**Unified Modeling Language (UML)** is a standard way to visualize the design of a software system.

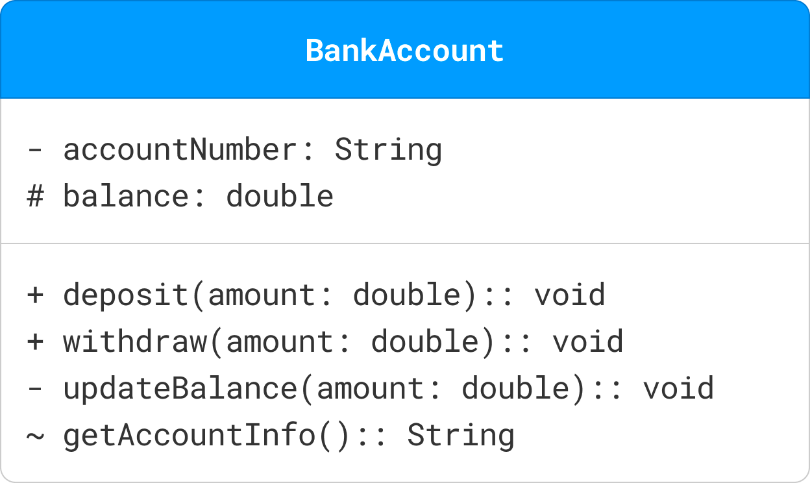
**UML Class Diagram**

UML class diagram provides a static view of an object oriented system, showcasing its **classes, attributes, methods, and the relationships among objects.**

**Building Blocks of UML Class Diagrams**

**1. Class**

A **class** is a blueprint or template that defines the properties and behavior of an object.



Represented as rectangles, classes are divided into three compartments:

* **Name (top compartment):** The unique identifier of the class (e.g., BankAccount).
* **Attributes (middle compartment):** The properties or data associated with the class (e.g., accountNumber, balance).
* **Operations (bottom compartment):** The actions or methods that can be performed by objects of the class (e.g., deposit(), updateBalance()).

**Visibility Markers**: Visibility markers indicate the accessibility of attributes and methods within a class.

* +**(Public):** The attribute or method is accessible from any class.
* -**(Private):** The attribute or method is only accessible within the same class.
* #**(Protected):** The attribute or method is accessible within the same class and its subclasses.
* ~**(Package):** The attribute or method is accessible within the same package.

**2. Attributes**

Attributes in a UML class diagram represent the properties or data fields of a class.

Attributes are typically written in the format: visibility name: type [multiplicity] = defaultValue

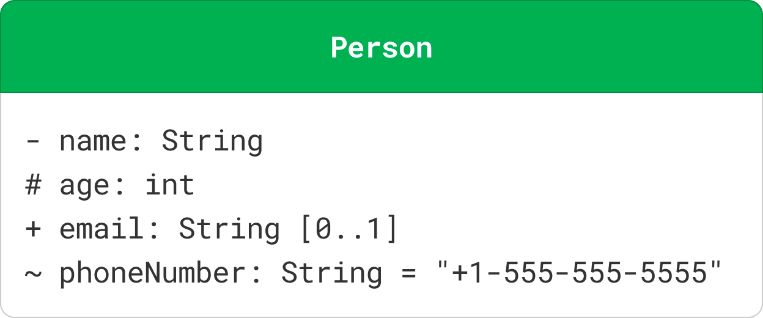
**Visibility:** Public / Private / Protected

**Name**: The name of the attribute.

**Type**: The data type of the attribute.

**Multiplicity**: (Optional) Indicates how many instances of the type are allowed.

**Default Value**: (Optional) The initial value of the attribute.



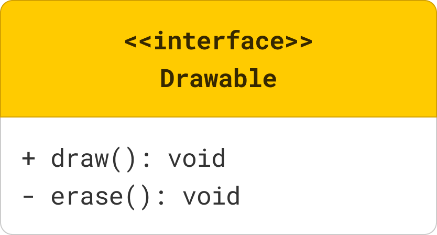
**3. Methods**

Methods (or operations) in a UML class diagram represent the functions or behaviors that a class can perform.

visibility name(parameterList): returnType

**4. Interfaces**

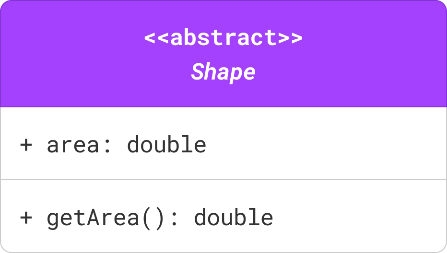
An interface defines a contract for classes that implement it. It specifies a set of methods that the implementing classes must provide.



**5. Abstract Class**

An abstract class is a class that cannot be instantiated (you can't create objects directly from it). It serves as a blueprint for other classes (subclasses) that inherit from it.

An abstract class in UML is represented with the class name in italics and the keyword «abstract» above the class name. Abstract methods within the class are also typically shown in italics.



**6. Enumeration**

An enumeration is a data type that defines a set of named values (e.g., colors, days of the week).



**7. Multiplicity**

Multiplicity specifies the number of instances of one class that can be related to a single instance of another class.

