Palash Bajpai

UML Class

Diagrams

Table of Contents

[**1. What is UML?** 2](#_Toc192169584)

[1. Types of UML 2](#_Toc192169585)

[**2. Understanding Class Diagrams** 4](#_Toc192169586)

[**3. Basics of UML Class Diagrams** 6](#_Toc192169587)

[Class Representation 6](#_Toc192169588)

[Attributes and methods 6](#_Toc192169589)

[**4. Relationships in Class Diagrams** 8](#_Toc192169590)

[1. Associations 8](#_Toc192169591)

[2. Aggregation 10](#_Toc192169592)

[3. Composition 11](#_Toc192169593)

[4. Generalization (Inheritance) 12](#_Toc192169594)

[5. Dependency 13](#_Toc192169595)

[**5. Advanced Class Diagram Concepts** 15](#_Toc192169596)

[Multiplicity 15](#_Toc192169597)

[Interfaces and Abstract Classes 16](#_Toc192169598)

[Enums 17](#_Toc192169599)

[Association Class 17](#_Toc192169600)

[Do’s and Don’ts of UML Class Diagram 18](#_Toc192169601)

[**6. UML Class Diagrams for Real-World Systems** 19](#_Toc192169602)

[UML Class Diagram for an E-commerce System 19](#_Toc192169603)

[UML Class Diagram for Library Management 20](#_Toc192169604)

# **1. What is UML?**

Unified Modeling Language (UML) is a standardized visual modeling language used to design and document software systems. It helps developers, architects, and stakeholders understand and communicate the structure and behavior of a system

✔️ provides a standard way to visualize system architecture.

✔️ Supports object-oriented design (OOD).

✔️ Used for designing, analyzing, and documenting software systems.

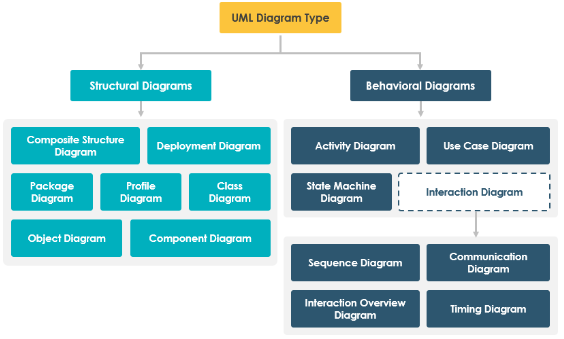
✔️ Independent of programming languages and development methodologies.

**Real World Example:**

Imagine you are developing an E-commerce Application (like Amazon). UML helps visualize:

* How users interact with the system.
* How products, orders, and payments are related.
* How different components communicate.

## 1. Types of UML



**Structural Diagrams**

These diagrams represent the static aspects of a system—how components are structured.

| **UML Diagram** | **Purpose** | **Example** |
| --- | --- | --- |
| **Class Diagram** | Defines object structure & relationships. | Customer has Orders in an e-commerce system. |
| **Object Diagram** | Represents instances of classes at a specific time. | A real-time snapshot of Order and its associated Product. |
| **Component Diagram** | Shows how components interact. | Payment module in a banking app. |
| **Deployment Diagram** | Represents hardware/software deployment. | AWS cloud infrastructure for a web app. |
| **Package Diagram** | Organizes related classes. | Organizing Customer, Order, and Product in different packages. |

**Behavioral Diagrams**

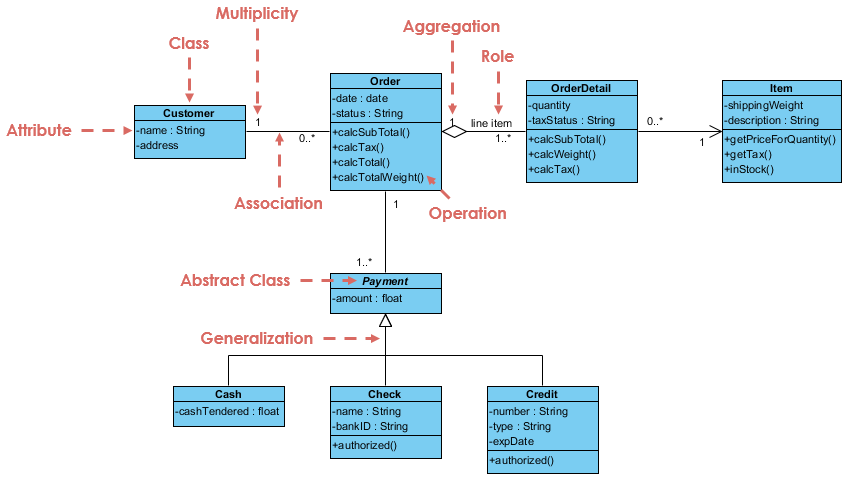
These diagrams represent the dynamic aspects of a system—how components behave over time.

