structural and a behavioral aspect. It is represented by a dotted ellipse. The classifier implementation (realisation d'un classeur) figure models a set of use cases for a credit card validation system. The operation implementation figure models the operations of an active class.



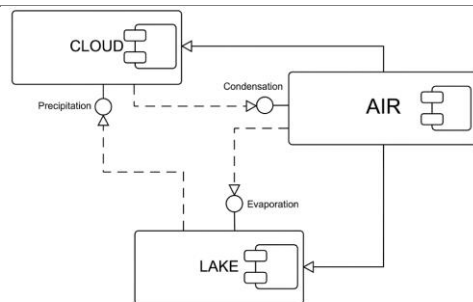


Use Case Diagram of an Online Library.

A use case is a sequence of actions a system performs for an actor to yield a result of value. Represented by an **ellipse**.

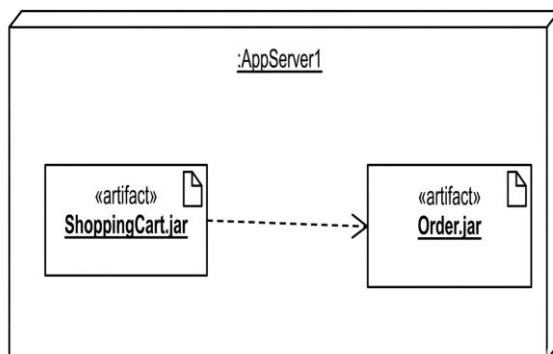


An active class is a class whose objects own a process or thread. Represented as a rectangle with **double vertical lines** on the sides.

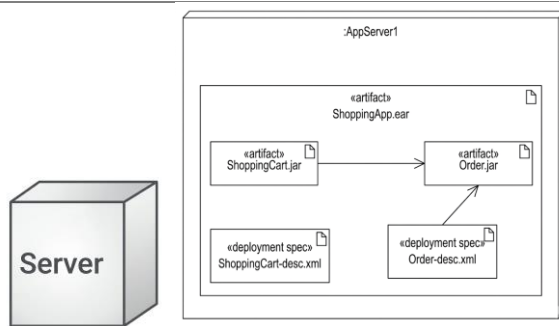


A component is a modular part of a system for which UML diagrams are created. Shown as a rectangle with a **special icon** in the corner.

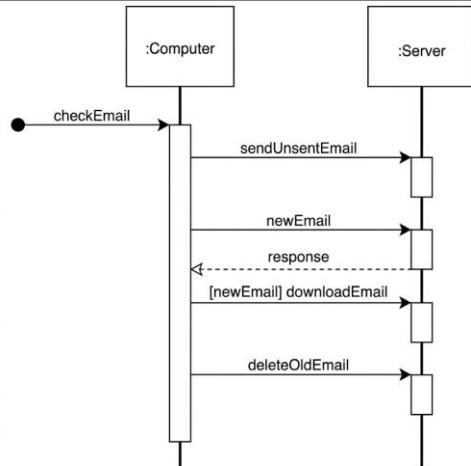
«artifact»
window.dll



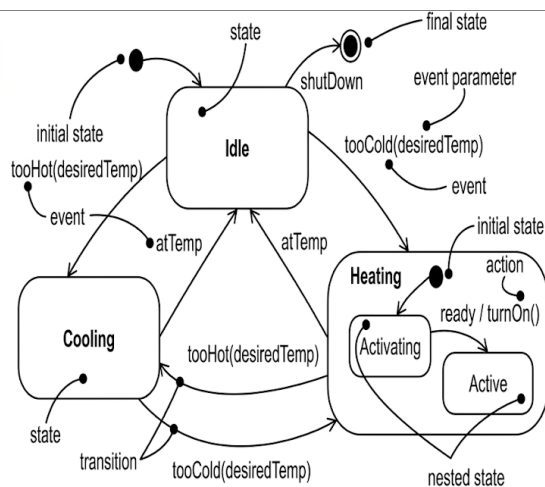
An artifact is a physical and replaceable part of a system (bits, scripts, executables). Represented as a rectangle with the «artifact» keyword.



A node is a physical runtime resource (like a server) with memory and processing power. Represented as a **3D cube**.



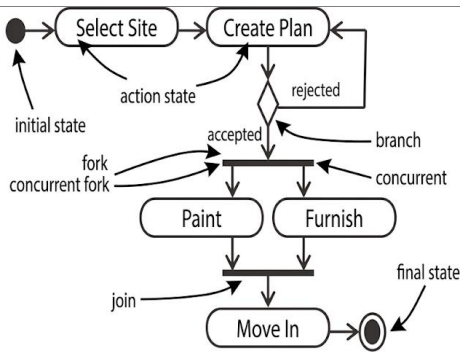
An interaction is a behavior comprising messages exchanged between objects to achieve a goal. Represented by a **directed line** (arrow).



A stat machine specifies the sequences of states an object follows during its existence. States are shown as **rounded rectangles**.

State: rounded rectangle.

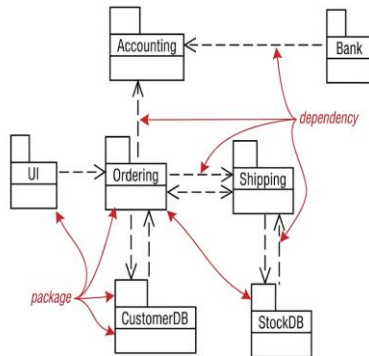
Event: text.



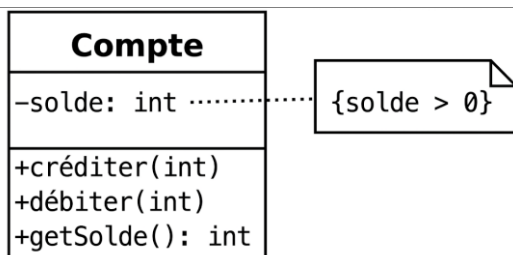
Activity : rounded rectangle.

An activity specifies a sequence of steps in a computational process. Steps are called actions and shown as **rounded rectangles**.

Package diagram



A package is a mechanism for organizing elements into groups. Represented as a **tabbed folder**.



A note is a symbol to express constraints or comments attached to elements. Represented as a rectangle with a **dog-eared corner**.

b. Relationships

Understanding the connections between model elements is essential for building a coherent system. UML offers four primary relationships :

- **Dependency**

A **dependency** is a "using" relationship. It indicates that one element (the client) requires another element (the supplier) for its implementation or specification. If the supplier

changes, the client may be affected. A dependency is represented using **dashed line** with an open arrowhead pointing toward the supplier.

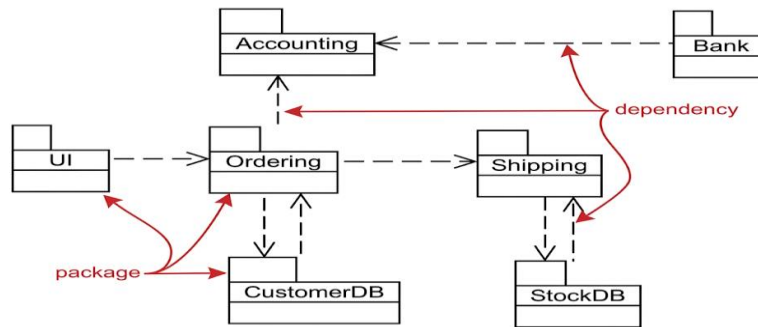


Figure 18. Example of dependency relationships

- **Association**

An association is a structural relationship between classes that describes a set of links, a link being a connection between objects that are instances of the classes. An association is represented as a solid, possibly directed, line, sometimes including a label and often containing other adornments, such as **multiplicity** and **role names**.

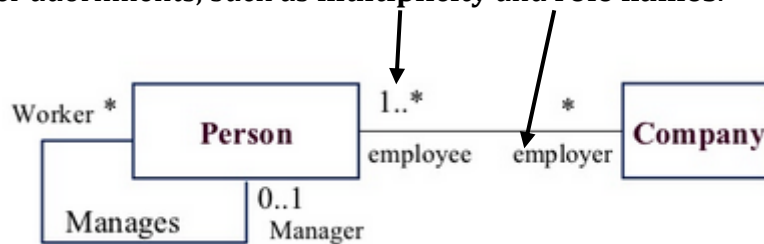


Figure 19. Example of association relationships

Multiplicity is used in UML associations to specify how many instances of one class can be linked to a single instance of another class.

Adornment	Semantics (Meaning)
0..1	Zero or 1
1	Exactly 1
0..* or *	Zero or more
1..*	1 or more
1..6	1 to 6

1..3, 7..10, 15, 19..* 1 to 3 or 7 to 10 or 15 exactly or 19 to many

In UML, we often need to model a **"whole/part"** relationship where one class (the whole) is made up of smaller entities (the parts). This is called **aggregation**. It represents an association relationship where the part can exist independently of the whole. It is drawn as a solid association line with an **unfilled diamond** at the "whole" end.

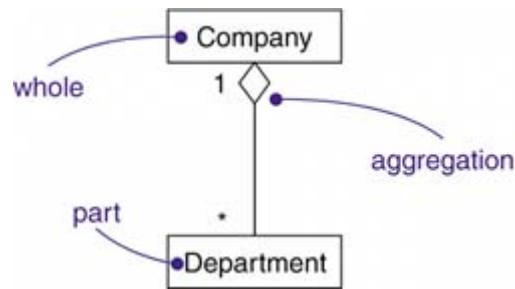


Figure 20. Example of an aggregation relationships

Composition is a strong form of the **whole/part** relationship, often referred to as the **"death relationship."** Unlike aggregation, the parts in a composition are strictly bound to the lifecycle of the whole. If the "whole" object is deleted, all its associated "part" objects are destroyed simultaneously. A part is owned exclusively by one whole at a time and cannot be shared between multiple instances. A composition is represented by a solid association line with a **filled (black) diamond** at the "whole" end.

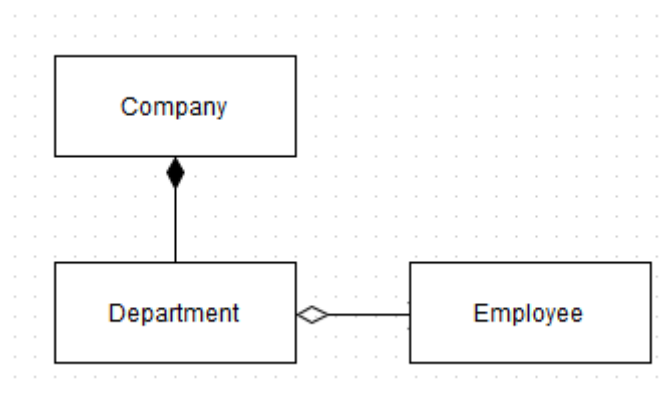


Figure 21. Example of composition relationships

Hierarchically, these relationships are nested: **composition** is a specialized form of **aggregation**, and **aggregation** is itself a specialized form of **association**.

- **Generalization**

Generalization is a relationship between a general element (parent) and a more specific element (child). This is the standard UML term for **inheritance**. It is represented using a **solid line with a hollow arrowhead** pointing toward the parent. The child inherits all attributes and behaviors of the parent while having the ability to add its own unique features.

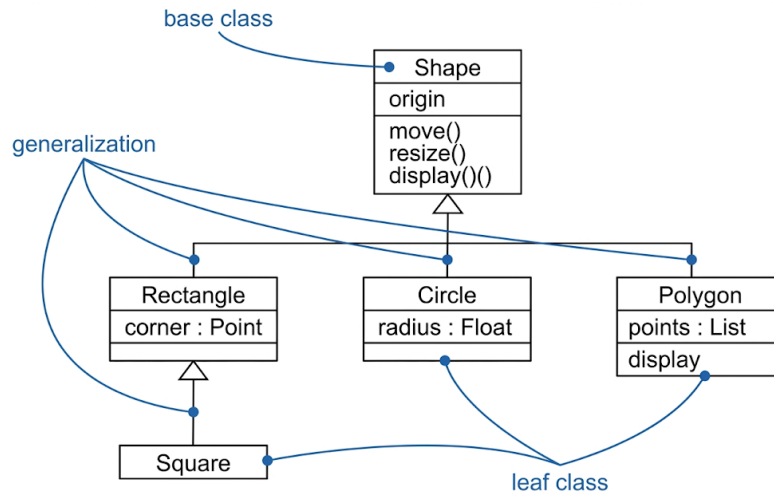


Figure 22. Example of generalization relationships

- **Realization**

A realization is a semantic relationship where one element (the implementation) carries out a "contract" defined by another element (the specification). It is represented using a **dashed line with a hollow arrowhead**. We can find a realization between an **Interface** and the **Class** that implements its operations or between a **Use Case** and the **collaboration** that describes how that use case is actually performed.

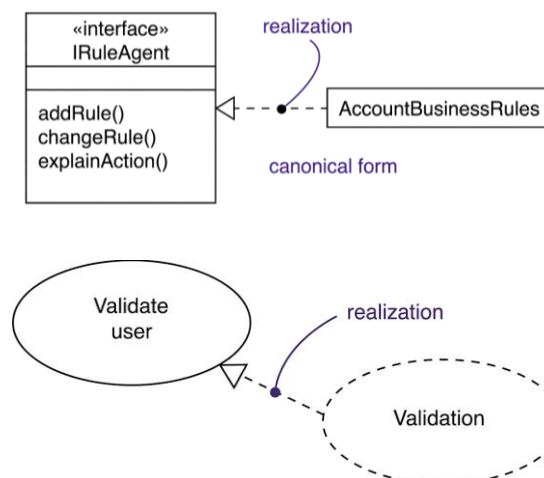


Figure 23. Example of realization relationships

c. Diagrams

A **diagram** is a graphical representation of system elements, typically structured as a connected graph of **nodes** (representing things) and **edges** (representing relationships). In UML, diagrams serve as specific projections or "views" of a system's model.

UML consists of **fourteen** unique diagram types, which are formally organized into three primary categories: **Structural**, **Behavioral**, and **Model Management**.