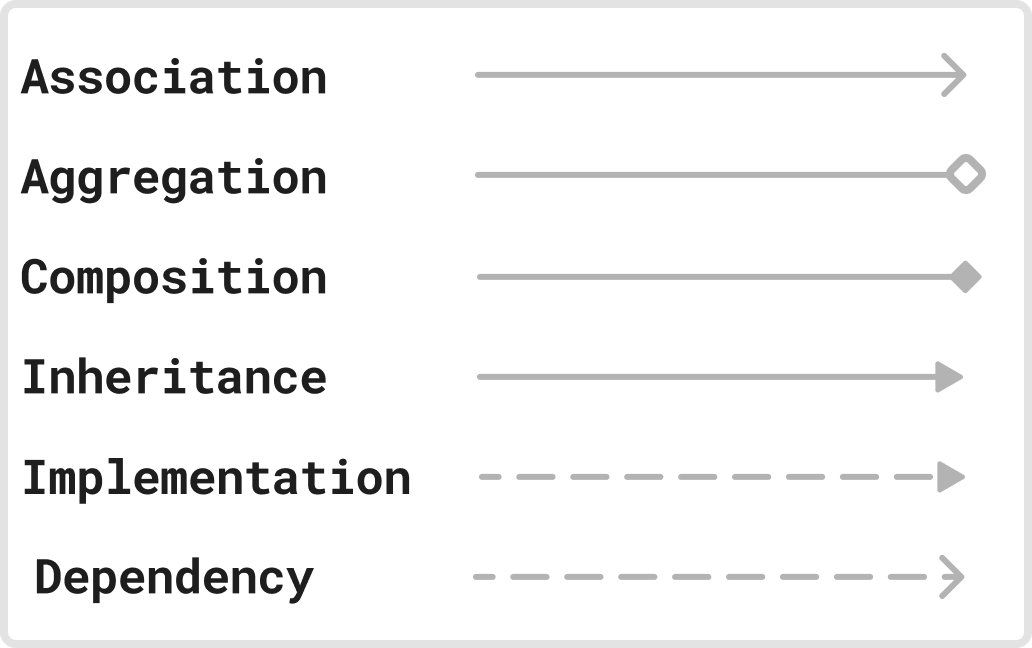
It is represented by a number or a range of numbers near the end of an association line.

Common multiplicities include:

* **1 - (exactly one),**
* **0..1 - (zero or one),**
* **\* - (zero or more),**
* **1..\* - (one or more).**

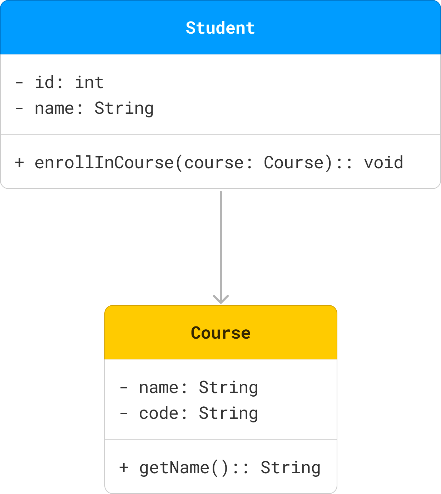
**Relationships in UML Class Diagrams**

There are six main types of relationships between classes:



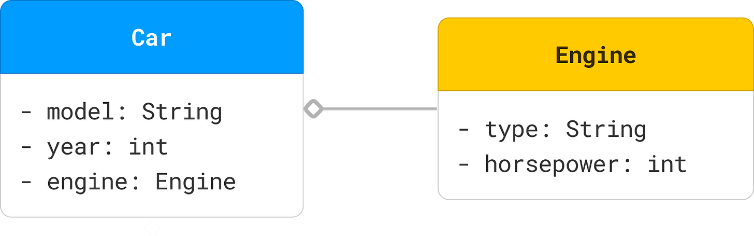
**1. Association**

Association represents a "uses-a" relationship between two classes where one class uses or interacts with the other.



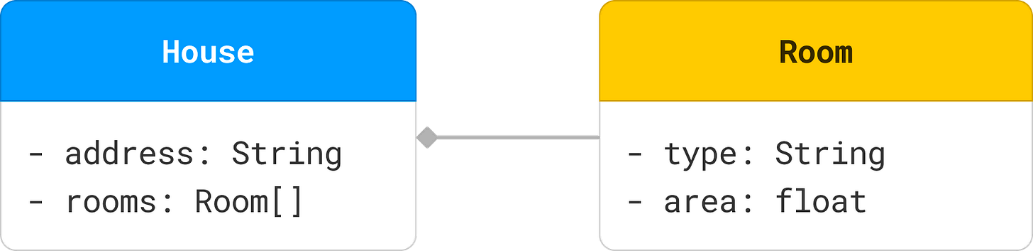
**2. Aggregation**

Aggregation represents a "has-a" relationship where one class (the whole) contains another class (the part), but the contained class can exist independently.



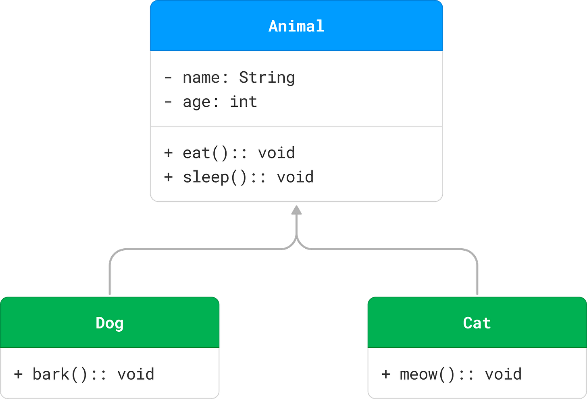
**3. Composition**

Composition represents a **strong** "has-a" relationship where the part cannot exist without the whole. If the whole is destroyed, the parts are also destroyed.