| **UML Diagram** | **Purpose** | **Example** |
| --- | --- | --- |
| **Use Case Diagram** | Defines user interactions with the system. | A Customer placing an Order. |
| **Sequence Diagram** | Shows message flow between objects. | User authentication in a login system. |
| **Activity Diagram** | Represents workflows & processes. | Steps in an online payment transaction. |
| **State Diagram** | Represents object states & transitions. | Order transitioning from Placed → Shipped → Delivered. |

# **2. Understanding Class Diagrams**

A Class Diagram is a UML diagram that represents the static structure of a system. It shows:

* Classes (objects) and their properties/methods.
* Relationships between classes (Association, Inheritance, etc.).
* Visibility (public, private, protected).✔️ provides a standard way to visualize system architecture.



**Role of Class Diagrams**

* Blueprint of the System: Helps design object-oriented software before coding.
* Easy Communication: Developers, architects, and stakeholders understand relationships better.
* System Maintenance: Helps in debugging and extending functionalities.
* Reduces Complexity: Breaks large software into smaller, manageable parts..

**Similarity with OOPs**

Mapping of UML concepts to OOP

| **UML Concept** | **OOP Concept** | **Example** |
| --- | --- | --- |
| **Class** | Blueprint of objects | BankAccount, Customer, Order |
| **Attributes** | Class variables | balance in BankAccount |
| **Methods** | Functions in a class | withdraw(), deposit() |
| **Association** | Object relationship | Customer has multiple Orders |
| **Generalization** | Inheritance | AdminUser extends User |
| **Composition** | "Has-a" relationship | Car has an Engine |

# **3. Basics of UML Class Diagrams**

## Class Representation

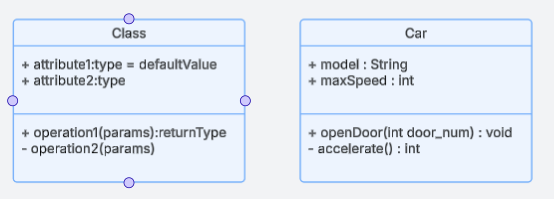
A Class in UML represents a blueprint for objects. It defines attributes (data) and methods (behavior).

A class in UML is depicted as a rectangle divided into three sections:

1️. Class Name (First Section)

2️. Attributes (Properties/Fields) (Second Section)

3️. Methods (Functions/Operations) (Third Section)



## Attributes and methods

**Attributes:** Attributes define the state of a class (fields/variables).

**Methods:** Methods define behavior (actions).

**Visibility:** Visibility notations indicate the access level of attributes and methods.

| **Visibility** | **Symbol** | **Meaning** |
| --- | --- | --- |
| Public | + | Accessible by any class |
| Private | - | Only accessible within the class |
| Protected | # | Accessible by the class and subclasses |

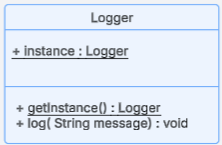
Static methods and attributes

Static Attributes: Belong to the class rather than an instance.

Static Methods: Operate at the class level rather than instance level.

**Notation in UML:**

* Static attributes are underlined in class diagrams.
* Static methods are underlined as well.



# **4. Relationships in Class Diagrams**

In UML, relationships between classes define how objects of different classes interact. Understanding these relationships is crucial for object-oriented design and software modelling.