Diagram Type	Category	Main Usage
Class Diagram	Structure	Models the static structure of the system (classes, attributes, operations)
Object Diagram	Structure	Shows snapshots of objects and their relationships at a specific moment
Component Diagram	Structure	Describes the organization and dependencies among software components
Composite Structure Diagram	Structure	Shows internal structure of a class and interactions between its parts
Package Diagram	Management	Organizes elements into packages; shows dependencies between them
Deployment Diagram	Structure	Models the physical deployment of software on hardware nodes
Profile Diagram	Structure	Customizes UML models using stereotypes and profiles
Use Case Diagram	Behavior	Represents functional requirements and interactions between users and system
Activity Diagram	Behavior	Describes workflows and sequence of activities (flowchart-like)
State Machine Diagram	Behavior	Models state changes of an object in response to events
Sequence Diagram	Behavior	Shows time-ordered interaction between objects or components
Communication Diagram	Behavior	Displays interactions focusing on object relationships
Interaction Overview Diagram	Behavior	Combines activity and sequence diagrams to show control flow
Timing Diagram	Behavior	Visualizes object behavior over time with timing constraints

B. Rules

Just as any natural language requires grammar to be understood, UML relies on a set of **Rules** to ensure that models are "well-formed"—meaning they are semantically consistent, harmonious, and logically sound. UML dispose of five essential rules for maintaining coherence in UML models:

- **Names**

Every entity in a UML model must have a name for identification and referencing. This includes elements (classes, objects), relationships, and the diagrams themselves. A name must refer to the same element throughout its context. If a class is named Customer, every reference to Customer must point to that specific definition. Names should be meaningful and follow standard conventions to prevent ambiguity.

- **Scope**

Scope defines the specific context or "boundary" within which a name has a particular meaning. The same name can be used in different parts of a system to mean different things, provided their scopes do not overlap. For this, UML requires clear boundaries (such as within a Class or a Package) so that a variable like status in one module isn't confused with a status variable in another.

- **Visibility**

Visibility determines the "reach" or access level of an element (like an attribute or operation) from other parts of the system. It mirrors access modifiers in programming. There are four standard visibility levels in UML:

- **Public (+)**: The element is accessible from any other class.
- **Private (-)**: The element is only accessible within the class itself.
- **Protected (#)**: The element is accessible within the class and its subclasses.
- **Package (~)**: The element is accessible to all classes in the same package.

- **Integrity**

Integrity ensures that relationships between elements remain logically and semantically correct. It ensures that associations, dependencies, and constraints (like multiplicity) are respected. For example, if a rule states a Customer can have 1..* (one-to-many) Orders, the integrity rule ensures this relationship is maintained across all design and execution phases.

- **Execution**

Execution involves simulating or verifying the dynamic behavior defined in the model. This applies to behavioral diagrams like **Sequence**, **State Machine**, and **Activity**

diagrams. It checks how the system behaves over time (ensuring that when a user triggers an action, the objects interact, change states, and execute operations exactly as intended).

c. Common mechanisms

Just as architectural styles define the structure and appearance of a building, UML uses **mechanisms** to define how models are structured, extended, and customized. These mechanisms provide the necessary depth and flexibility to adapt UML to various domains.

There are four fundamental mechanisms in UML:

- Specification

UML is not just a collection of drawings. Every graphical element is backed by a **specification** (a detailed textual definition of its syntax and semantics). For example, behind a simple class rectangle lies a specification that defines its full set of attributes, operations, and behaviors in a way that tools can interpret.

- Adornments

Adornments are graphical or textual markers added to a basic element to provide more detail. They prevent diagrams from becoming overly complex by keeping basic symbols simple while allowing for "layers" of detail. For example, adding a + or - to an attribute to show visibility, or using *italics* to indicate an abstract class.

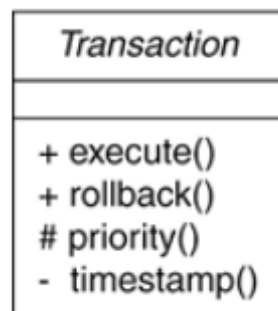


Figure 24. Example of adornment by showing visibility

- Common Divisions

UML encourages partitioning a system into manageable parts to distinguish between different logical layers:

- **Classes vs. Objects:** Separates the **blueprint** (Design-time) from the **actual instance** (Runtime). For example, Car is the class, and myCar:Car is the specific object.

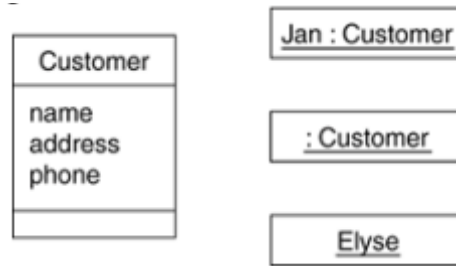


Figure 24. Example of division by separating classes from objects

- **Interface vs. Implementation:** Distinguishes **what** an element does (its contract) from **how** it does it (the logic). This promotes encapsulation and interchangeability.

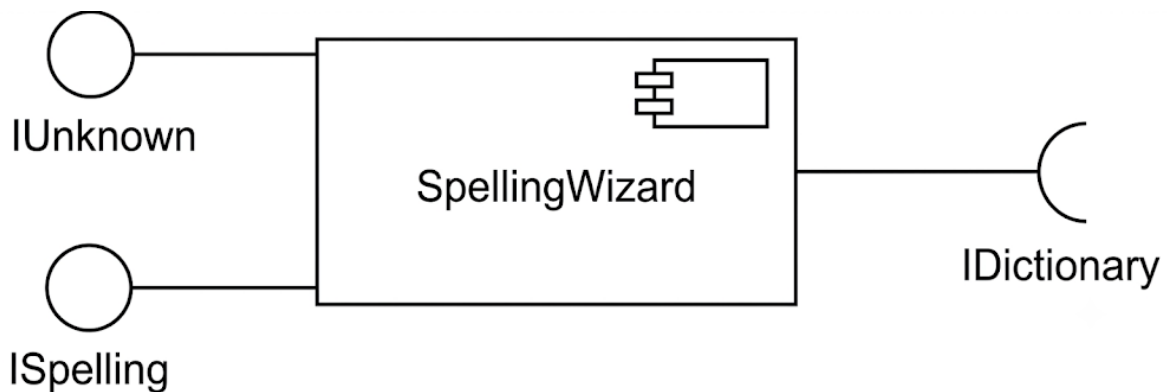


Figure 25. Example of division by separating interface from implementation

- **Role vs. Type:** Separates the **definition** of an object from the **function** it performs in a specific scenario. An object of the type Person might play the role of Customer in one context and Employee in another.

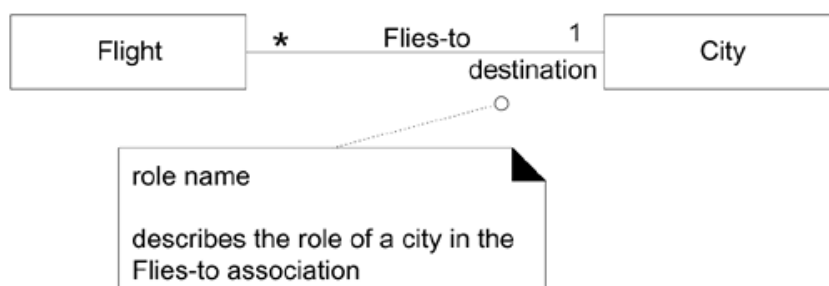


Figure 26. Example of division by separating role from type

- Extensibility Mechanisms

UML is designed as an "open" language. Because a single, fixed vocabulary cannot capture the unique requirements of every industry (such as aerospace, healthcare, or web development). UML provides four **Extensibility Mechanisms**. These allow you to adapt and broaden the language's capabilities in a controlled and standardized way.

- **Stereotypes**

Stereotypes are used to expand the UML vocabulary by creating new types of building blocks based on existing ones. They are typically denoted by double guillemets (e.g., «interface»). They allow you to categorize elements specifically for your problem domain. For example, you might mark a standard class as a «sensor» or «database» to clarify its specific role in the system.

- **Tagged Values**

Tagged values allow you to add custom properties to a model element's specification. They are usually written as a key-value pair within curly braces, such as {version = 2.1}. While stereotypes define a new "kind" of element, tagged values define new data for that element. This is useful for tracking metadata like author names, deadlines, or hardware requirements that aren't part of the standard UML definition.

- **Constraints**

Constraints are used to extend the semantics (meaning) of a base element by adding new rules or modifying existing ones. They are written as text strings within curly braces. They define logic that the model must follow. For instance, you could add a constraint to an attribute to ensure {value > 0} or to a relationship to specify that it is {ordered}.

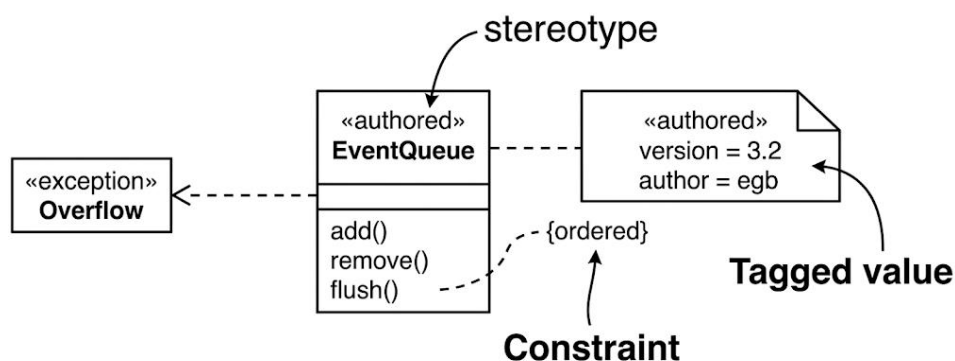


Figure 27. Example of stereotype, constraint and tagged value

- **Profiles**

A Profile is a high-level organization mechanism used to bundle stereotypes, tagged values, and constraints into a single package tailored for a specific domain or platform. Instead of applying individual extensions one by one, a Profile allows you to apply a whole "dialect" of UML at once.

The following schema resume the UML concepts:

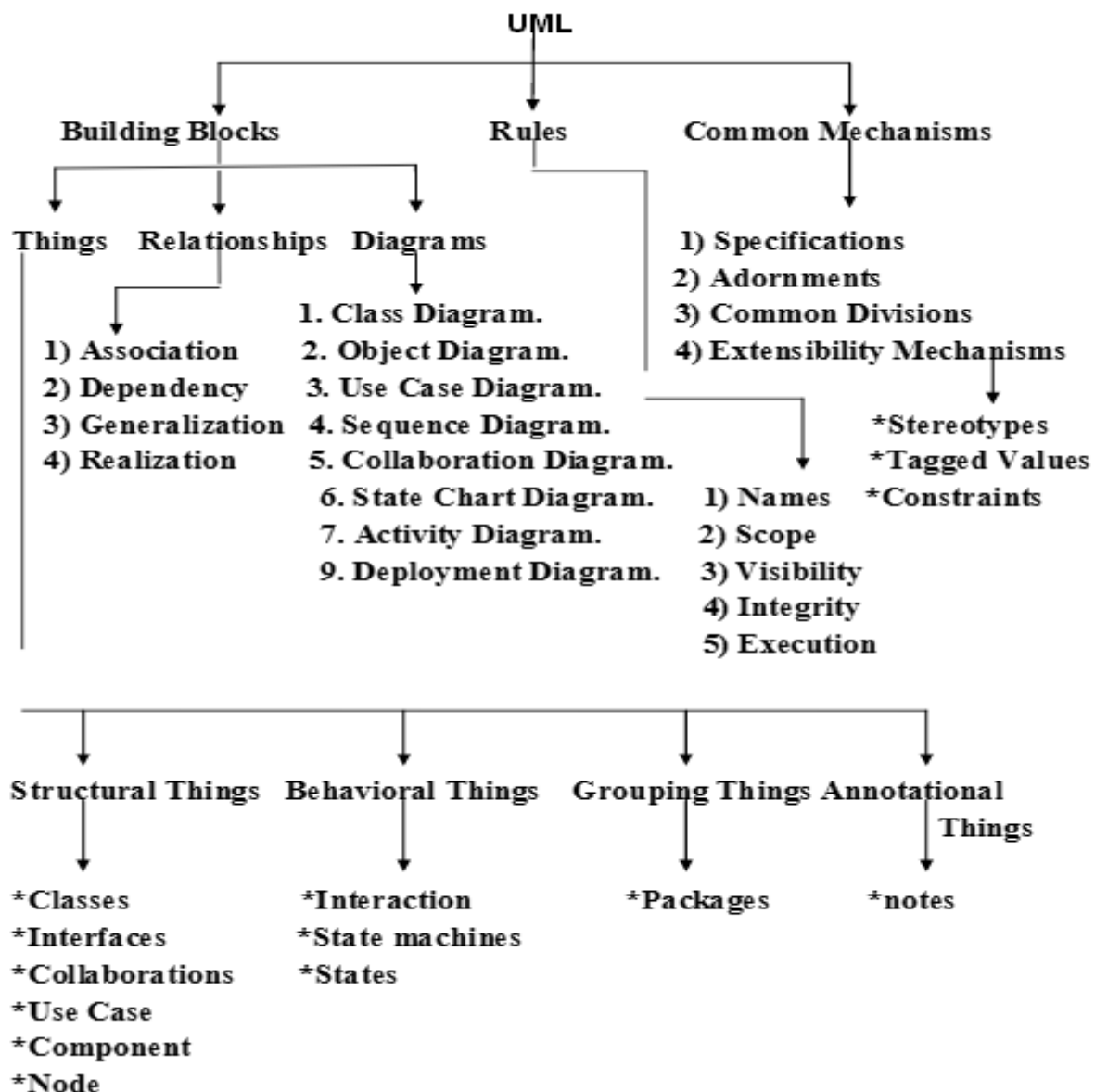


Figure 28. General schema of UML

Chapter 5: Behavioral Modeling: Use Case Diagrams

In every professional field, modeling serves as a vital bridge between a concept and its creation. While we have previously explored the static "blueprints" of a system's structure, we now turn our focus to the dynamic forces that drive it. Just as an aeronautical engineer must model not only the shape of a wing but also how it reacts to changing wind speeds, software developers use **Behavioral Modeling** to capture how a system functions in the real world.

Today, behavioral modeling—starting with the **Use Case Diagram**—is a cornerstone of successful IT projects. It addresses the reality that a system's internal logic is meaningless if it does not align with the user's needs. By creating a Use Case model, we achieve four critical goals: we **visualize** the interactions between the system and its environment, **specify** the functional requirements from an external perspective, **provide a template** for testing and validation, and **establish a clear scope** that manages stakeholder expectations.

Ultimately, Use Case modeling is the art of defining the system's purpose, ensuring that the final product remains useful, intuitive, and perfectly aligned with the goals of those who will use it.

1. Use case Definition

The **Use Case Diagram** is a unique graphical model used to capture how external entities expect to interact with a system. Unlike other diagrams that focus on internal logic, this diagram describes the users involved, the services they require, and the services they must provide to the system.