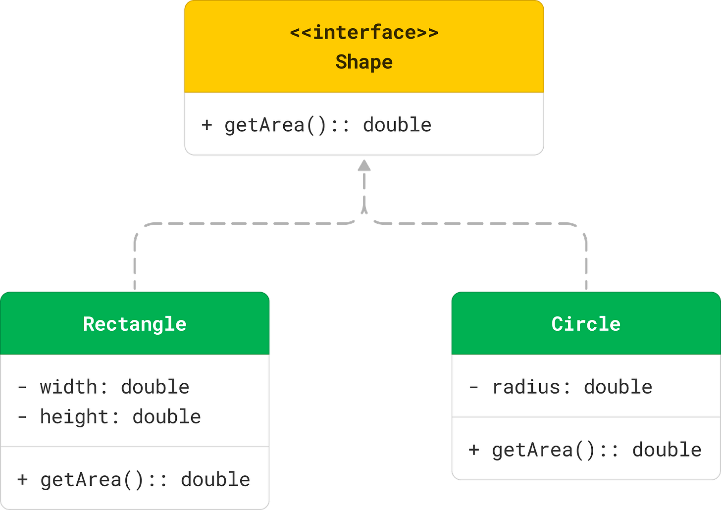
**4. Inheritance**

Inheritance (or Generalization) represents an "is-a" relationship where one class (subclass) inherits the attributes and methods of another class (superclass).



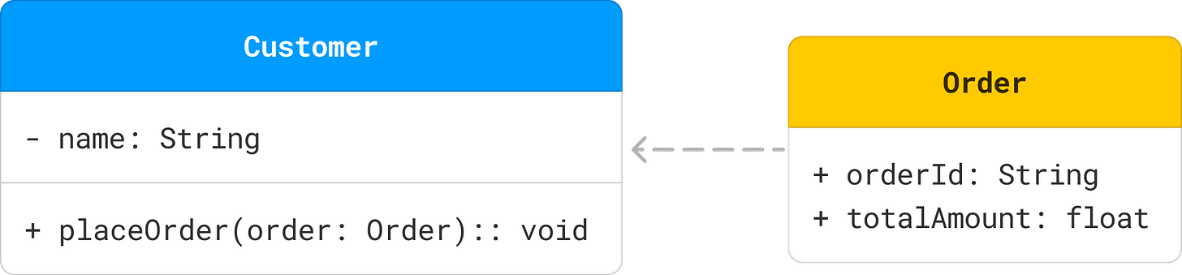
**5. Realization (Implementation)**

Realization or implementation represents a relationship between a class and an interface, where the class implements the methods declared in the interface.



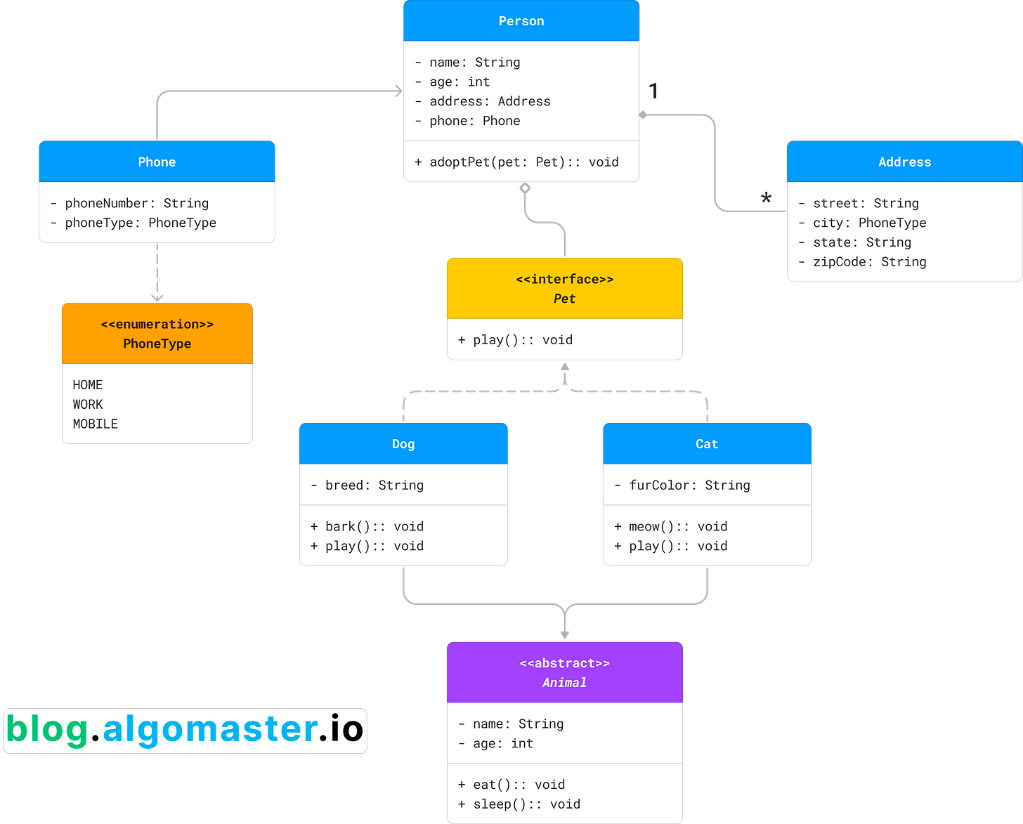
**6. Dependency**

Dependency represents a "uses" relationship where a change in one class (the supplier) may affect the other class (the client).



