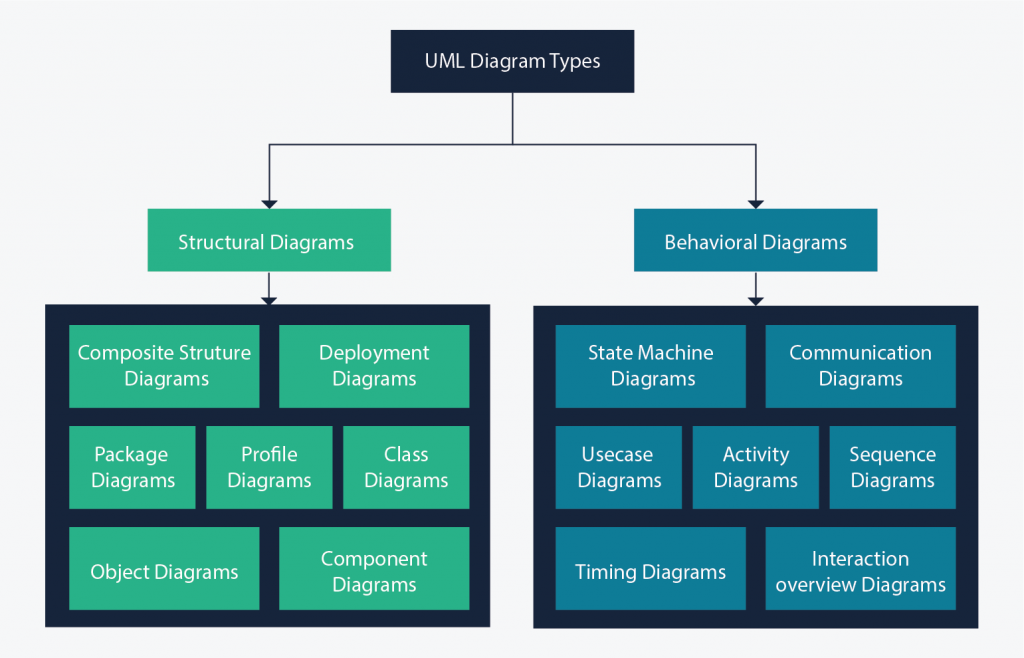
**UML(Unified Modelling Language)**

UML (Unified Modeling Language) is a standardized modeling language used to visually represent the design and architecture of software systems. It provides a set of diagrams to represent various aspects of a system, including its structure, behavior, and interactions.

Here’s an overview of the **basic UML diagram types:**





**1. Class Diagram**

Class diagrams represent the static structure of a system by showing the system's classes, their attributes, methods, and the relationships between the classes (such as inheritance, association, and aggregation).

*In these diagrams, classes are depicted as boxes, each containing three compartments for the class name, attributes, and methods. Lines connecting classes illustrate associations, showing relationships such as one-to-one or one-to-many.*

**Purpose of Class Diagrams**

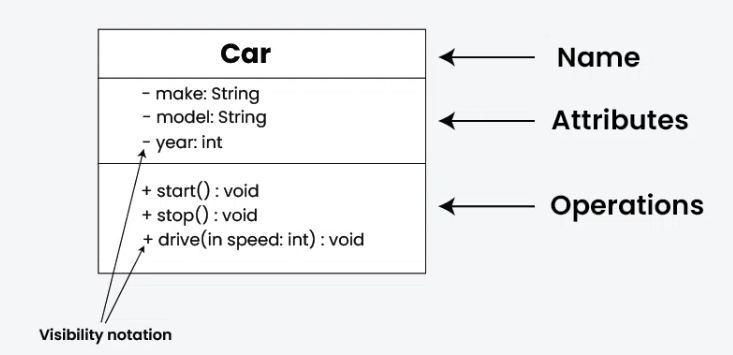
The main purpose of using class diagrams is:

* This is the only UML that can appropriately depict various aspects of the OOPs concept.
* Proper design and analysis of applications can be faster and efficient.
* It is the base for deployment and component diagram.
* It incorporates forward and reverse engineering.

**UML Class Notation**

Class notation is a graphical representation used to depict classes and their relationships in object-oriented modeling.