The use case diagram is a cornerstone of the **Object-Oriented Software Engineering (OOSE)** method, originally known as **Objectory** (Object Factory). Published in 1992 by **Ivar Jacobson**, the technique proved highly effective for large-scale projects at companies like Ericsson and HP.

Its success caught the attention of other leading methodologists, such as **Grady Booch** and **James Rumbaugh**, whose methods lacked a dedicated way to document requirements from a user-centric perspective. Consequently, the use case diagram was formally integrated into UML 0.9 in June 1996.

A diagram alone is rarely sufficient. In professional practice, the diagram is part of a comprehensive three-tiered approach:

- **The Use Case Diagram:** Provides a high-level visual map of system boundaries and stakeholders.
- **The Use Case Narrative:** A detailed textual description addressing specific functional requirements.
- **The Use Case Scenarios:** Exploration of different execution paths (options) used to test requirements and provide a high-level test plan for future development.

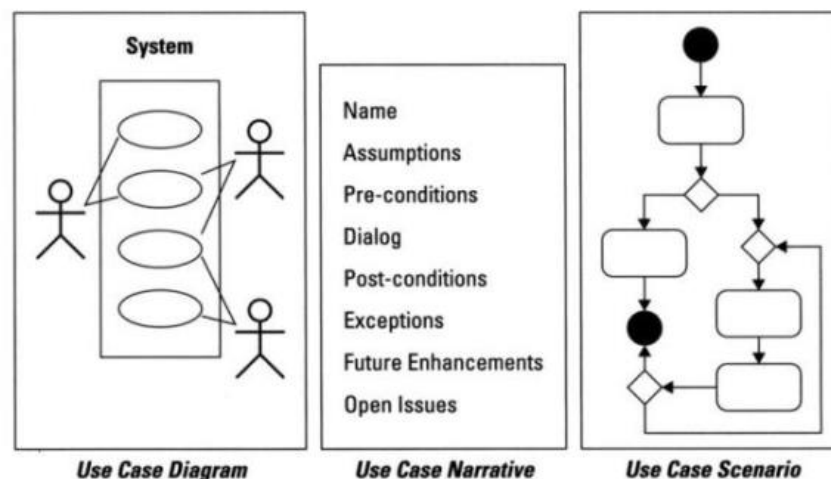


Figure 29. The use case diagram as a part of a comprehensive three-tiered modeling

The Use Case Diagram is composed of six essential elements that define the system's external behavior:

- **Actor:** Represents a role played by a person, system, device, or organization that interacts with the system.
- **Use Case:** Identifies a key system behavior or goal. Without this specific behavior, the system fails to meet the actor's requirements.

- **Association:** A line identifying the interaction between an actor and a use case, representing a "dialogue" that is later detailed in the narrative.
- **The «include» Relationship:** Represents a reusable use case that is **unconditionally** incorporated into the execution of a "base" use case. The base use case controls when this happens.
- **The «extend» Relationship:** Identifies a reusable use case that **conditionally** interrupts a base use case to add functionality. The decision logic resides within the extension itself.
- **Generalization:** An inheritance relationship where one actor or use case inherits the properties and behaviors of a more general one.

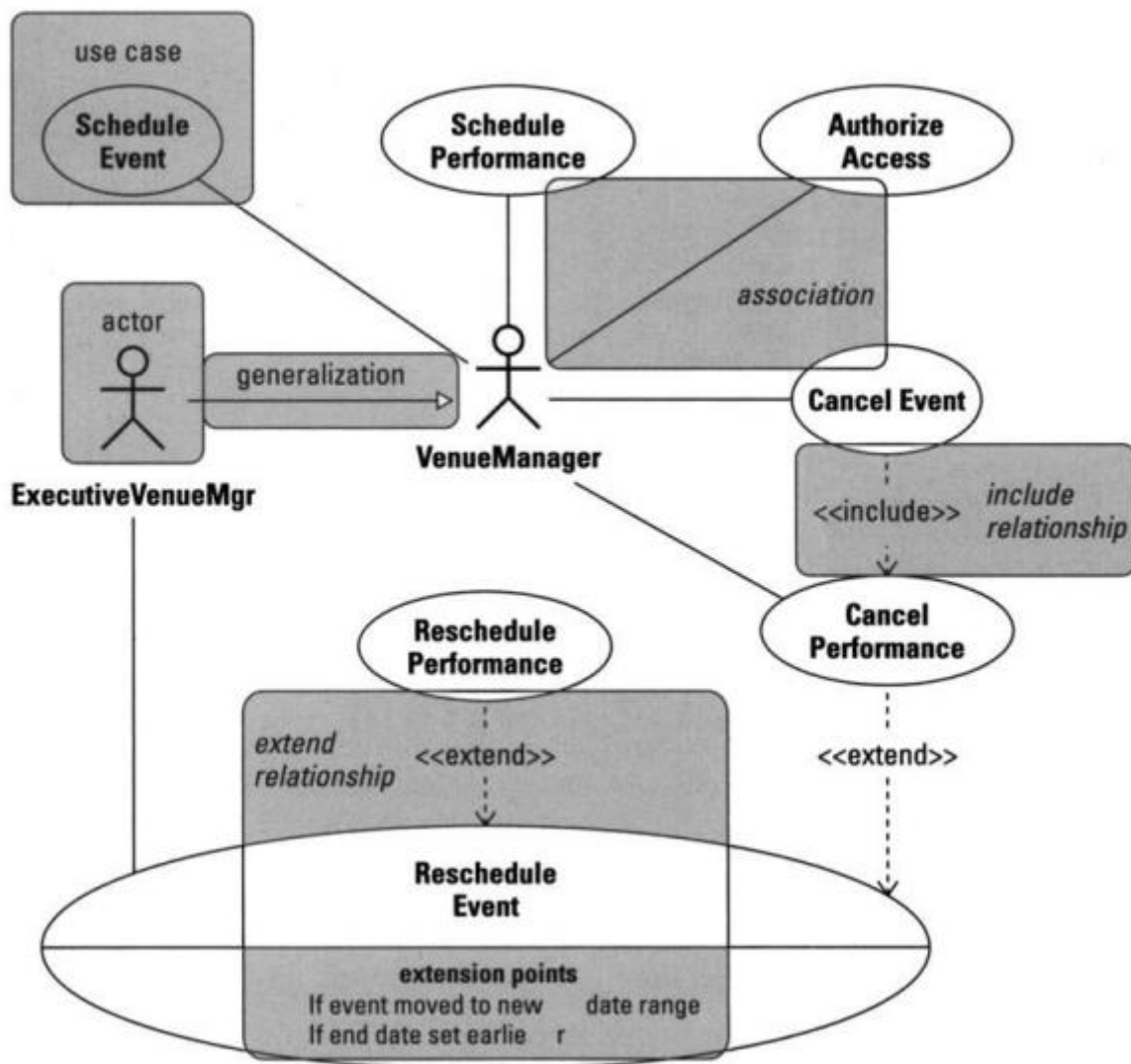


Figure 30. Use case diagram example of a venue management

2. Steps for Creating a Use Case Diagram

Developing a well-structured use case diagram is an iterative process that moves from identifying broad interactions to refining specific relationships.

Step 1: Define the System Context

The first goal is to establish the boundaries of your system and identify its primary participants.

- **Identify the Actors:** Determine who or what (people, external systems, or devices) interacts with the system and define their specific responsibilities.
- **Identify the Use Cases:** List the core system behaviors required to meet the actors' goals. Each use case should represent a discrete result or value produced by the system.

Step 2: Refine Actors and Use Cases

Evaluate your initial list to ensure clarity and appropriate granularity.

- **Merge or Split:** If two use cases perform nearly identical tasks, merge them. Conversely, if a single use case is too complex, split it into smaller, more manageable units.

Step 3: Identify «include» Relationships

Look for common sequences of behavior shared across multiple use cases.

- **Factor Out Commonality:** If several use cases require the same step (e.g., "User Authentication"), extract that behavior into its own use case and use an «include» relationship to avoid redundancy.

Step 4: Identify «extend» Relationships

Search for optional or exceptional behaviors that only occur under specific conditions.

- **Capture Conditional Logic:** Identify points where a "base" use case might be interrupted to perform an additional task (e.g., "Calculate Discount" during a "Checkout" process) using an «extend» relationship.

Step 5: Detect Generalizations

Analyze both actors and use cases to find shared properties or behaviors.

- **Apply Inheritance:** If multiple actors share the same basic rights but have specialized roles (e.g., a "Manager" who inherits from "Employee"), use **Generalization** to simplify the model and show the hierarchy.

3. Elements of the use case diagram

To build an effective Use Case diagram, you must understand the specific graphical symbols and the logic governing their interactions.

- The Actor

In UML, the term **Actor** refers to anything external that interacts with the system, including people, other software systems, hardware devices, or even external organizations. While often depicted as a "stick figure," the icon can vary depending on the actor's nature (e.g., a box for a system). Actors should represent **roles**, not specific individuals. For instance, "Store Manager" is a role, while "John Doe" is an instance.

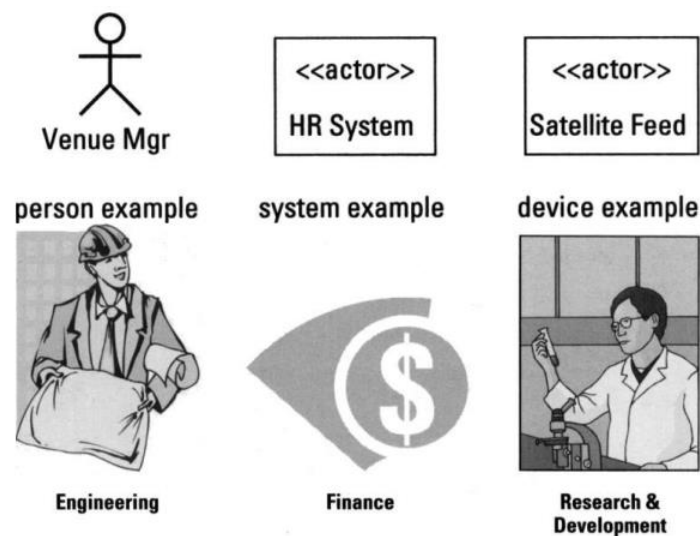


Figure 31. Actor representation in Use case diagram

The representation of an actor could use generalization, for example, specific roles like PrivateCustomer and CorporateCustomer inherit behaviors and properties from a more general Customer actor.

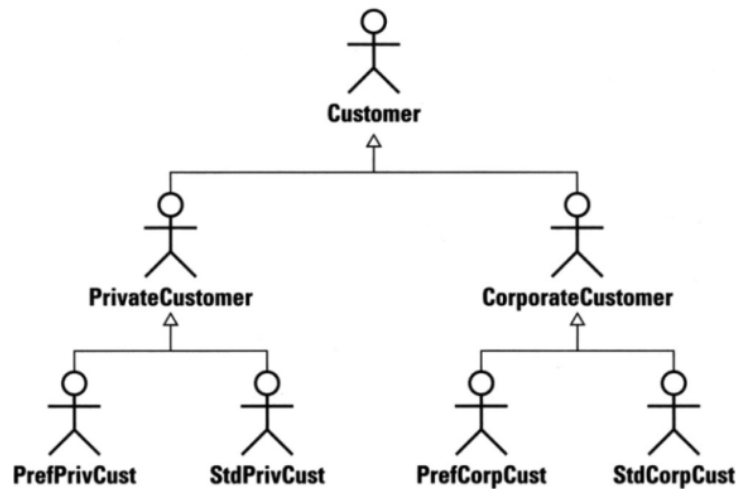


Figure 32. Actor generalization representation in use case diagram

- The Use Case

A Use Case represents a high-level system goal or a discrete service provided to an actor. Use cases are always named with a **verb phrase** (e.g., Create Agent Contract, Reschedule Program). The focus remains strictly on **what** the system achieves for the user, rather than the internal technical process of how it happens.

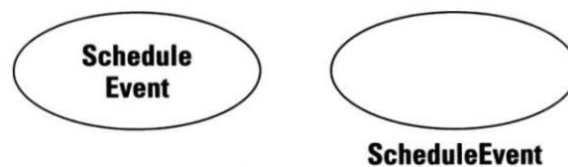


Figure 33. Use case representation

- The Association

The association is the simplest form of relationship, represented by a **solid line** connecting an actor to a use case. It indicates that a "dialogue" or communication occurs between the two.

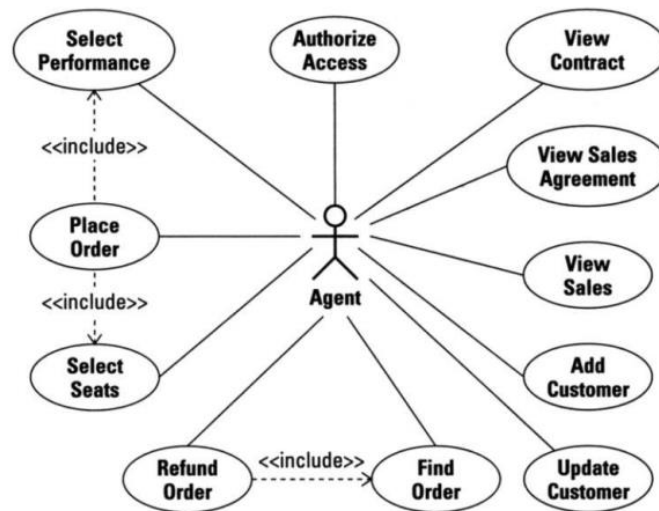


Figure 34. Association representation in use case diagram

- The «include» Relationship

This relationship is used to extract common behavior into a separate, reusable use case. It is governed by two strict constraints:

- **Dependency:** The calling (base) use case depends only on the *result* of the included behavior.
- **Unconditionality:** The calling use case **always** requires the execution of the included behavior to be complete.

The **«include» Relationship** is represented by a dashed arrow points **from** the base use case **to** the included use case.