**Combined Example**

Here's a comprehensive example that includes various types of relationships:

[[](https://substackcdn.com/image/fetch/$s_!lMAf!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F72d3654b-44b5-4708-a634-527534bd0937_3772x3040.png)v](https://substackcdn.com/image/fetch/$s_!lMAf!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F72d3654b-44b5-4708-a634-527534bd0937_3772x3040.png" \t "_blank)

The relationships between the classes are as follows:

* **Inheritance:** Dog and Cat inherit from Animal.
* **Realization/Implementation:** Dog and Cat implement the Pet interface.
* **Aggregation:** Person has an aggregation relationship with Pet, indicating that a person can have multiple pets.
* **Composition:** Person has a composition relationship with Address, indicating that an address cannot exist without a person.
* **Association:** Person has an association relationship with Phone, indicating that a person can have multiple phone numbers.
* **Dependency:** Phone depends on the PhoneType enumeration for the phoneType attribute.

**NOTE:**

Among the six types of relationships, the code structure of composition, aggregation, and association is the same, and it can be understood from the strength of the relationship. The order from strong to weak is: **inheritance → implementation → composition → aggregation → association → dependency**

**Use Case Diagram**

A **Use Case Diagram** is a visual representation of how different users (also called **actors**) interact with a system.

They only show:

* Who is using the system
* What they want to do
* How the system responds to those actions

**Building Blocks of a Use Case Diagram**

**1. Actors**

An **actor** represents **anything that interacts with the system from outside**. Most often, actors are people (users), but they can also be external systems, sensors, or services.

There are two types:

* **Primary actors**: Initiates an interaction (e.g., a user logging in)
* **Secondary actors**: Helps fulfill a use case but don't initiate it (e.g., a payment gateway)

**2. Use Cases**

A **use case** is a **functionality or goal** that the system provides to the actor. Think of it as an action the user wants to perform.

Each use case should:

* Start with a verb (e.g., "Register", "Search", "Book Ticket", “Make Payment“)
* Represent a complete interaction from the user's point of view
* Deliver a meaningful result

**3. System Boundary**

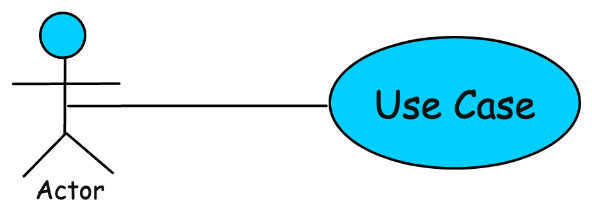
The **system boundary** defines **what’s inside the system and what’s outside**. This helps clearly define scope.

**Examples**: "Library Management System", "Online Banking System"

**4. Relationships**

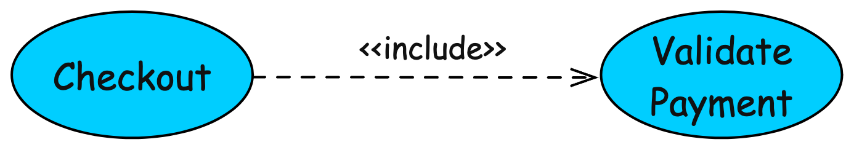
Relationships describe how actors and use cases are connected or how different use cases relate to one another. There are four main types:

**a. Association**

[[](https://substackcdn.com/image/fetch/$s_!5oRw!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F3b78ce1b-758b-4f76-9abe-525cce77387e_926x344.png)](https://substackcdn.com/image/fetch/$s_!5oRw!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F3b78ce1b-758b-4f76-9abe-525cce77387e_926x344.png" \t "_blank)

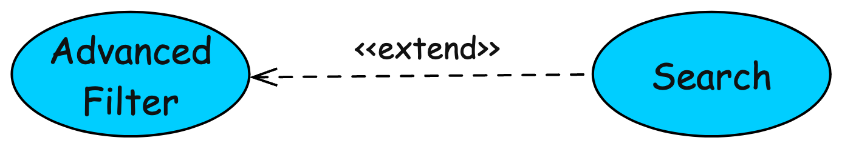
* Connects an actor to a use case
* Example: Customer → Place Order
* **Notation:** Represented by a solid line

**b. Include**

[](https://substackcdn.com/image/fetch/$s_!Rd2D!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F5d92afb2-bb17-4700-bdae-e0e733027f6a_1386x244.png)

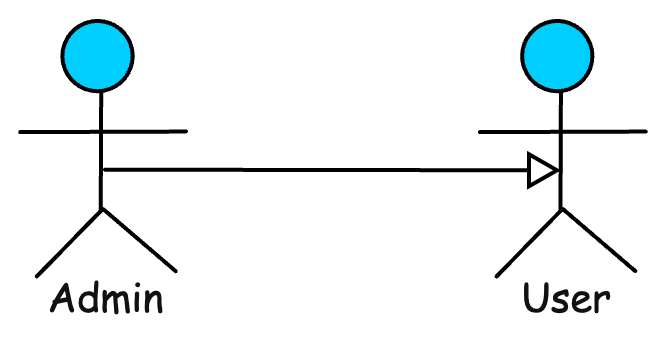
* Represents **common functionality** shared between use cases
* Example: Checkout includes Validate Payment
* Think of it as: "Always includes this"
* **Notation**: Dashed arrow with label <<include>>

**c. Extend**

[](https://substackcdn.com/image/fetch/$s_!oxs2!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F06df621f-2654-45b1-b336-350995b14eab_1386x246.png)

* Represents **optional or conditional behavior**
* Example: Search can extend to Advanced Filter
* Think of it as: "Sometimes adds this"
* **Notation**: Dashed arrow with label <<extend>>

**d. Generalization**

[](https://substackcdn.com/image/fetch/$s_!zjnG!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F72ed9b33-1eaa-4426-9d23-e3b542d58401_666x344.png)

* Shows **inheritance (**parent-child relationship**)** between actors or use cases
* The child actor/use case is an enhancement of the parent use case.
* Example: Admin is a specialized User
* **Notation**: Directed arrow with a triangle arrowhead from child to parent

**How to Draw a Use Case Diagram (Step-by-Step)**

Use case diagram for a **Movie Ticket Booking System** as we go along.