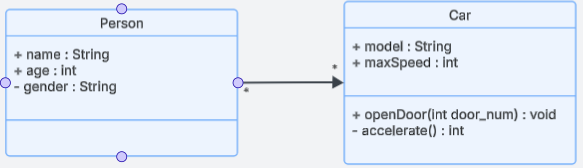
## 1. Associations

An Association represents a relationship between two or more classes where objects of one class are linked to objects of another class. It can be bidirectional or unidirectional.

**Notation**

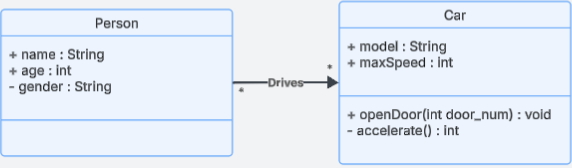
* Solid Line (—) connects related classes.
* Arrow (→) for unidirectional associations.

Eg1: User can use car, but car can’t use user



Multiple times we can even add uses on the connecting arrows

Eg2: Person drives the car



**Types of Associations:**

1️. One-to-One

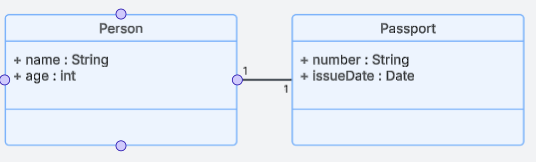
2️. One-to-Many

3️. Many-to-Many

1. **One-to-One (1:1)**

Each instance of Class A is related to exactly one instance of Class B.

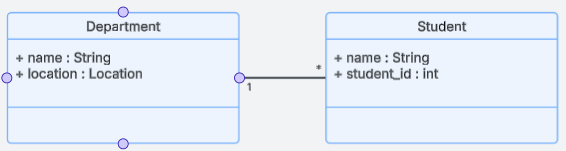
Eg: A person can have exactly one Passport



1. **One-to-Many (1:M)**

One instance of Class A is related to multiple instance of Class B.

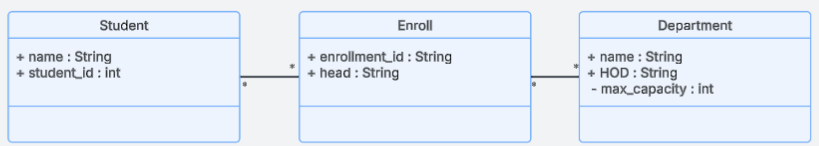
Eg: A department can have multiple students



1. **Many-to-Many (1:M)**

Multiple instance of Class A is related to multiple instance of Class B.

Eg: A Student can enrol in multiple Courses, and a Course can have multiple Students.



## 2. Aggregation

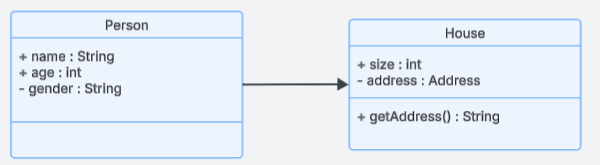
Both Aggregation and Composition represent “whole-part” relationships but differ in ownership strength.

Aggregation denotes a stronger relationship where one class (the whole) contains or is composed of another class (the part).

**Key points**

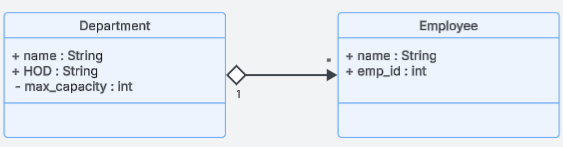
* Represents "Has-A" relationship
* Is a weaker relationship
* Child can exist independently of Parent
* **Notation**: Hollow Diamond (◊) at the whole

Eg: Person has a house, but house can exist independent from person



Aggregation is represented by a diamond shape on the side of the whole class. In this kind of relationship, the child class can exist independently of its parent class.

Eg 2: A Department (whole class) has multiple Employees, but Employees can exist independent of department too.



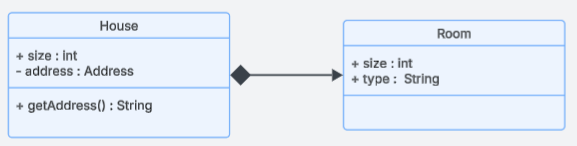
## 3. Composition

Composition is a stronger form of aggregation, indicating a more significant ownership or dependency relationship. In composition, the part class cannot exist independently of the whole class.