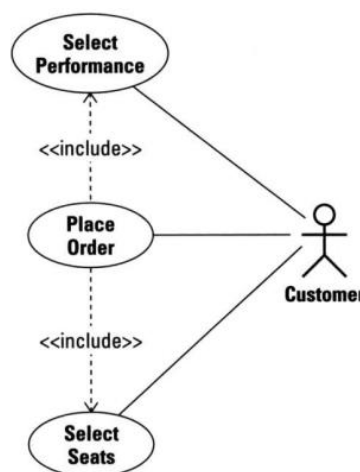


Figure 35. Representation of the «include» Relationship in use case diagram

- The «extend» Relationship

The «extend» relationship allows for optional or exceptional behavior to be added to a base use case without modifying its core logic. It requires four distinct parts:

- **Base Use Case:** The primary process that can be extended (e.g., Reschedule Event).
- **Extended Use Case:** The additional behavior that executes only if needed (e.g., Cancel Performance).
- **Extension Relationship:** A dashed arrow pointing **from** the extended use case back **to** the base use case.
- **Extension Points:** Specific locations within the base use case where a **condition** is checked. If the condition is met, the extension interrupts and executes.

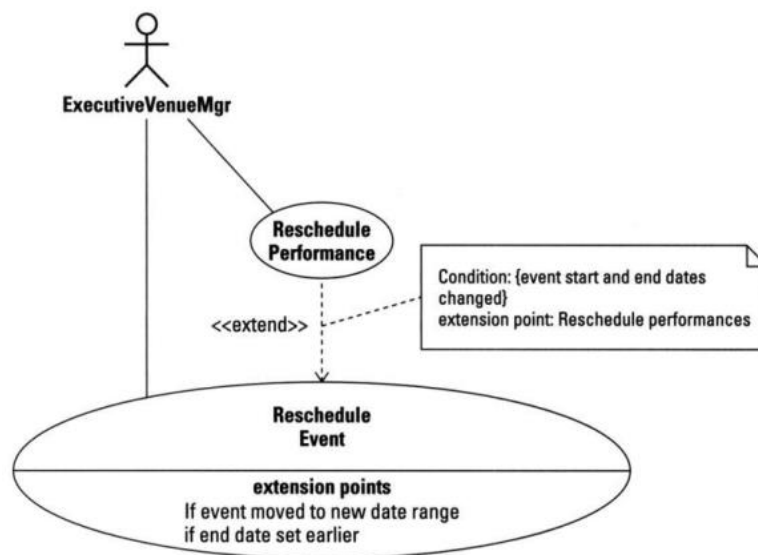


Figure 36. Representation of the «extend» Relationship in use case diagram

Chapter 6: Structural Modeling: Class Diagrams

If Use Case diagrams are the "voice" of the user's needs, **Class Diagrams** are the "skeleton" upon which the entire system is built. In structural modeling, we shift our perspective from the external behavior of the system to its internal composition. By defining the static structure, we establish the fundamental blueprint that developers use to translate abstract requirements into concrete, object-oriented code.

Today, class modeling is the primary tool for managing the internal complexity of IT projects. It addresses the challenge of organizing data and logic into a cohesive, manageable whole. By utilizing Class Diagrams, we achieve four critical objectives: we **visualize** the core entities and their properties, **specify** the exact relationships and multiplicities between them, **provide a direct template** for physical database and class construction, and **establish the technical documentation** necessary for long-term maintenance.

Ultimately, structural modeling ensures that the foundation of our software is robust and logical. It allows us to refine the system's "vocabulary"—the classes and their associations—ensuring that the final architecture is scalable, modular, and perfectly capable of supporting the behaviors defined in earlier stages of design.

1. Definition

The **Class Diagram** is the most widely used modeling tool in UML. Its importance is unique because it serves as the primary diagram for **code generation**, allowing developers to translate visual models directly into executable software. While other diagrams focus on behavior, the class diagram defines the system's "vocabulary"—the essential building blocks and relationships that remain constant regardless of the system's state. Class diagrams are versatile enough to model everything from aircraft navigation and medical equipment to factory automation and billing systems. Despite the variety of these applications, they all rely on a core set of elements: **classes** and **relationships**.

2. Modeling classes

A class is a blueprint or definition of an entity. It encapsulates the **characteristics** (states) and **operations** (behaviors) of that entity. It is crucial to distinguish between **class**, which is the general definition (e.g., Car), and an **object** which is a specific, unique instance of that class (e.g., myRedSedan). In UML, a class is represented as a rectangle divided into three compartments: the **Name**, the **Attributes**, and the **Operations**.

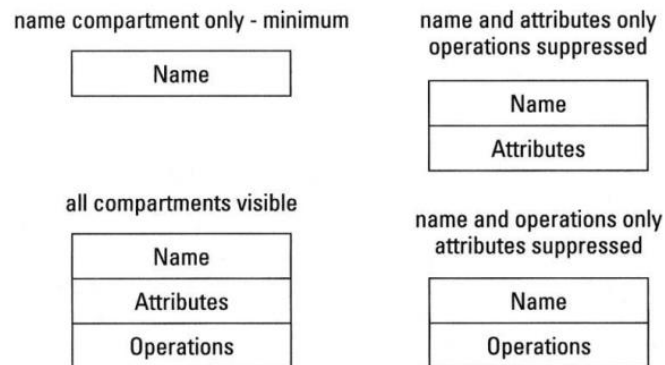


Figure 37. Different representations of a class

- The Class Name

The name identifies the class and must be a precise, concise descriptor. It must avoid including attributes in the name. For example, use Employee instead of ExemptEmployee, as "Exempt" is a state/attribute, not a distinct type of entity. If a class belongs to a specific package, it can be identified using the format Package_Name::Class_Name.

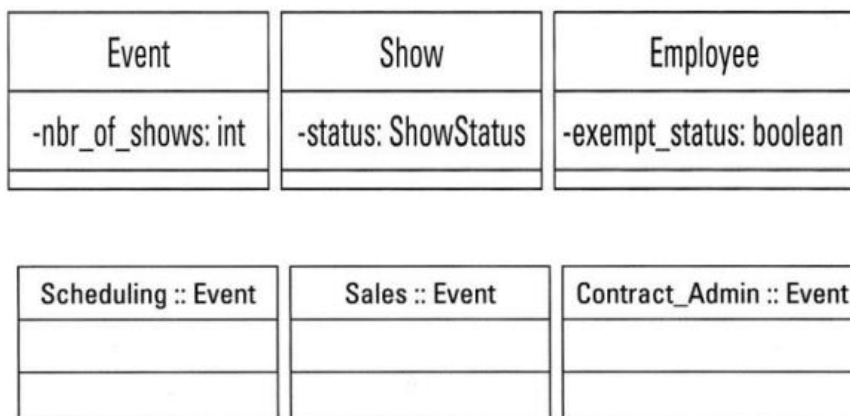


Figure 38. Example of representation of a class name

UML allows also the use of stereotypes to define specialized types of classes:

- **«entity»**: Describes objects that form the core subject of the system (e.g., Venue, Seat).

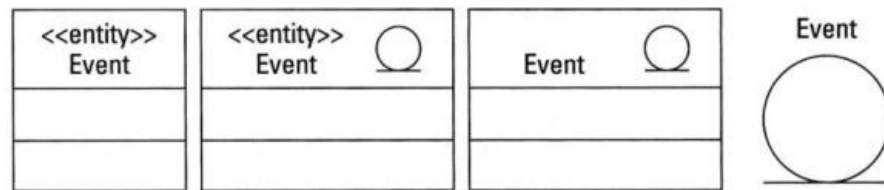


Figure 39. Example of representation of entity class

- **«control»**: Describes objects that manage application logic (e.g., Scheduling).

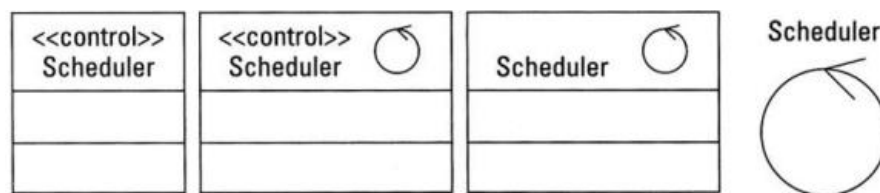


Figure 40. Example of representation of control class

- **«abstract»**: Represents a class that cannot be instantiated on its own (often written in *italics*).

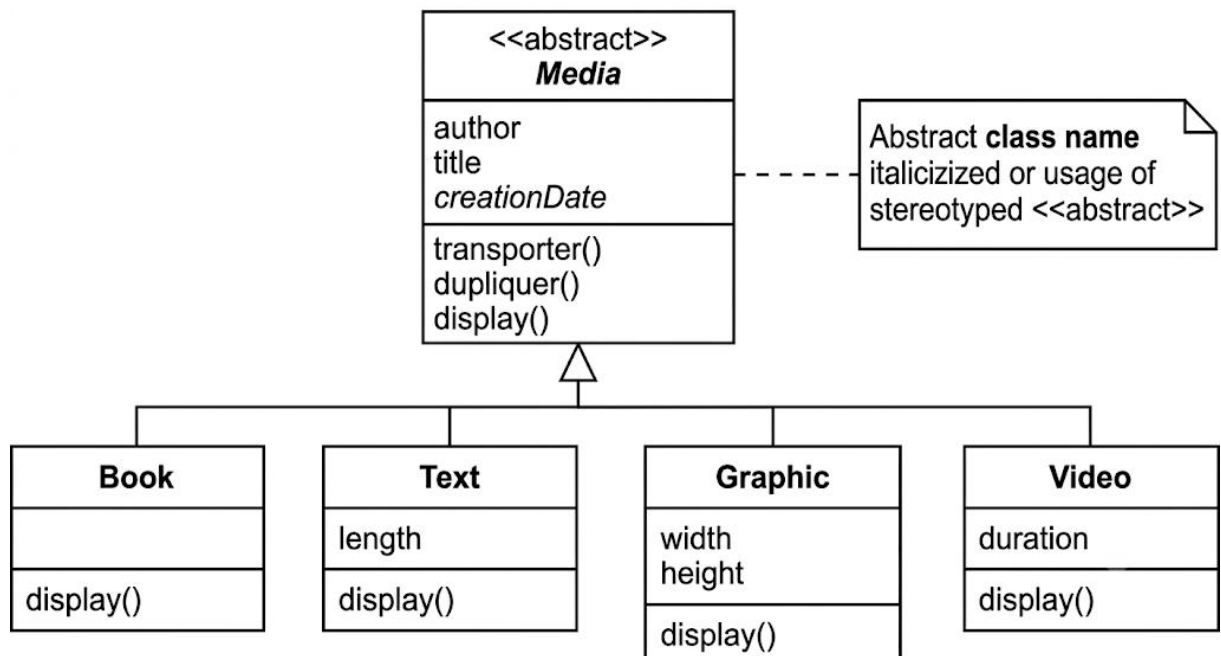


Figure 41. Example of representation of abstract class

- **«enumeration»**: A user-defined data type representing constant values (e.g., AccountType).

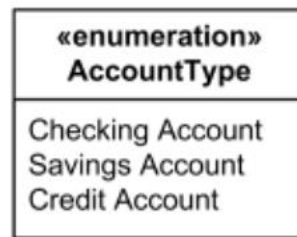


Figure 42. Example of representation of abstract class

Above the class name, we can also add **properties** using **tagged values**.

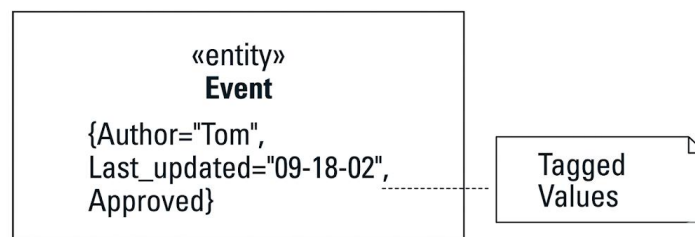


Figure 43. Example of representation of class using tagged values

- Attributes

Attributes represent the information an object possesses. For example, in a theater system, different objects provide specific information about the overall operation. The **Venue** object contains details such as its seating capacity and current status, which may be open, closed for holidays, closed for repairs, or reserved for a private event. The **Event** object provides information about its start and end dates, the number of shows already scheduled, the maximum number of shows that can be scheduled, and its current status, which can be tentative, contracted but not yet scheduled, scheduled, completed, or canceled. Meanwhile, the **Seat** object describes its location within the venue as well as its current condition, indicating whether it is disabled or available for use.

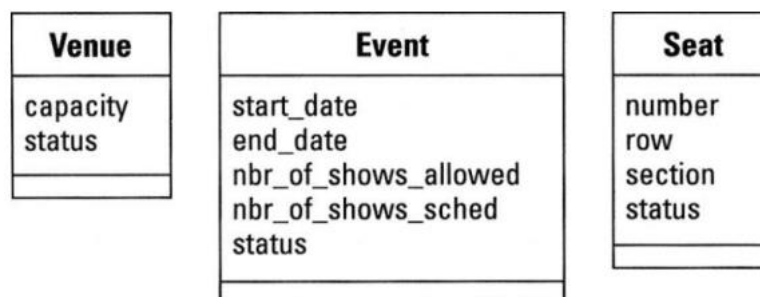


Figure 44. Example of representation of attributes in class diagram

In UML, they follow a specific syntax: **[visibility] [/] name [: type] [multiplicity] [=default] [{property-string}]**

- **Visibility:** Defines who can access the data: **Private (-)**, **Package (~)**, **Public (+)**, or **Protected (#)**.