**Step 1: Identify Actors**

Start by identifying **who** will interact with your system. These could be:

For our example:

* **Customer**: A person who books and cancels tickets, browses movies.
* **Admin:** An individual manages movie listings and show schedules.
* **Payment Gateway**: An external system that processes payments.

**Step 2: Identify Use Cases**

Now, list out **what the actors want to do**. These are your use cases, the actions or goals the system should support.

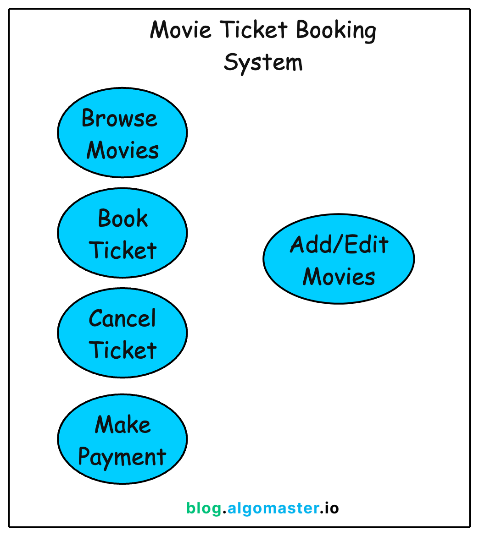
For the movie booking system:

* **Browse Movies:**Customers can browse the available movies.
* **Book Ticket:**Customers can book tickets.
* **Cancel Booking:**Customers can cancel tickets.
* **Make Payment:**Customers can make payments for their tickets.
* **Add/Edit Movie Listings:**Admins manage the movie listings.

**Step 3: Define the System Boundary**

Draw a rectangle and label it with the name of your system.

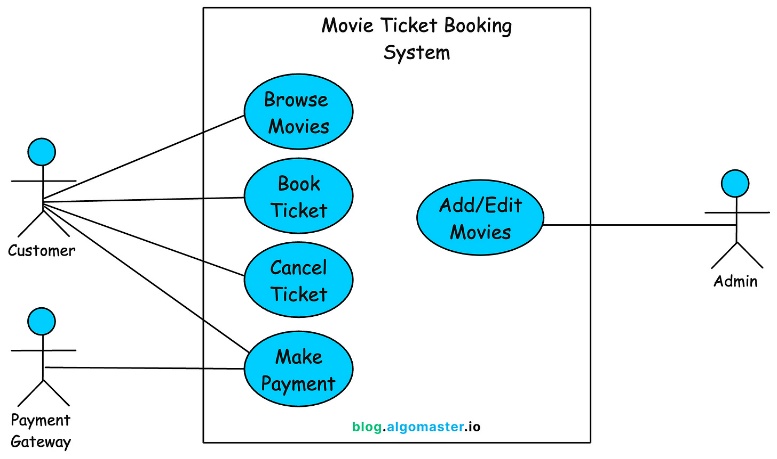
For our example: **Movie Ticket Booking System**

[[](https://substackcdn.com/image/fetch/$s_!2Q87!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F36541681-45ee-483e-b1bc-dff3de1c2e94_1044x1170.png)](https://substackcdn.com/image/fetch/$s_!2Q87!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F36541681-45ee-483e-b1bc-dff3de1c2e94_1044x1170.png" \t "_blank)

**Step 4: Connect Actors to Use Cases**

Now, link each actor to the relevant use cases using **solid lines** (associations).

* The **Customer** is connected to most of the features
* The **Admin** is only connected to the movie management use case
* The **Payment Gateway** interacts only with the payment flow

[[](https://substackcdn.com/image/fetch/$s_!heyz!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F83dafc80-b7e5-48c5-aa23-16fba9b7779a_2016x1194.png)](https://substackcdn.com/image/fetch/$s_!heyz!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F83dafc80-b7e5-48c5-aa23-16fba9b7779a_2016x1194.png" \t "_blank)

**Step 5: Model Relationships Between Use Cases**

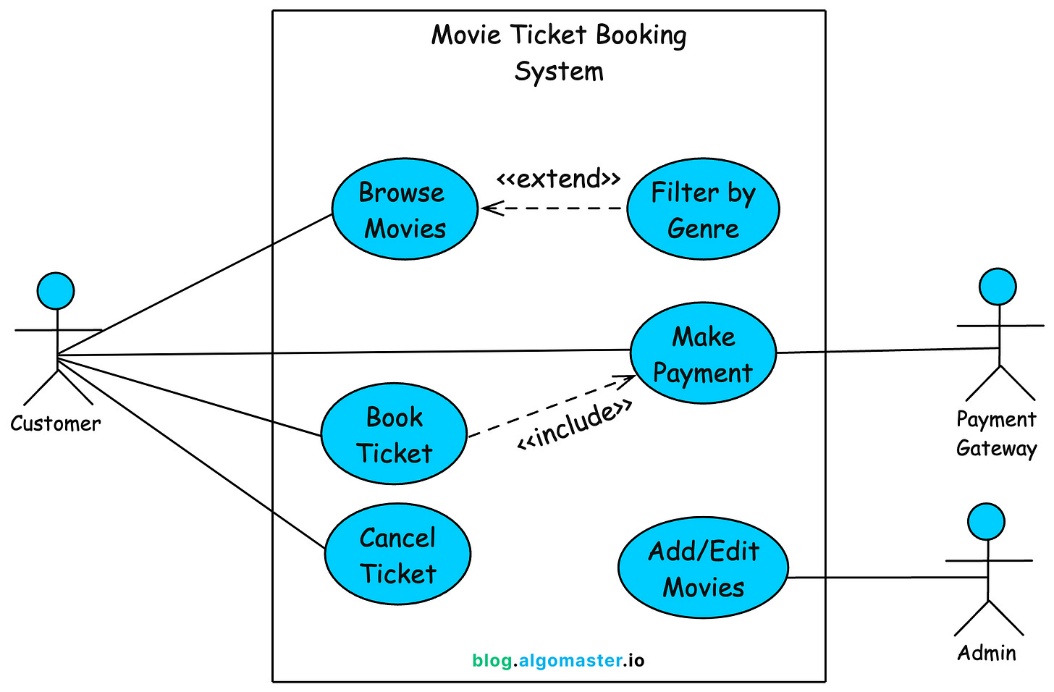
Use arrows and labels to show the relationships among use cases.