**Key Points:**

* Represents "Part-Of" relationship
* Child CANNOT exist without Parent
* Notation: Filled Diamond (◆) at the whole

Eg: A House has multiple Rooms, but a Room cannot exist without a House.



**Key Differences:**

| **Feature** | **Aggregation** | **Composition** |
| --- | --- | --- |
| **Dependency** | Child exists independently | Child cannot exist without Parent |
| **Lifespan** | Child’s life is independent | Child’s life depends on Parent |
| **Example** | Company & Employees | House & Rooms |

## 4. Generalization (Inheritance)

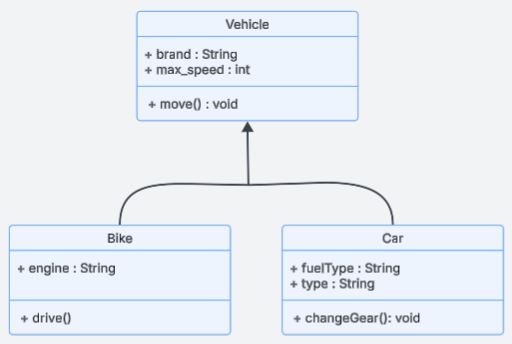
It represents an "is-a" relationship between a parent class (superclass) and its child classes (subclasses). It models inheritance in object-oriented design, allowing the child classes to inherit attributes and behaviors from the parent class while also enabling specialization.

**Key Points:**

* Represents "IS-A" relationship (Parent-Child relationship).
* Superclasses (Parent) define common attributes and methods.
* Subclasses (Child) inherit these features.
* Notation: Arrow (▲) pointing to Superclass

Eg: Vehicle -> Car, bike

* Car and Bike inherit Vehicle properties (brand, speed, move()).
* Car has its own method drive(), and Bike has ride().

****

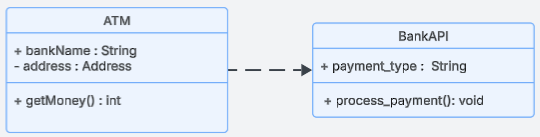
## 5. Dependency

It represents a weak relationship between two classes where one class depends on another temporarily. It signifies that a change in one class may impact the other, but they do not have a strong association like inheritance (generalization) or composition.

**Key Points:**

* It is a one-way relationship where one class relies on another for its functionality.
* It is represented by a dashed arrow pointing from the dependent class to the independent class.
* The dependent class uses the other class but does not own it permanently.

Eg: An ATM class depends on a BankAPI class to retrieve account details and process transactions.



**Types of Dependency:**

1. **Parameter Dependency:** A class depends on another when an object of one class is passed as a parameter.

class Order {  
 void processPayment(Payment payment) {  
 payment.pay();  
 }  
}

Here, Order depends on Payment but does not store it permanently.

1. **Return Type Dependency:** A method returns an object of another class.

class UserRepository {  
 User getUser(int id) {  
 return new User(id, "John");  
 }  
}

1. **Local Variable Dependency :** A method creates and uses an instance of another class.

class Order {  
 void process() {  
 Payment payment = new Payment();  
 payment.pay();  
 }  
}

1. **Exception Dependency:** A class depends on another when it throws an exception of that class type.

class Database {  
 void connect() throws SQLException {  
 // Code to connect to DB  
 }  
}

# **5. Advanced Class Diagram Concepts**

## **Multiplicity**

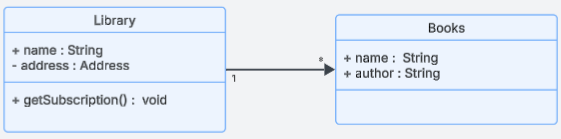
Multiplicity in UML defines how many instances of a class can be associated with an instance of another class. It helps in defining cardinality constraints in relationships between objects in a UML class diagram.

**Key concepts:**

* Specifies the number of instances that can participate in a relationship.
* Expressed as ranges (e.g., 1..\* means at least 1 and unlimited maximum).
* Used in association, aggregation, and composition relationships.