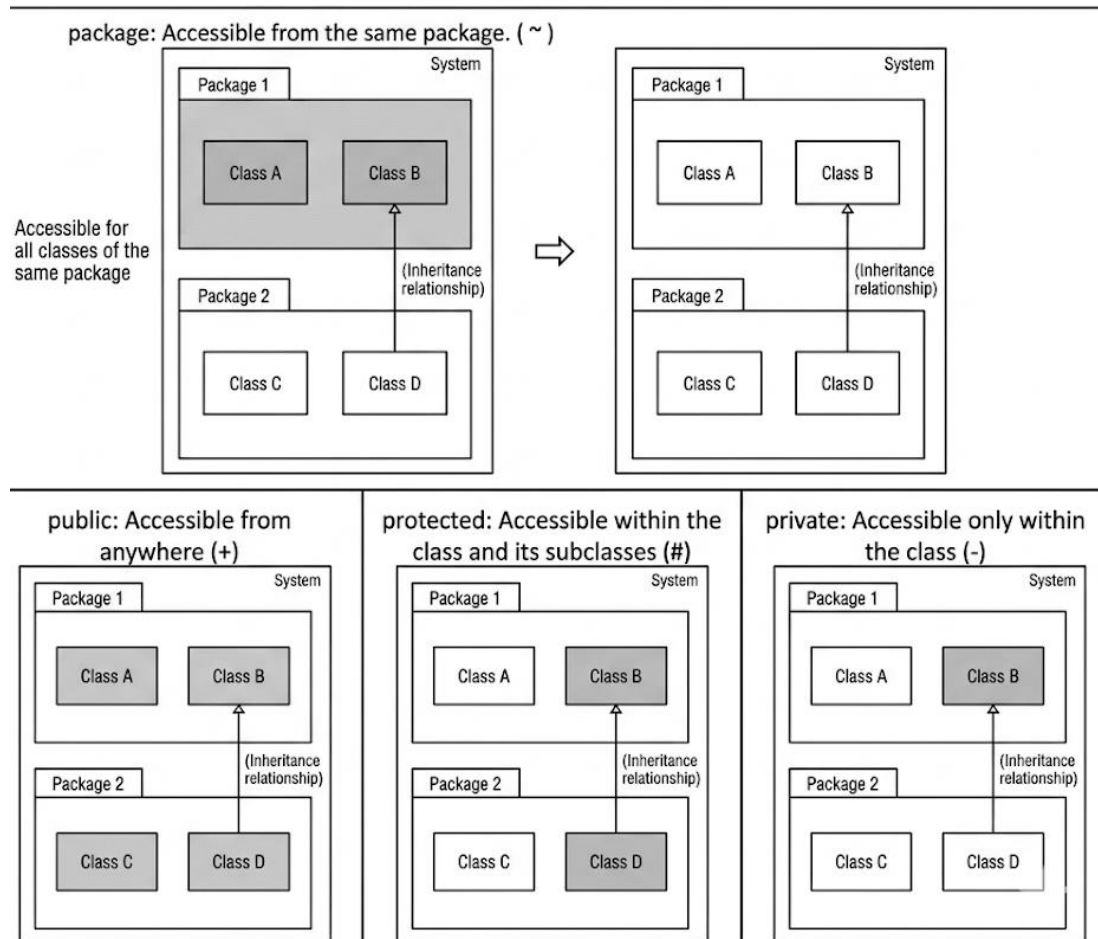


Figure 45. Different representations of attributes visibility

- **Derived Values (/):** Indicated by a forward slash, these are values calculated from other data (e.g., /age derived from birthDate).

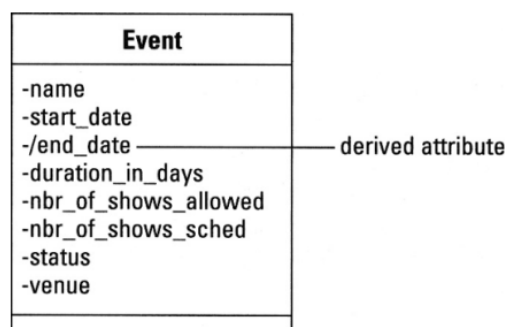


Figure 46. Example of derived attributes in a class

- **Name:** It must be unique within the class.
- **Data type:** It represents the nature of the information that can be stored in the attribute. It can refer to a data type (e.g., *integer*, *float*, *long*, *short*, *string*, etc.), or to another class in the system, or to an enumeration class.

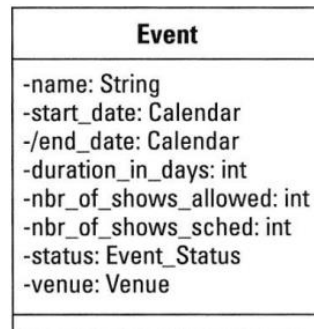


Figure 47. Example of type attributes in a class

- **Multiplicity:** Indicated by [], this defines how many values an attribute can hold (default is 1).

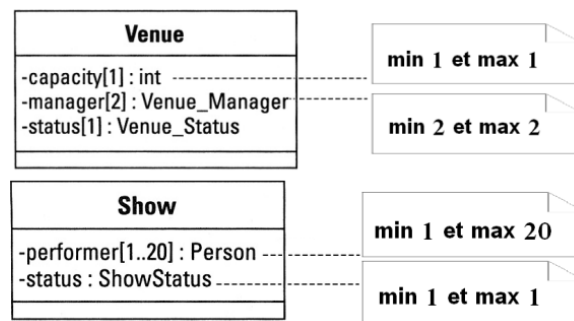


Figure 48. Example of attribute multiplicity in a class

- **Default values:** They help protect the system from being corrupted due to missing or invalid values and provide significant and easy-to-use improvements for the client.

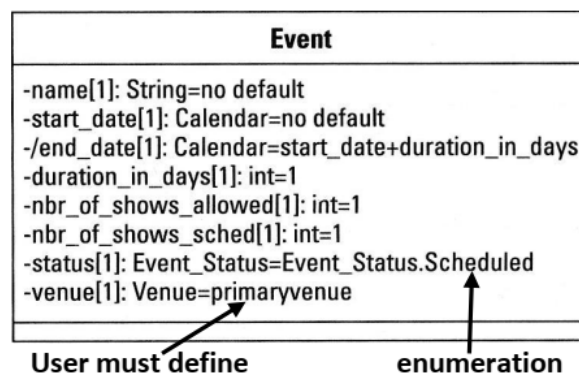


Figure 49. Example of default values of attribute in a class

- **String properties:** A text including any other information regarding the attribute. It is written between { }.

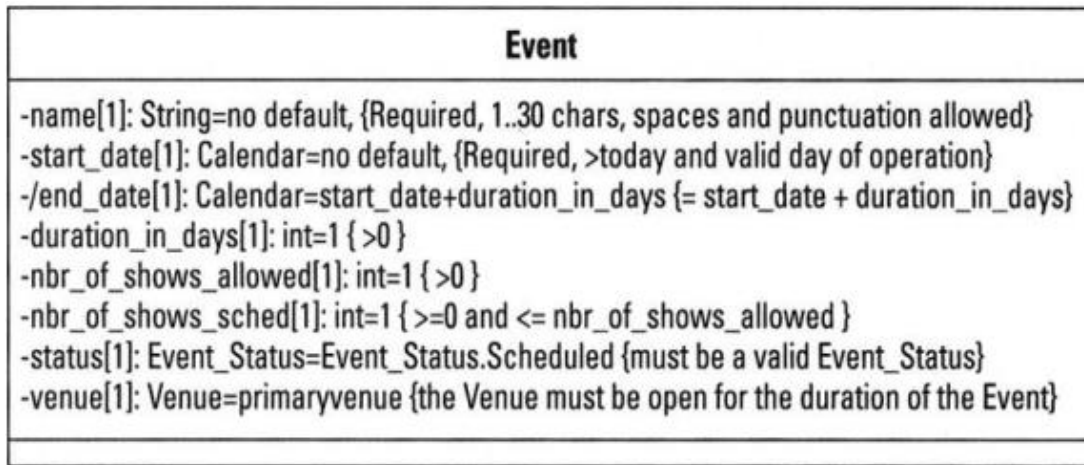


Figure 50. Example of String property of attribute in a class

- **Static Attributes:** While most attribute values belong to a specific object, an instance of a class (called instance-level attributes), **Class-level attributes** are shared by all instances of a class and are **underlined** in the diagram.

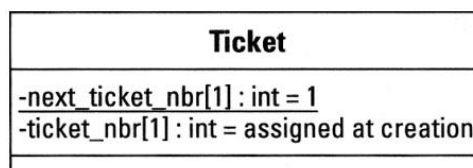


Figure 51. Example of class-level attributes

- Operations

Operations define the behavior of the object (the actions it can perform). They can use stereotype to label their usage.

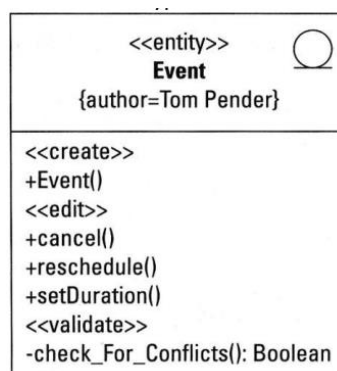


Figure 52. Example of stereotype usage in class operation

Operations follow this syntax: **[visibility] name ([parameter-list]) ':' [return-result] [{properties}]**

The UML operation syntax begins with **visibility** (+, -, #, ~) and the **name**, followed by a **parameter-list** in parentheses. Each parameter can include an optional direction (in, out, inout), a name, a data type, and a default value. This part of the notation clearly defines the operation's identity and the specific inputs required for it to function.

After the parentheses, a colon introduces the **return-result**, which specifies the data type the operation produces, or void if nothing is returned. The signature ends with **properties** inside curly braces, providing extra metadata or constraints. Common examples include {readOnly}, which ensures the object's state isn't modified, or {ordered} to indicate a sorted result.

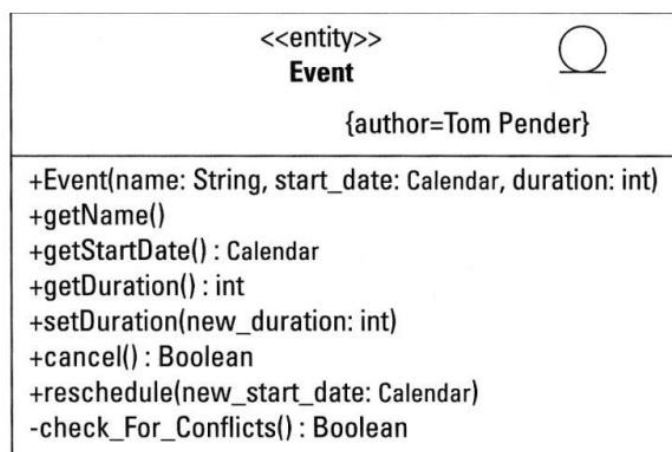


Figure 53. Example of operations in class operation

The most common operations, owned and executed by a specific object are called **instance-level** operations, while **class-level (Static)** Operations are those accessible by the class itself rather than a specific instance. Like static attributes. As static attributes, static operations are underlined in the class.

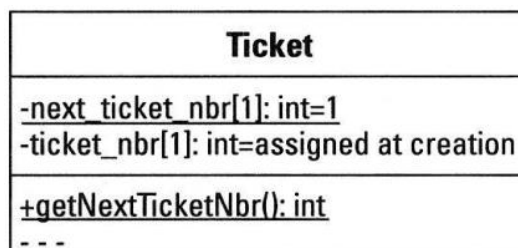


Figure 54. Example of static operations in class operation

By defining these elements clearly, structural modeling ensures that the foundation of the software is robust, manageable, and ready for construction.

3. Modeling relationships

Just as human beings communicate with each other, objects must also find a way to communicate among themselves. To do this, objects use three types of relationships: **association**, **generalization**, and **dependency**

a- Association

The purpose of an association is to establish the reason why two classes need to be aware of each other, as well as the rules that govern their relationship. For example, an event may take place at a location. The event must know where it is taking place, and the location must know what is happening within it. These are two perspectives within the same association. The modeling of an association begins with identifying the participating classes. There are different types of associations: binary associations (between two classes) and n-ary associations.

D. Binary association

It is an association that consists of a connection and two association ends. The instance of a class is called an *object*, and the instance of an association is called a *link*.

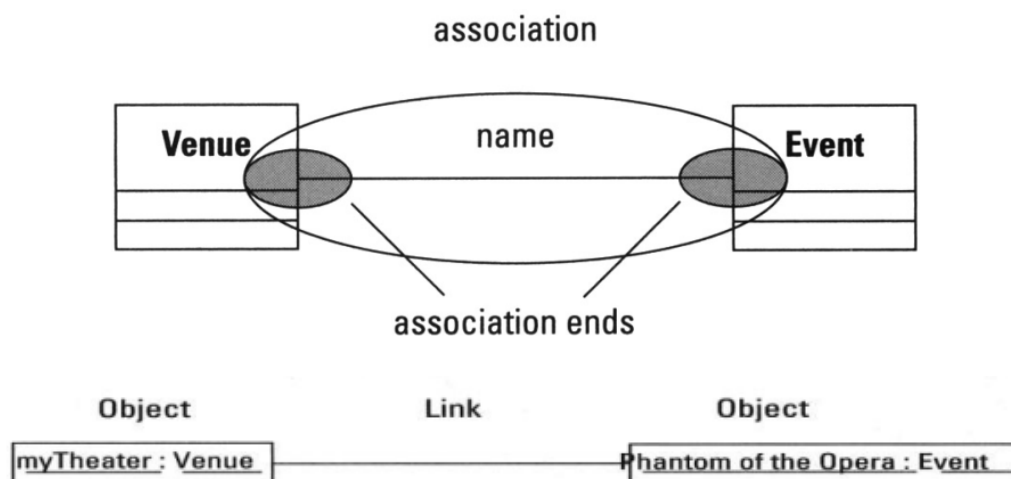


Figure 55. Example of binary association

An association is generally named using a verb or a verbal phrase. The name of the association becomes even more important when two classes have more than one reason

to collaborate. In the theater example, one venue might sponsor an event, while others host the event.

Association ends define the participation of each object in the association. Each association end can include the following characteristics: role, interface specifier, visibility, multiplicity, order, constraint, qualifier, navigability, and changeability.

- **Role**

The role explains how an object participates in the association. For example, in the theater scenario, the venue may act as the sponsor of an event and may also be the host of an event.

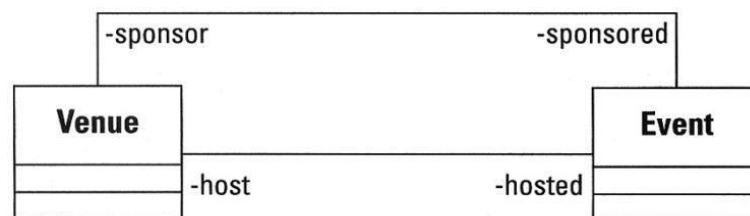


Figure 56. Example of a role name in a binary association

The role can generate code during implementation (it becomes an attribute that refers to the class it belongs to). The code for the previous example would be:

```
class Venue{

private Event hosted ;

private Event sponsored;

}

class Event{

private Venue host;

private Venue sponsor;

}
```

- **Interface specifier**

It is a property of an association end that defines which specific interface an object must use when interacting with another via that link. Instead of an object seeing the entire target class, it only sees the methods and attributes defined in a specific interface. This is a key principle for **decoupling** systems, as it ensures a class only depends on the behaviors it actually needs.



Figure 57. Example of interface specifier in a binary association

for example, when an **Event** interacts with a **Venue** over this specific link, it is restricted from accessing the **Venue's** full range of public methods. Instead, it must interact strictly through the operations defined in the **SponsorInterface**. This ensures the **Event** only sees the sponsorship-related functionality of the **Venue**, effectively hiding any unrelated attributes or behaviors of the **Venue** class from that specific connection.