**a. Include :** Whenever someone books a ticket, they must make a payment.

**b. Extend :**While browsing movies, the user might choose to **filter by genre**—but it's not mandatory.

**c. Generalization :**A Registered User and Guest User both act like a Customer, but with slight differences. Use generalization to reflect that.

**Bringing It All Together**

[[](https://substackcdn.com/image/fetch/$s_!tbPS!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F6033a3a4-103b-4ba1-b374-cd593012b87c_2038x1338.png)](https://substackcdn.com/image/fetch/$s_!tbPS!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F6033a3a4-103b-4ba1-b374-cd593012b87c_2038x1338.png" \t "_blank)

**Sequence Diagrams**

Shows **how objects in a system interact with each other**, step by step.

These diagrams model:

* **Who** the participants (objects or actors) are
* **What** messages are exchanged
* **In what order** the messages occur
* **How long** each participant is active

**Building Blocks of a Sequence Diagram**

**Actors**

Actors are **external entities** (usually users or external systems) that **interact with your system**.

**Notation:**Stick figure labeled with the actor’s role**.**They’re shown on the **far left** and are usually the first to send a message.

**Objects / Participants**

Objects (also called participants or lifeline owners) are **instances of classes or components** in your system.

**Notation:** Each object will have a **vertical dashed line (lifeline)** extending downward to represent its activity over time.

**Lifelines**

A **lifeline** is a **dashed vertical line** drawn below each participant. It shows that the object exists during the interaction and represents the **flow of time**—top is the beginning, bottom is later.

**Activation Bars**

Activation bars (rectangles over lifelines) show when an object is **actively processing a message** or performing some task.

**Notation:**A thin rectangle on a lifeline

**Types of Messages in Sequence Diagrams**

They reflect **how different parts of your system communicate**—whether they wait for a response, fire-and-forget, or return data.

**1. Synchronous Message (→)**

A synchronous message is like a **phone call**. You ask a question and **wait** for the other person to answer before you move on.

* **Arrow Style:** Solid line with filled arrowhead →
* **Sender waits** for the receiver to finish
* **Used for:** method calls, API requests where a response is needed

**Example:**User calls login() on LoginController

**2. Asynchronous Message (→>)**

An asynchronous message is like **sending a text**—you don’t wait for a response. You just fire off the message and continue your own work.

* **Arrow Style:** Solid line with open arrowhead →>
* **Sender doesn’t wait** for a response
* **Used for:** background tasks, event notifications, message queues

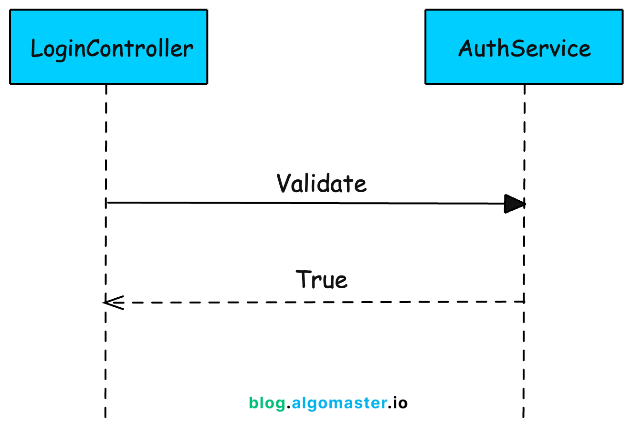
**Example:**Send a welcome email after successful registration.

**3. Return Message (←--)**

Return messages indicate that the receiver is **sending a response back** to the sender. Think of it like the **reply you get after asking a question**.

* **Arrow Style:** **Dashed** line with open arrowhead ←--
* Usually follows a synchronous message

**Example:**

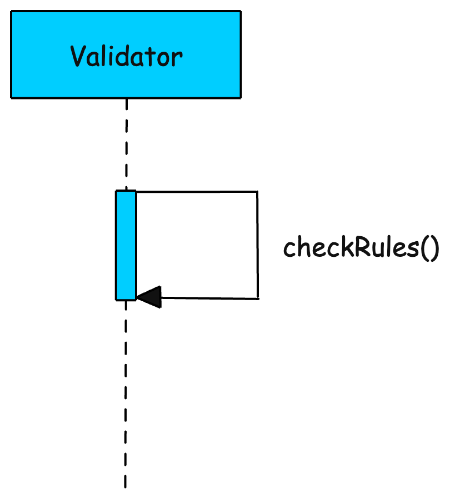
[[](https://substackcdn.com/image/fetch/$s_!G-A6!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F2f630b75-2180-4cd2-9b0e-b805f33f4068_1324x904.png)](https://substackcdn.com/image/fetch/$s_!G-A6!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F2f630b75-2180-4cd2-9b0e-b805f33f4068_1324x904.png" \t "_blank)

**4. Self-Message (Recursive Call)**

Sometimes, an object needs to talk to itself. This is shown using a **looped arrow** that points back to the same lifeline.