**Example:**

****

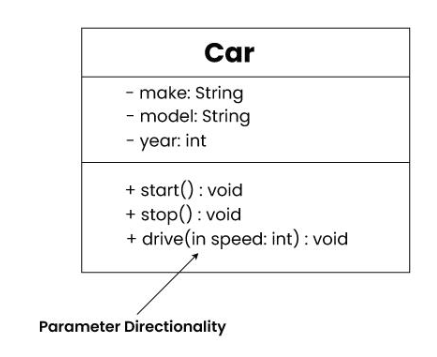
1. **Class Name:**
   * The name of the class is typically written in the top compartment of the class box and is centered and bold.
2. **Attributes:**
   * Attributes, also known as properties or fields, represent the data members of the class. They are listed in the second compartment of the class box and often include the visibility (e.g., public, private) and the data type of each attribute.
3. **Methods:**
   * Methods, also known as functions or operations, represent the behavior or functionality of the class. They are listed in the third compartment of the class box and include the visibility (e.g., public, private), return type, and parameters of each method.
4. **Visibility Notation:**

Visibility notations indicate the access level of attributes and methods. Common visibility notations include:

* + + for public (visible to all classes)
  + - for private (visible only within the class)
  + # for protected (visible to subclasses)
  + ~ for package or default visibility (visible to classes in the same package)

**Parameter Directionality**

In class diagrams, parameter directionality refers to the indication of the flow of information between classes through method parameters. It helps to specify whether a parameter is an input, an output, or both. This information is crucial for understanding how data is passed between objects during method calls.

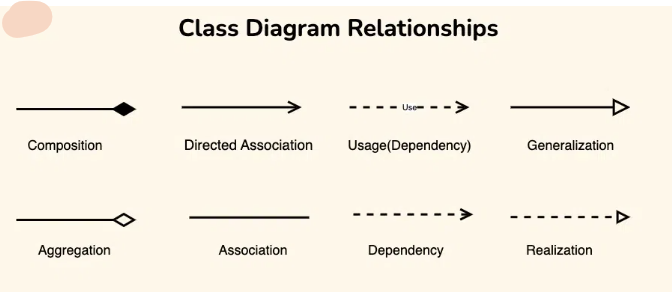


There are three main parameter directionality notations used in class diagrams:

* **In (Input):**
  + An input parameter is a parameter passed from the calling object (client) to the called object (server) during a method invocation.
  + It is represented by an arrow pointing towards the receiving class (the class that owns the method).
* **Out (Output):**
  + An output parameter is a parameter passed from the called object (server) back to the calling object (client) after the method execution.
  + It is represented by an arrow pointing away from the receiving class.
* **InOut (Input and Output):**
  + An InOut parameter serves as both input and output. It carries information from the calling object to the called object and vice versa.
  + It is represented by an arrow pointing towards and away from the receiving class.

**Relationships between classes**

In class diagrams, relationships between classes describe how classes are connected or interact with each other within a system. There are several types of relationships in object-oriented modeling, each serving a specific purpose. Here are some common types of relationships in class diagrams:



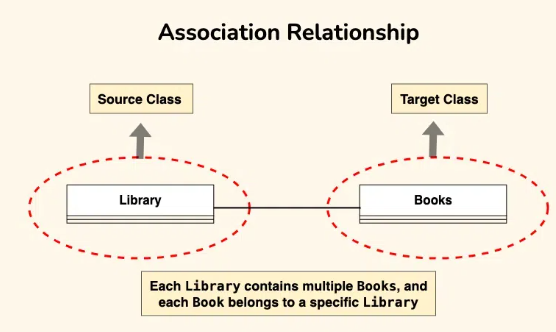
**1. Association**

An association represents a bi-directional relationship between two classes. It indicates that instances of one class are connected to instances of another class. Associations are typically depicted as a solid line connecting the classes, with optional arrows indicating the direction of the relationship.

**Example:**

*Let’s consider a simple system for managing a library. In this system, we have two main entities: Book and Library. Each Library contains multiple Books, and each Book belongs to a specific Library. This relationship between Library and Book represents an association.*

The “Library” class can be considered the source class because it contains a reference to multiple instances of the “Book” class. The “Book” class would be considered the target class because it belongs to a specific library.



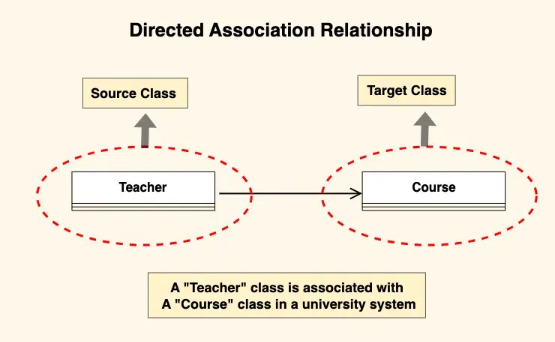
**2. Directed Association**

A directed association in a UML class diagram represents a relationship between two classes where the association has a direction, indicating that one class is associated with another in a specific way.