- **Visibility**

Indicates whether the associated object can be seen by other elements outside the association. It is typically denoted by symbols like + (public), - (private), # (protected), or ~ (package).

- **Multiplicity**

Defines the number of instances of one class that can be linked to a single instance of another class (e.g., 0..1, 1..*, *). the "hosts" association specifies that one or more **Venues** can be linked to zero or more **Events**, while the "sponsors" association indicates that exactly one **Venue** acts as a sponsor for any number of **Events**.

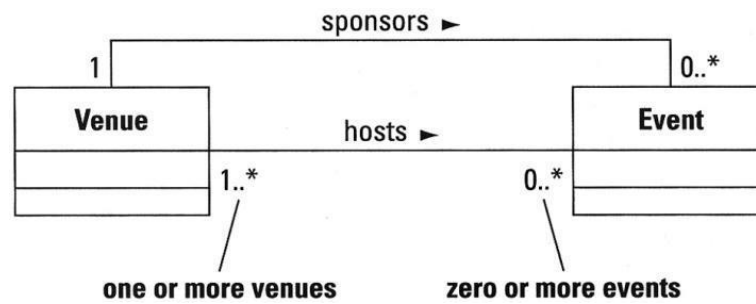


Figure 58. Example of multiplicity in a binary association

- **Order**

It specifies whether the set of related objects at that end follows a specific sequence. Common values include ordered (a specific sequence is maintained) or unordered. For example, when a venue hosts a certain number of events, the list of events must be organized in a specified sequence, whereas it is not when a venue sponsors an event.

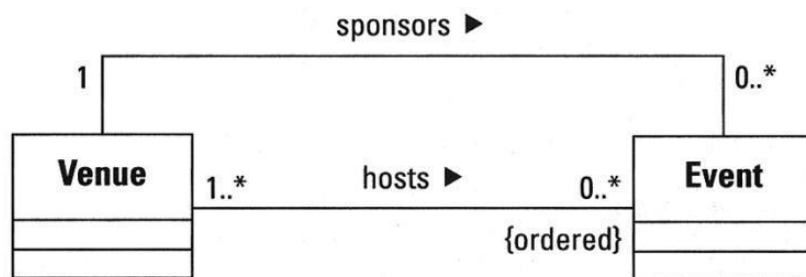


Figure 59. Example of order in a binary association

- **Constraint**

This is one of the three UML extension mechanisms used to define rules concerning an association. For example, when a venue hosts an event, it must be ensured that the cost of this event does not exceed a certain amount, that events must be sorted by start date, and that each event must be approved by the venue manager (VM).

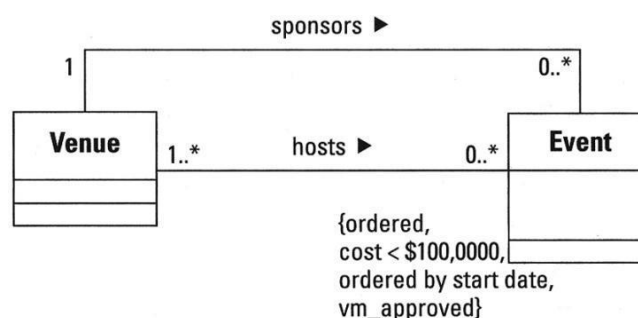


Figure 59. Example of constraints in a binary association

- **Qualifier**

It is a property or attribute used to uniquely identify or refine the related objects in an association. It acts like a key to distinguish instances in an association. For example, A **Venue** can host multiple **Events** and each **Event** at a Venue is uniquely identified by its **Eventid** (the qualifier).

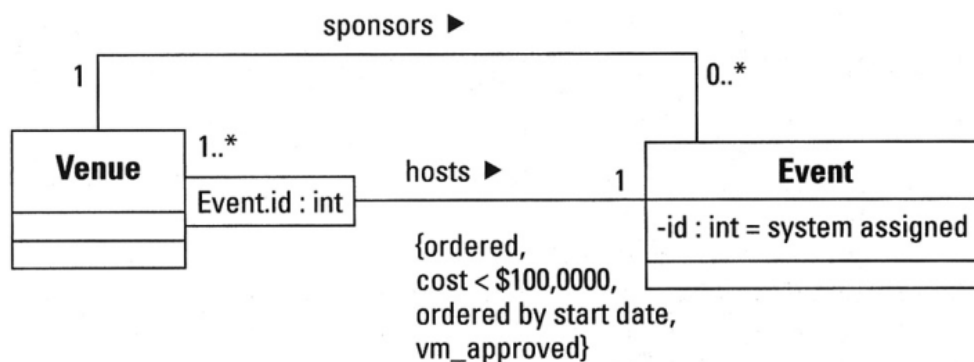


Figure 60. Example of qualifier in a binary association

- **Navigability**

Determines if an object can "see" or access the other object at the end of the link. It is shown by an arrow for one-way navigation or a line without arrows for two-way navigation.

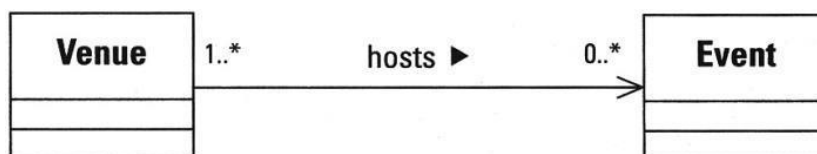


Figure 61. Example of navigability in a binary association

- **Changeability**

This property allows us to specify the operations permitted on the links defined by an association. The predefined options include {frozen}, which means that once a link has been established, it cannot be modified or moved. If the application only allows creating new links (but no deletion), use the {addOnly} property.

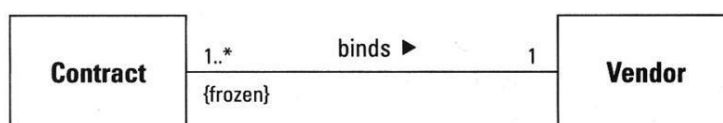


Figure 62. Example of changeability in a binary association

E. Reflexive association

A **reflexive association** occurs when a class has an association relationship with itself. This means that an instance of the class can be linked to other instances of the same class. For example, a single VenueManager act as an **auditor** oversees zero or more other VenueManagers who fulfill the **coordinator** role.

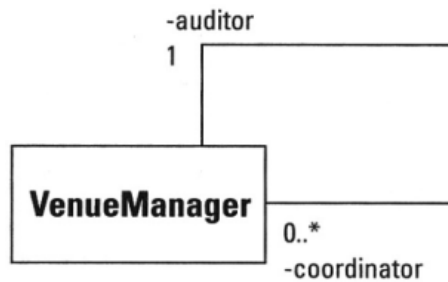


Figure 63. Example of reflexive association

F. N-ary association

This type of association is rarely used in UML models. It is an association between multiple classes. The symbol used to represent this type of association is a diamond. For example, to define a calendar, there must be an event, a location manager, and a venue for the event. As an example of an **n-ary association**, the **Venue**, **Event**, and **VenueManager** classes are connected to a central diamond to show they are part of one atomic relationship. This structure indicates that a specific booking or assignment only exists when all three participants are present simultaneously.

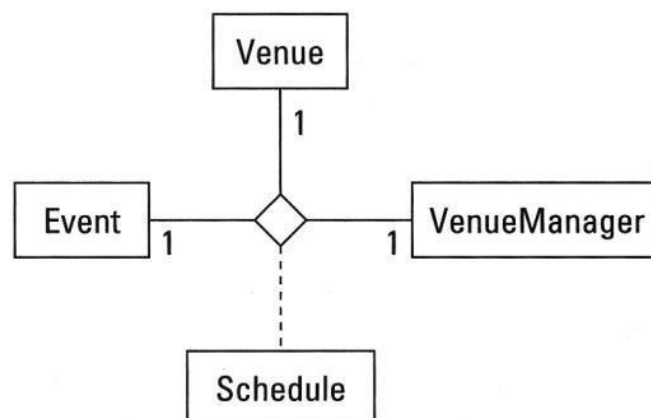


Figure 64. Example of n-ary association

G. Aggregation

In a typical association, the participating classes are peers. Each class remains independent of the other and neither is superior. Aggregation, on the other hand, refers to an assembly or configuration of elements to create a larger and more complex unit. Aggregation includes a control point (called the aggregate) that represents the master object having control over its parts (no actions on the parts are allowed without the aggregate's permission). An aggregation can represent either a physical or a logical assembly. For example, a computer is composed of several devices (keyboard, mouse, etc.), and a director manages several employees (agents, department heads, etc.). The control point can represent multiple levels — for instance, an engine controls its parts, and the car controls the engine and its parts. An aggregation is represented by an **unfilled diamond** next to the aggregate (the assembling class). For example, an **Agency** acts as the aggregate "whole" composed of at least one **Agent**, where each agent functions in the role of an employee. While an agency must have one or more members (1..*), an individual employee is restricted to belonging to no more than one agency and may currently belong to none (0..1). Furthermore, the model enforces a specific constraint where an agent is only considered an active employee within the agency if they possess a valid, current contract.

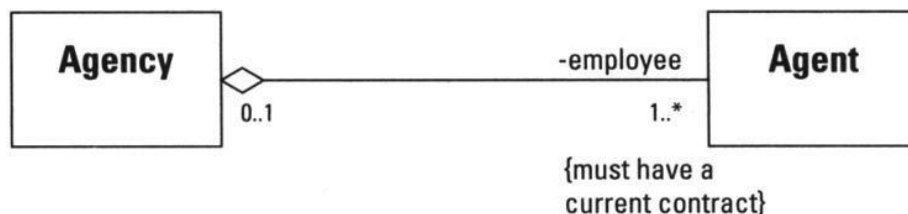


Figure 65. Example of aggregation association

H. Composition

Composition is used for aggregations where the lifetime of the member object depends on the lifetime of the aggregate. The aggregate not only controls the behavior of the member but also has control over its creation and destruction. This high level of responsibility held by the aggregate is why composition is referred to as a **strong aggregation**. Composition is represented similarly to aggregation, but with a **filled diamond** next to the aggregate (the owning class).

b- Dependency

Dependency represents a client-supplier relationship in which a change to the supplier requires a change to the client. Dependency can also represent precedence — for example, in a theater system, selecting a seat on the touchscreen precedes the payment of the ticket.

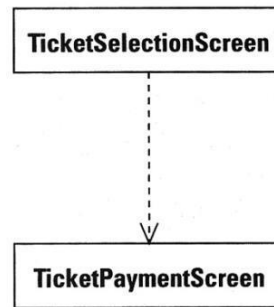


Figure 66. Example of dependency relationships

c- Generalization

Generalization is the process of organizing the properties of a set of objects that share the same purpose. Generalization is also synonymous with **inheritance**. For example, in a theater system, a play is a type of event, so a play inherits all the functionalities of an event. A play is also a **specialization** of an event because it inherits all the generalized functionalities of an event and adds unique functionalities, or replaces/redefines existing ones. **Event** is called a **superclass**, and the other classes are called **subclasses**, and the diagram is referred to as a **class hierarchy**. Another representation of generalization can also be used

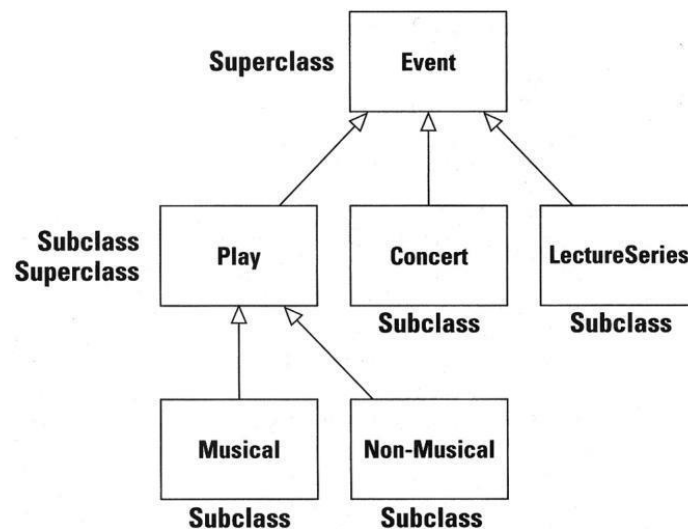


Figure 67. Example of generalization relationships

An important point to note is that a **leaf class** in the hierarchy can never be an **abstract class** (a class from which no objects can be instantiated).

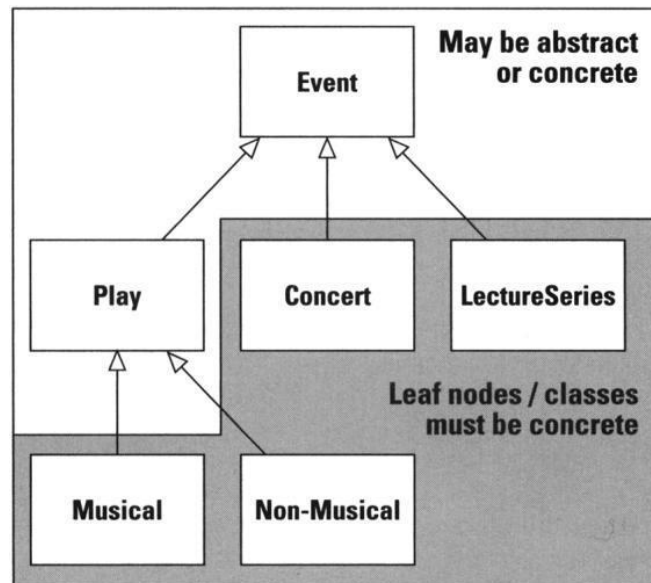


Figure 68. Example of difference between leaf and nodes in a generalization relationships

When modeling a generalization, you must describe how the subclasses of a superclass were identified. For example, concerts can be categorized based on the size of the performance (solo or group). A solo concert can be further categorized according to the size of the band, and a group concert according to the size of the group.