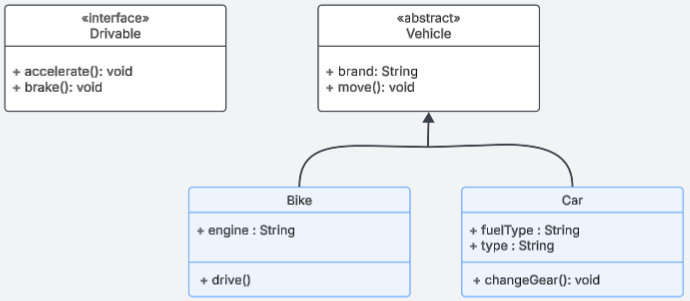
| **Notation** | **Meaning** |
| --- | --- |
| 1 | Exactly one instance |
| 0..1 | Zero or one instance (optional) |
| \* or 0..\* | Zero or more instances (unlimited) |
| 1..\* | At least one instance (one or more) |
| n | Exactly n instances |
| m..n | Between m and n instances |

Eg: A Library can have multiple Books



## **Interfaces and Abstract Classes**

| **Feature** | **Abstract Class** | **Interface** |
| --- | --- | --- |
| **Definition** | A class that **cannot be instantiated** and may contain **both abstract and concrete methods**. | A contract that contains **only method declarations** (no implementation). |
| **Purpose** | Provides **partial implementation** for subclasses. | Defines a **standard set of methods** that different classes must implement. |
| **Methods** | Can have **both abstract and non-abstract** methods. | Only **abstract methods** (default in Java 7 and earlier). |
| **Variables** | Can have instance variables. | Variables are **public, static, and final** by default. |
| **Constructors** | Can have a constructor. | Cannot have a constructor. |
| **Multiple Inheritance** | Supports **single inheritance** (A class can extend only one abstract class). | Supports **multiple inheritance** (A class can implement multiple interfaces). |
| **UML Representation** | Italicized class name with a **stereotype** <<abstract>>. | Italicized class name with a **stereotype** <<interface>>. |



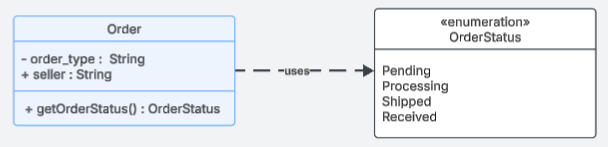
## **Enums**

Enums define a fixed set of values that a variable can take.

Used when a property has limited predefined values.

**Notation in UML:**

* Enum classes use <<enumeration>>.
* Enum values are listed inside the class.



## **Association Class**

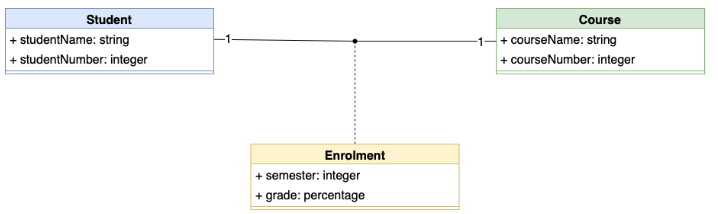
Used when a relationship itself has attributes.

Represents a relationship as a separate entity.

When a relationship itself carries important data.

**Notation in UML:** A dashed line box between two related classes.

**Eg**: A Student enrolls in a Course. The Enrollment relationship has attributes like Grade, Enrollment Date.



## **Do’s and Don’ts of UML Class Diagram**

Do’s

1. Use meaningful class names
2. Use Inheritance when necessary
3. Keep diagram readable and organized
4. Instead of one massive diagram, split it into multiple diagrams according to usecase
   * **Core Business Objects** (e.g., User, Order, Payment).
   * **Subsystems** (e.g., Shipping, Inventory, Security).
   * **Microservices Interaction Diagrams** (for distributed architectures).
5. Use Color Coding and Legend for Large Diagrams