* **Used for:** recursive functions, internal helper method calls

**Example:**

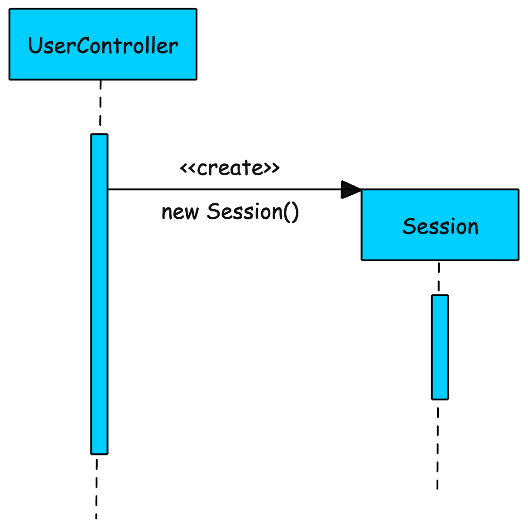
[[](https://substackcdn.com/image/fetch/$s_!iFVm!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fefbd323a-3541-4e2e-b5ae-bce9cb3b03bc_810x896.png)](https://substackcdn.com/image/fetch/$s_!iFVm!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fefbd323a-3541-4e2e-b5ae-bce9cb3b03bc_810x896.png" \t "_blank)

**5. Create Message**

When a message **creates a new object**, it’s called a **create message**.

* Typically ends with the **new object's lifeline starting**
* Arrow points to the object's head

**Example:**Creating a new Session object.

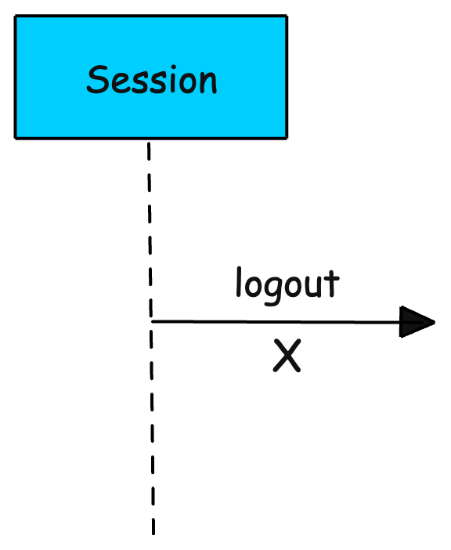
[[](https://substackcdn.com/image/fetch/$s_!8sBf!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fb21dcc34-5342-4b0a-8b18-b136a93156cc_1166x1166.png)](https://substackcdn.com/image/fetch/$s_!8sBf!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fb21dcc34-5342-4b0a-8b18-b136a93156cc_1166x1166.png" \t "_blank)

**6. Destroy Message (Optional)**

Sometimes you want to indicate that an object is destroyed after a certain point—like closing a file or deleting a session.

* Marked with an **'X'** at the end of the lifeline
* Used rarely, but useful for **resource cleanup** and **lifecycle clarity**

**Example:**

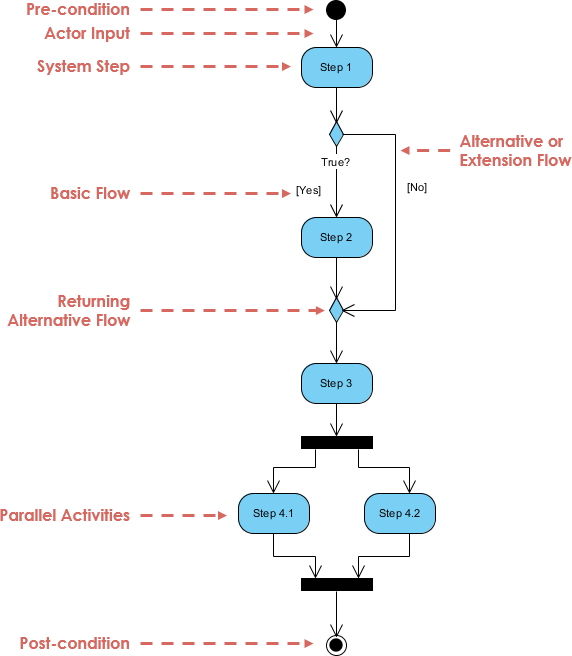
[[](https://substackcdn.com/image/fetch/$s_!38_I!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fe1b45414-f7b3-4af0-8fe8-342ada4427bb_576x702.png)](https://substackcdn.com/image/fetch/$s_!38_I!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fe1b45414-f7b3-4af0-8fe8-342ada4427bb_576x702.png" \t "_blank)

**Activity Diagram**

Describe dynamic aspects of the system. Activity diagram is essentially an advanced version of flow chart that modeling the flow from one activity to another activity.

**Activity Diagram - Learn by Examples**

A basic activity diagram - flowchart like



**Activity Diagram Notation Summary**

1. **Activity**

Is used to represent a set of actions



1. **Action**

A task to be performed

Activity Diagram Notation - Action

1. **Control Flow**

Shows the sequence of execution

Activity Diagram Notation - Control Flow

1. **Object Flow**

Show the flow of an object from one activity (or action) to another activity (or action).