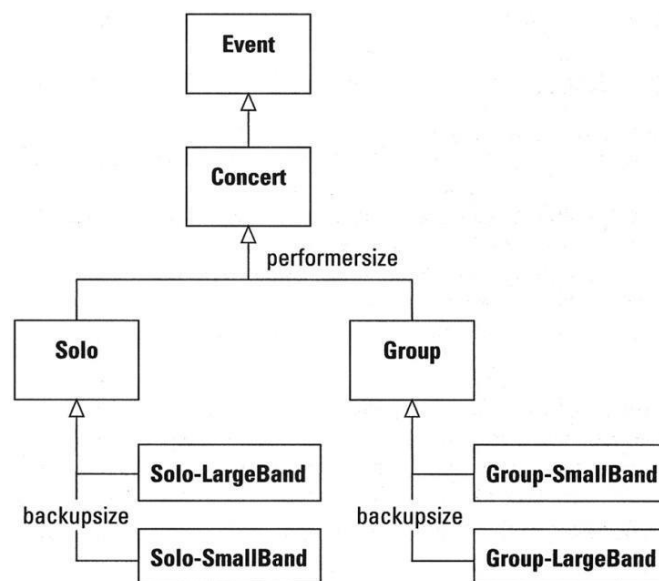


Figure 68. Example of generalization identification

d- Association class

An association between two classes often requires information not only about the participating objects but also about the association itself. For example: When did the association begin? When will it end? Are there any conditions that apply to the association?

An association class is a class that describes a relationship between classes. For instance, an association between agents and the theater company can be described in terms of start date, end date, terms and conditions, and authorization. These descriptive attributes are encapsulated in a class called Contract. The Contract class is then connected to the Authorization association it describes using a dashed line. The dashed line has no arrow.

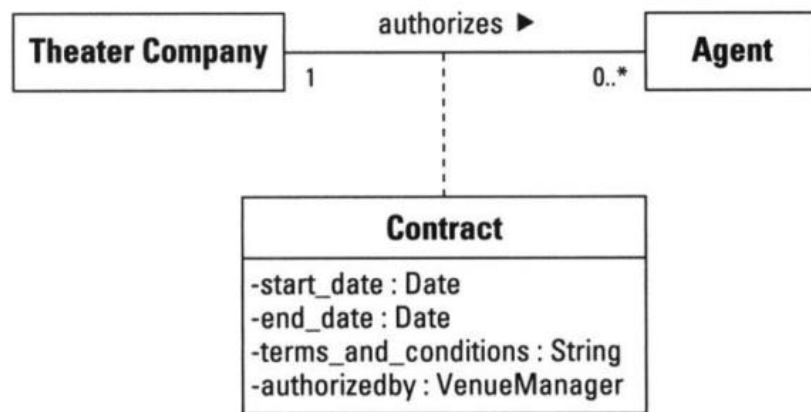


Figure 69. Example of class association

An association class is a class like any other class, and therefore, it can have associations with other classes.

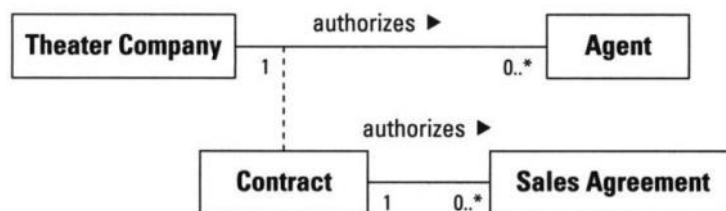


Figure 70. Example of class nature of a class association

Chapter 7: Behavioral Modeling: Sequence Diagrams

In the realm of software engineering, capturing the static structure of a system is only half the battle. While a map shows us the layout of a city, it tells us nothing of the traffic flowing through its streets or the rhythm of its daily life. To truly understand a system, we must look beyond its components and witness its **behavior**. Behavioral modeling serves as the pulse of the design process, shifting our focus from what a system *is* to how it *acts*.

Sequence diagrams are the definitive tool for this exploration, providing a chronological narrative of communication. By mapping the exchange of messages between objects over time, these models transform abstract logic into a clear, visual choreography. In this chapter, we delve into the mechanics of the **Sequence Diagram**, mastering the art of documenting interaction to ensure that every process—from the simplest login to the most complex transaction—is executed with precision and clarity.

1. Definition

The **Sequence Diagram** is the primary tool used to model the execution of a scenario within a system. It is widely considered the most important UML diagram after the Class Diagram because it focuses on the **sequential order** of interactions.

While developers use them to map out code execution, these diagrams are equally valuable for business stakeholders. They provide a clear visual for discussing how a business currently operates by showing how different entities—from users to databases—interact to accomplish a task. Essentially, it answers two questions: **Which messages are sent, and when?**

A sequence diagram is made up of two basic building blocks: the frame and the interaction.

2. Frame

The frame is the space in which the sequence diagram is drawn. It represents the border that surrounds the diagram and consists of a header and a contents area.

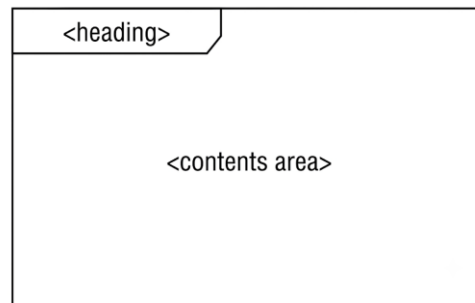


Figure 71. Frame representation in UML

3. Interaction

Simply put, an interaction is a sequence of messages exchanged between objects to accomplish a task. Objects can be created and destroyed. They can ask questions or make requests to other objects by invoking operations.

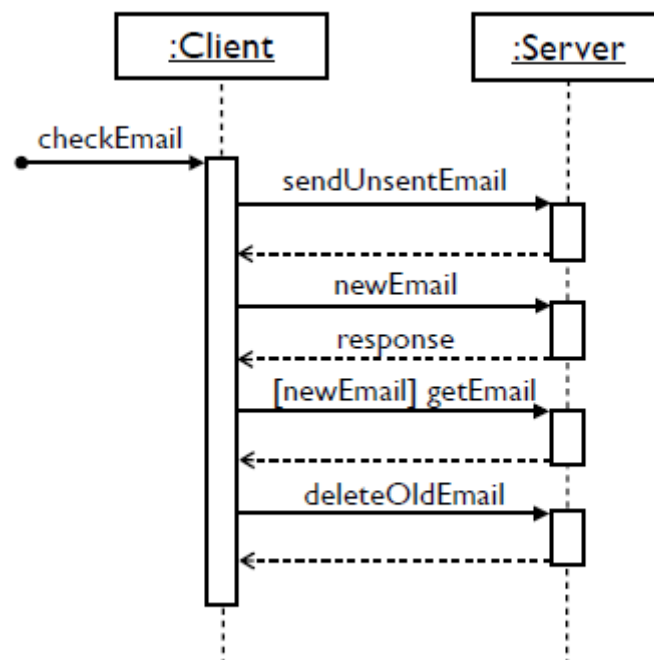


Figure 72. Example of interaction in sequence diagram

To help illustrate how interaction works in a real-world scenario, we can examine a common interaction: **Checking Email**. This process involves a series of messages between a Client (the user's device) and a Server. The interaction follows a specific chronological order to ensure data integrity and synchronization:

- The user clicks the "Check Email" button.
- First, the client sends all the emails that have not yet been sent.
- After receiving an acknowledgment of receipt, the client asks the server if there are any new emails.
- If there are, it downloads the new email.
- Then, it deletes the old overwritten emails from the server.

To draw a sequence diagram, we need to master three main elements: **who** is involved, **when** they are working, and **what** they are saying.

- Lifeline (Who)

The Lifeline represents an object's existence over time. It is represented by a box with the object's name at the top and a vertical dashed line stretching downward.

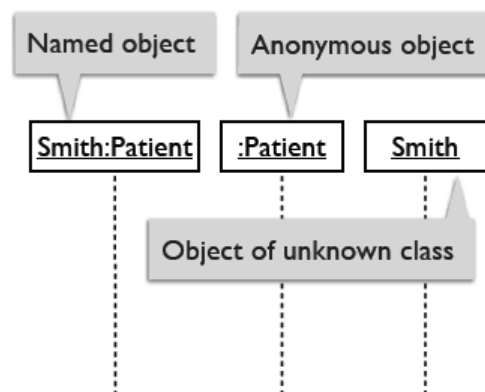


Figure 73. Example of representation of lifeline in a sequence diagram

In sequence diagrams, timing is everything. One of the most important concepts to visualize is the **creation of an object** during the system's execution. There is a critical difference between **object creation** and a **standard message call**. When the message arrow points directly to the **object box**, it indicates a **Constructor Message** (for example creation of the `:Performance`). When the message arrow points to the **dashed lifeline**, it signifies that the object already exists and is simply being invoked to execute a method.

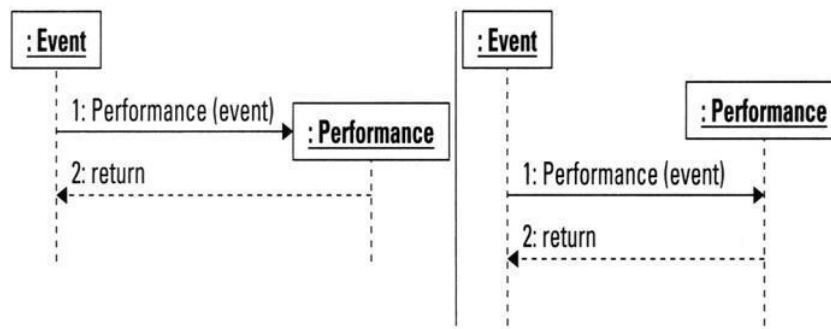


Figure 74. Example of difference between constructor and method calling in sequence diagram

- Control focus (When)

The control focus describes the period during which an object performs an action, either directly or through a subordinate procedure. The activation of an object is represented by the start and end of the control focus.

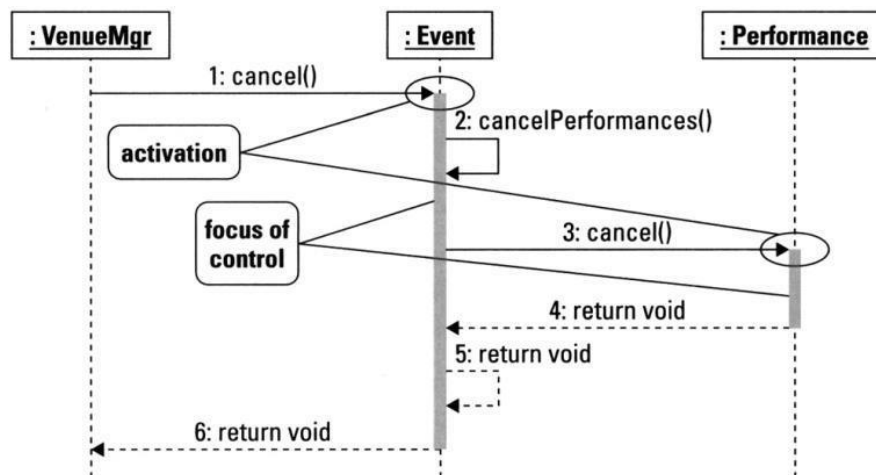


Figure 75. Example of control focus in sequence diagram

- Message

A message is the basic form of communication in a sequence diagram. It defines a specific type of communication. The communication may invoke an operation, and create or destroy an instance. The message specifies not only the type of communication but also the sender and the receiver. The message is represented by horizontal arrows. We can distinguish several types of message:

➤ **Synchronous message**

A synchronous message calls a method of an object and waits for a response. The call parameters must match the parameters of the invoked method's prototype.

➤ **Asynchronous message**

An asynchronous message calls a method of an object and does not wait for a response. The call parameters must match the parameters of the invoked method's prototype.

➤ **Constructor message**

A message that creates an object is called a constructor. It is represented by a dashed line with an open arrowhead.

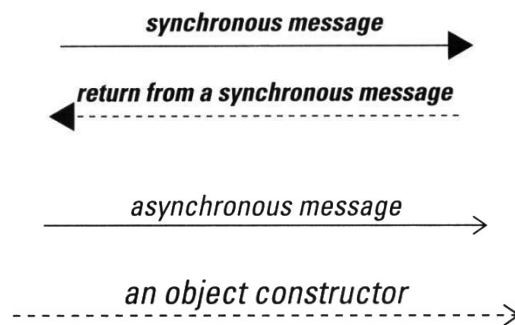


Figure 76. Principle of synchronous, asynchronous and constructor message

➤ **Reflexive message**

A reflexive message represents a call to an internal method of an object.

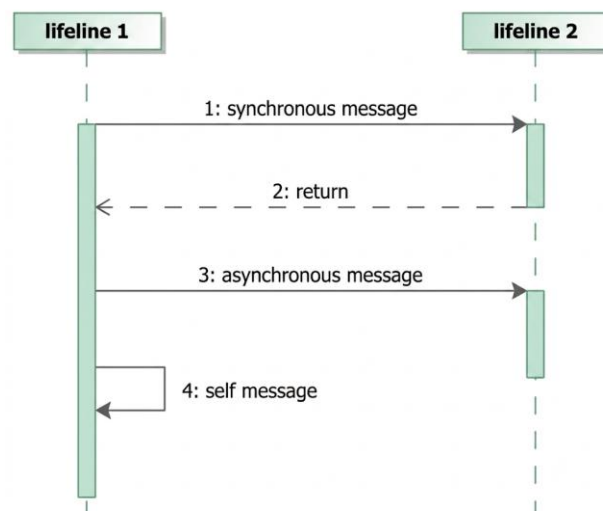


Figure 77. Example of reflexive message in sequence diagram

4. Example of sequence diagrams

In this section, we give a set of examples that resume most frequent blocs found in sequence diagrams.

- Example of synchronous message

In this example, labeled **sd Rechercher livre** (Search book), a **Client** initiates a synchronous interaction by invoking the `chercher("Tintin")` method on the **:Médiathèque** object. This is represented by a solid line with a filled arrowhead, indicating that the client must wait for a result before proceeding. Upon receiving the request, a **control focus** appears on the library's lifeline to show it is actively processing the search. Finally, the library returns the result, `nombreLivres`, via a dashed response arrow, which signals the end of the operation and allows the client to resume its activity.

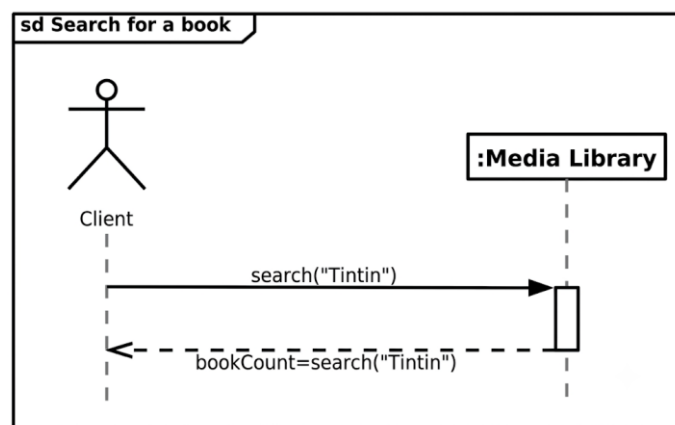


Figure 78. Example of synchronous message in sequence diagram

- Example of asynchronous message

In this **sd Retrait argent** (Cash withdrawal) example, a **Client** initiates an asynchronous interaction by sending the `introduireCarte` method to the **:Distributeur** object. This is represented by a solid line with an **open arrowhead**, signifying that the client triggers the action and does not wait for a response before continuing. The diagram highlights the message lifecycle through four key events: the **sending event** at the client's lifeline, the **reception event** at the dispenser, and the subsequent **start and end of execution** events. The control focus on the **:Distributeur** lifeline illustrates the period during which the machine processes the card independently, allowing the system to handle multiple parallel actions without blocking the sender.