Don’ts

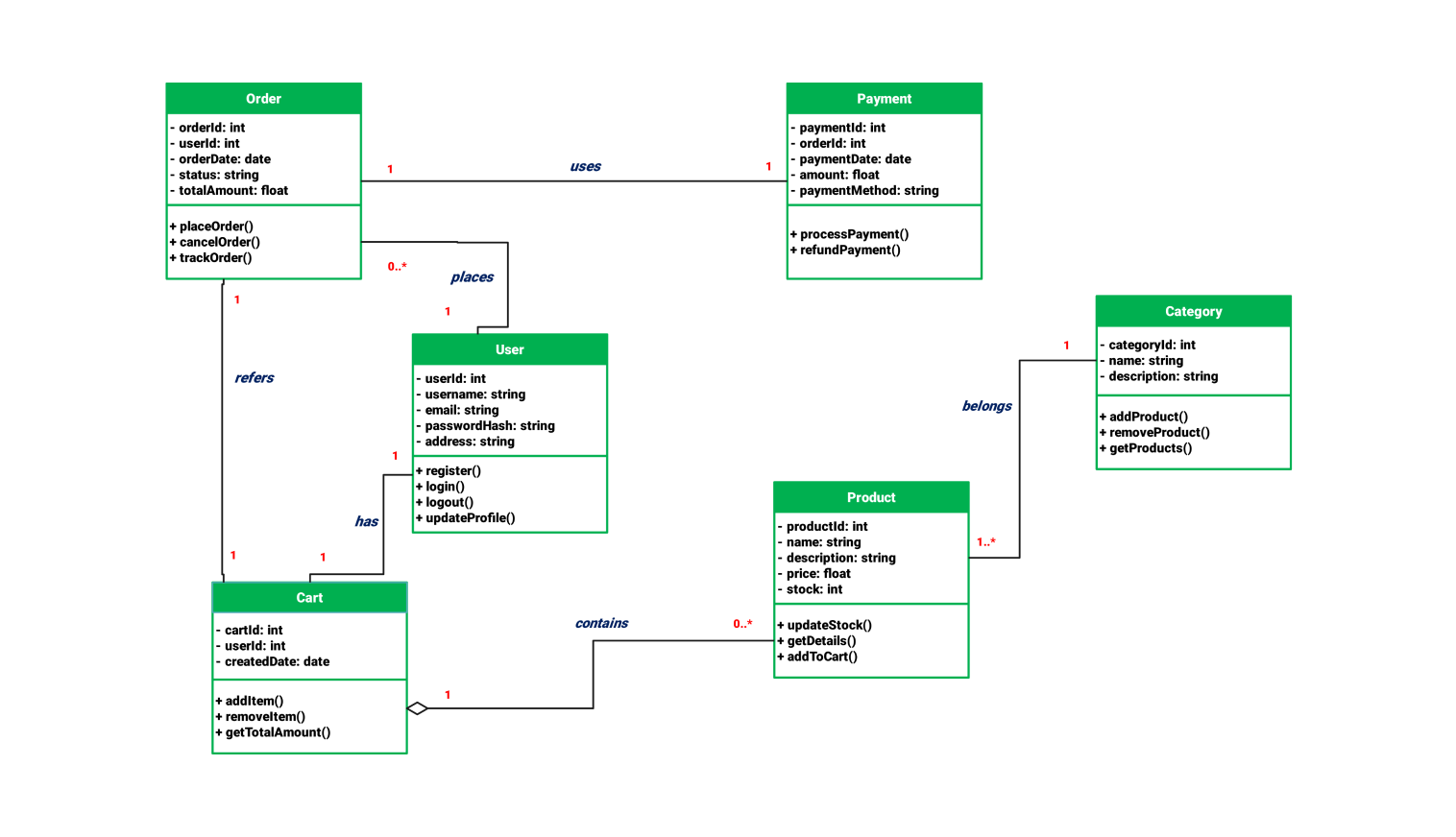
1. Don’t add unnecessary methods and attributes
2. Don’t add too many classes in one diagram

# **6. UML Class Diagrams for Real-World Systems**

## **UML Class Diagram for an E-commerce System**

**Classes & Relationships:**

* User: Can place multiple Orders.
* Order: Contains multiple Products and is associated with a Payment.
* Product: Can be part of multiple Orders.
* Payment: Associated with one Order.



## **UML Class Diagram for Library Management**

­­