Activity Diagram Notation - Object Flow

1. **Initial Node**

Portrays the beginning of a set of actions or activities



1. **Activity Final Node**

Stop all control flows and object flows in an activity (or action)



1. **Object Node**

Represent an object that is connected to a set of Object Flows

Activity Diagram Notation - Object Node

1. **Decision Node**

Represent a test condition to ensure that the control flow or object flow only goes down one path

Activity Diagram Notation - Decision Node

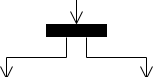
1. **Merge Node**

Bring back together different decision paths that were created using a decision-node

Activity Diagram Notation - Merge Node

1. **Fork Node**

Split behavior into a set of parallel or concurrent flows of activities (or actions)



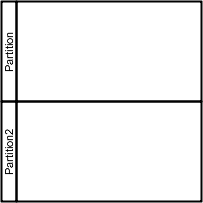
1. **Join Node**

Bring back together a set of parallel or concurrent flows of activities (or actions).

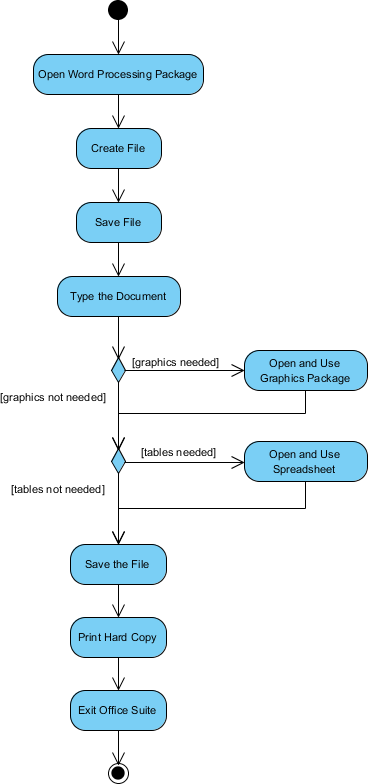
Activity Diagram Notation - Join Node

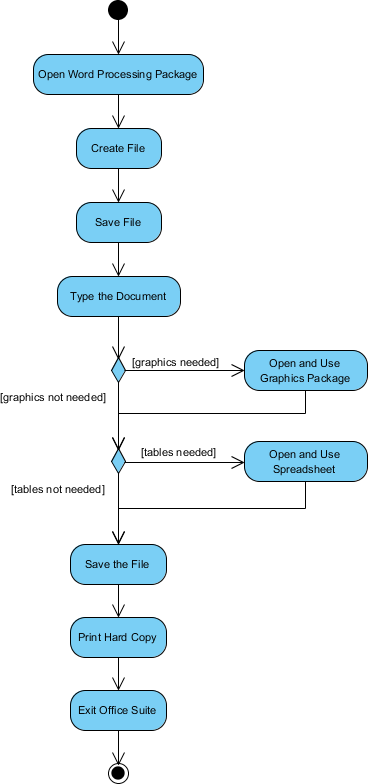
1. **Swimlane and Partition**

A way to group activities performed by the same actor on an activity diagram or to group activities in a single thread



**Activity Diagram - Modeling a Word Processor**



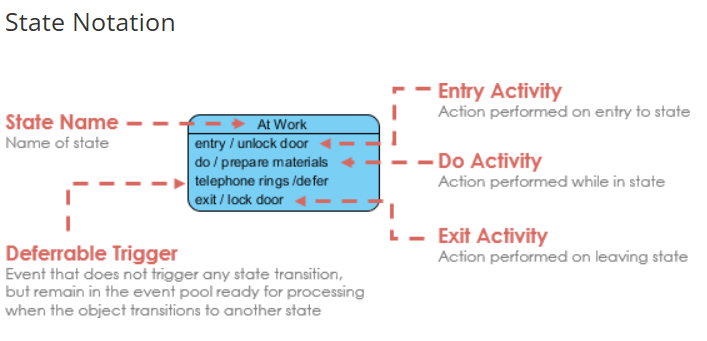


**State Machine Diagram**

 An object responds differently to the same event depending on what state it is in.

**State**

A state is an abstraction of the attribute values and links of an object.



The **initial state** of a state machine diagram, known as an initial pseudo-state, is indicated with a solid circle.

The **final state** of a state machine diagram is shown as concentric circles.

**Events**

An event signature is described as Event-name (comma-separated-parameter-list).

Events appear in the internal transition compartment of a state or on a transition between states.

An event may be one of four types:

1. Signal event - corresponding to the arrival of an asynchronous message or signal
2. Call event - corresponding to the arrival of a procedural call to an operation
3. Time event - a time event occurs after a specified time has elapsed
4. Change event - a change event occurs whenever a specified condition is met

**Transition**

Transition lines depict the movement from one state to another. Each transition line is labeled with the **event** that causes the transition.

**Actions**

Action is an executable atomic computation, which includes operation calls, the creation or destruction of another object, or the sending of a signal to an object.

**Activity**

Activity is associated with states, which is a non-atomic or ongoing computation.

**Entry and Exit Actions**

Entry and Exit actions specified in the state. It must be true for every entry / exit occurrence. If not, then you must use actions on the individual transition arcs

* **Entry Action** executed on entry into state with the **notation: Entry / action**
* **Exit Action** executed on exit from state with the **notation: Exit / action**

**Substates**

A simple state is one which has no substructure. A state which has substates (nested states) is called a composite state. Substates may be nested to any level.

**History States**

Unless otherwise specified, when a transition enters a composite state, the action of **the nested state machine starts over again at the initial state** (unless the transition targets a substate directly).

**Concurrent State**

As mentioned above, states in state machine diagrams can be nested. Related states can be grouped together into a single composite state. Nesting states inside others is necessary when an activity involves concurrent sub-activities.