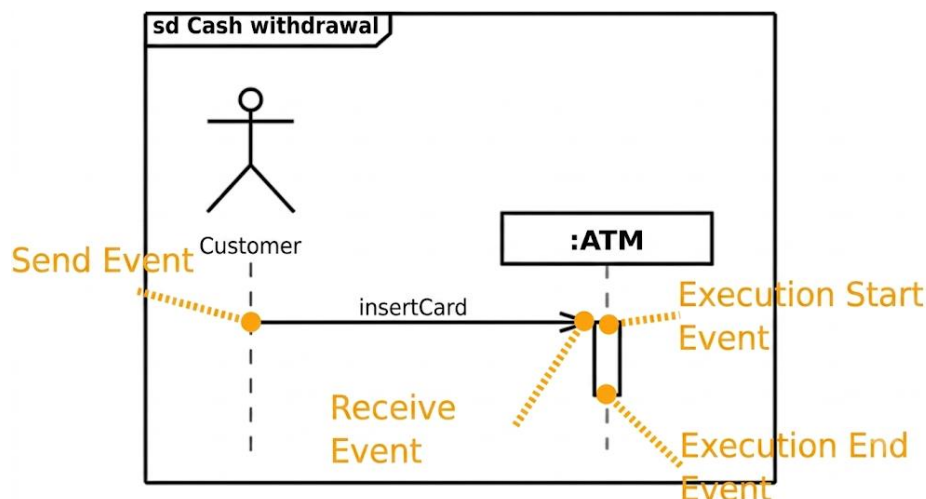


Figure 79. Example of asynchronous message in Cash withdrawal sequence diagram

Another example is the common interpersonal interaction, where the person **P1** sends the message Envoyer SMS (Send SMS) to the person **P2**. This is depicted as an asynchronous interaction because the sender, **P1**, does not wait for a response or a "read receipt" before moving on with their own lifeline. Once the message reaches **P2**, a brief control focus appears on their lifeline, representing the moment the message is received and processed, while **P1** remains free to initiate other tasks or complete their own execution independently.

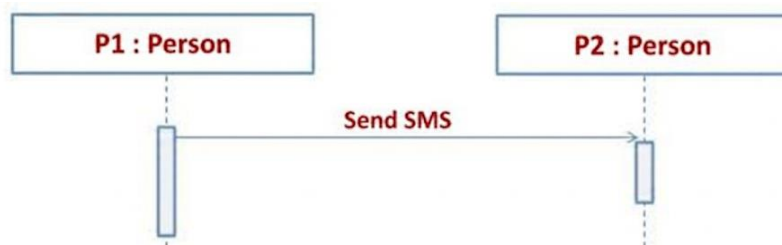


Figure 80. Example of synchronous message in SMS delivery sequence diagram

- Example of constructor and destructor messages

In these examples, the system manages the lifecycle of an account within an online library environment. The object **:LibrairieEnLigne** creates a new instance **c: Compte** by invoking the `Compte()` constructor. This is represented by a dashed line with an open arrowhead pointing directly to the object's box, which is vertically positioned to mark the exact moment of its creation. The same system terminates the object's existence by sending a `<<destroy>>` message. This is visually confirmed by the large **red "X"** at the end

of the **c: Compte** lifeline, indicating that the object has been removed from memory and its lifecycle has officially ended.

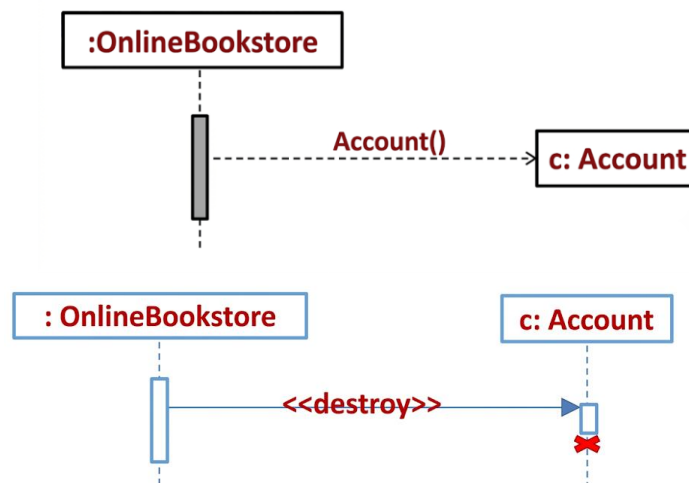


Figure 81. Example of object creation and destruction in a sequence diagram

- Example of lost message and unknown source

In this example, the diagram illustrates how a system handles communication failures and external inputs. The interaction begins with a small circle, representing a **found message** from an **unknown source**, where the sender exists outside the scope of the current diagram. The message `introduireCarte` (`insertCard`) is dispatched toward the **:Distributeur** object but is intercepted by a red "X", which is the standard notation for a **lost message**. This signifies that the transmission failed and never reached its destination. As a result, the expected **start of execution event** on the dispenser's lifeline is never triggered, visually demonstrating a scenario where the receiver remains inactive because the signal was lost in transit.

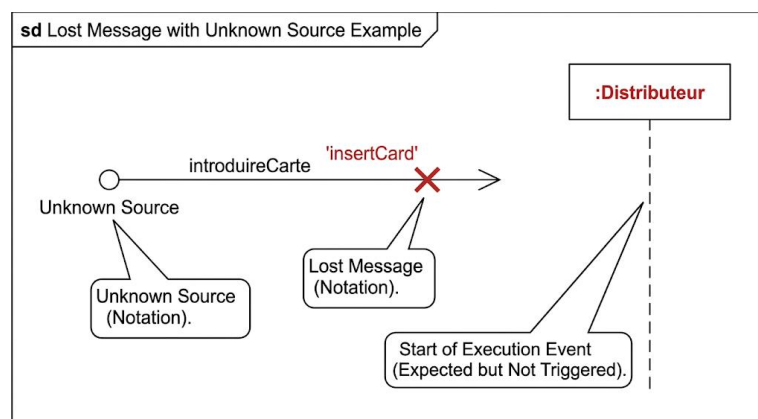


Figure 82. Example of lost message and unknown source in a sequence diagram

5. Advanced interaction fragments

In addition to standard message flows, UML sequence diagrams use **Interaction Fragments** (combined fragments) to model complex logic like conditions, loops, and error states. These are represented by frames with a specific operator in the top-left corner.

- Option fragment (OPT)

The **Opt** fragment models an "if-then" scenario. It contains a single interaction that only occurs if a specific **guard condition** is met. If [condition] is **True**, the Action() is performed. If **False**, the entire block is ignored.

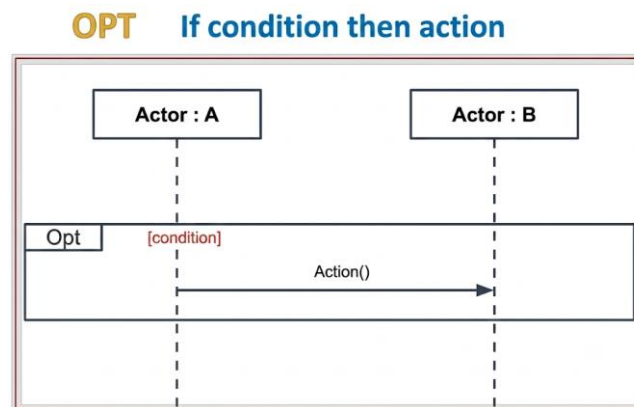


Figure 83. Representation of option fragment in a sequence diagram

- Iterative fragment (LOOP)

The Loop fragment represents repetitive behavior. The interaction inside the frame is executed multiple times based on a condition or a specific range. The bloc continue its execution as long as the [condition] remains True. It can also be noted as LOOP (min, max) to define exactly how many times the sequence repeats.

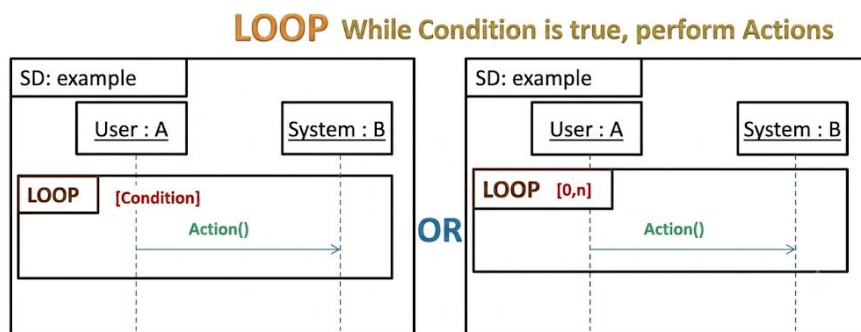


Figure 84. Representation of iterative fragment in a sequence diagram

- Alternative fragment

The Alt fragment models an "if-then-else" structure. The frame is split into two or more horizontal operands separated by a dashed line. If the first [condition] is met, ActionA() is executed. If that condition is false, the [else] path is taken, and ActionB() is performed instead. Only one of the paths can ever be executed.

ALT If condition then actionA otherwise actionB

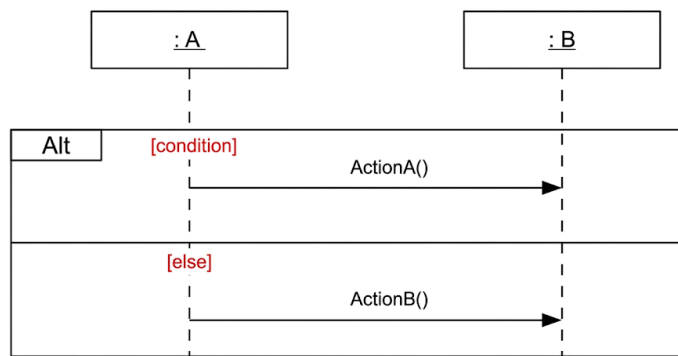


Figure 85. Representation of alternative fragment in a sequence diagram

- Interruption fragment

The Break fragment is a specialized scenario used to model an "interruption" or an early exit. If the [condition] is verified, the interactions inside the Break frame are performed, and then the rest of the enclosing fragment is skipped. This is often used to model exception handling or "cancel" operations.

Break Fragment executed and terminates the enclosing fragment if a condition is true

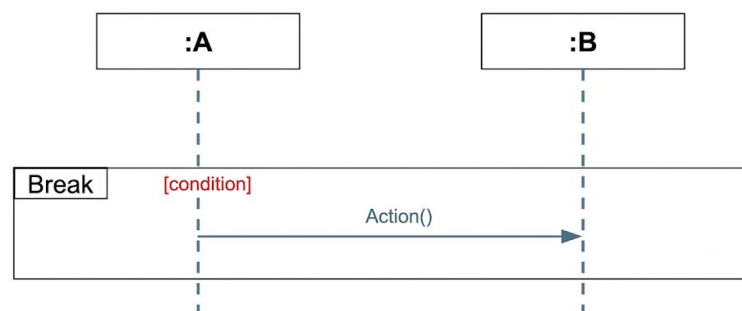


Figure 86. Representation of interruption fragment in a sequence diagram

An example of application of some of this fragment is the **sd Online_Bookshop** example. Here the the system uses a **loop** fragment to allow the **:Web Customer** to repeat searches within a single session. After receiving search results, two independent **opt** fragments

provide the flexibility to either view book description or add to shopping cart based on user preference. This demonstrates how combined fragments model iterative behavior and conditional logic without forcing a strictly linear execution.

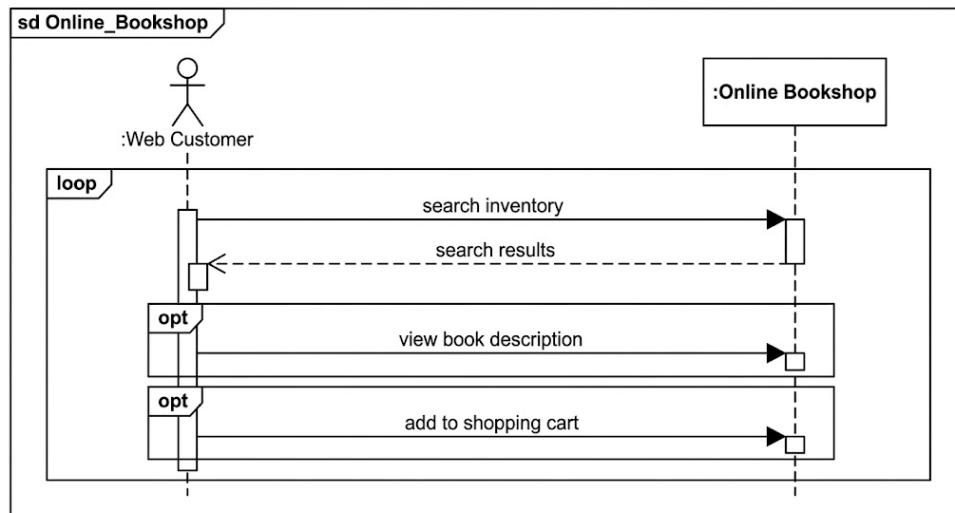


Figure 87. Example of fragment application in a sequence diagram

Conclusion

As students conclude this journey through the fundamentals of JAVA and the strategic blueprints of the Unified Modeling Language, it becomes clear that professional software development is a dual discipline of logic and design. By mastering the syntax of JAVA alongside the architectural clarity of UML, students have moved beyond simply writing code to engineering resilient, scalable systems.

The chapters within this handbook have equipped students with the tools to translate complex real-world requirements into structured class diagrams and dynamic sequence models. Whether students are defining the functional scope of a project or optimizing internal mechanics, the principles of encapsulation, inheritance, and formal modeling remain their most reliable guides. As students step forward into their own development projects, it is vital to remember that the most effective software is not just functional—it is well-documented, thoughtfully modeled, and built upon a foundation of structural integrity.

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.).