* In a directed association, an arrowhead is added to the association line to indicate the direction of the relationship. The arrow points from the class that initiates the association to the class that is being targeted or affected by the association.
* Directed associations are used when the association has a specific flow or directionality, such as indicating which class is responsible for initiating the association or which class has a dependency on another.

*Consider a scenario where a “Teacher” class is associated with a “Course” class in a university system. The directed association arrow may point from the “Teacher” class to the “Course” class, indicating that a teacher is associated with or teaches a specific course.*

* The source class is the “Teacher” class. The “Teacher” class initiates the association by teaching a specific course.
* The target class is the “Course” class. The “Course” class is affected by the association as it is being taught by a specific teacher

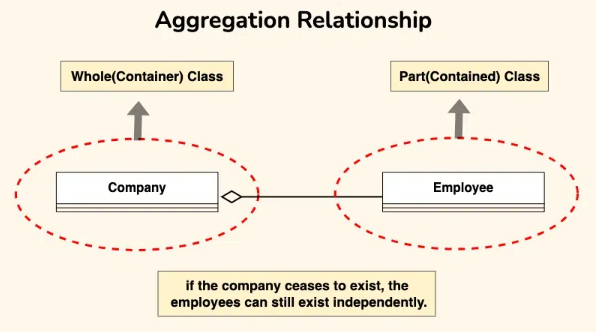


**3. Aggregation**

Aggregation is a specialized form of association that represents a “whole-part” relationship. It denotes a stronger relationship where one class (the whole) contains or is composed of another class (the part). 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.

Let’s understand aggregation using an example:

*The company can be considered as the whole, while the employees are the parts. Employees belong to the company, and the company can have multiple employees. However, if the company ceases to exist, the employees can still exist independently.*

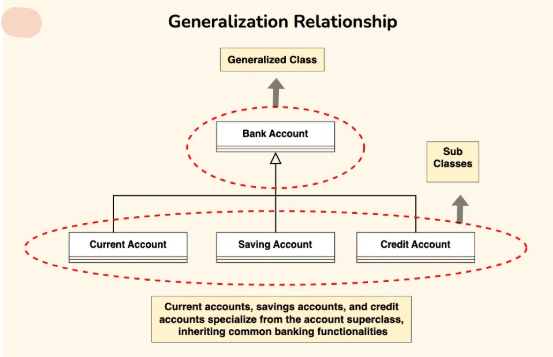


**4. Generalization (Inheritance)**

Inheritance represents an “is-a” relationship between classes, where one class (the subclass or child) inherits the properties and behaviors of another class (the superclass or parent). Inheritance is depicted by a solid line with a closed, hollow arrowhead pointing from the subclass to the superclass.

*In the example of bank accounts, we can use generalization to represent different types of accounts such as current accounts, savings accounts, and credit accounts.*

The Bank Account class serves as the generalized representation of all types of bank accounts, while the subclasses (Current Account, Savings Account, Credit Account) represent specialized versions that inherit and extend the functionality of the base class.



**Benefits of Class Diagrams**

Below are the benefits of class diagrams:

* Class diagrams represent the system’s classes, attributes, methods, and relationships, providing a clear view of its architecture.
* They shows various relationships between classes, such as associations and inheritance, helping stakeholders understand component connectivity.
* Class diagrams serve as a visual tool for communication among team members and stakeholders, bridging gaps between technical and non-technical audiences.
* They guide developers in coding by illustrating the design, ensuring consistency between the design and actual implementation.
* Many development tools allow for code generation from class diagrams, reducing manual errors and saving time.

**How to draw Class Diagrams**

Drawing class diagrams involves visualizing the structure of a system, including classes, their attributes, methods, and relationships. Here are the steps to draw class diagrams:

* **Step 1: Identify Classes:**
  + Start by identifying the classes in your system. A class represents a blueprint for objects and should encapsulate related attributes and methods.
* **Step 2: List Attributes and Methods:**
  + For each class, list its attributes (properties, fields) and methods (functions, operations). Include information such as data types and visibility (public, private, protected).
* **Step 3: Identify Relationships:**
  + Determine the relationships between classes. Common relationships include associations, aggregations, compositions, inheritance, and dependencies. Understand the nature and multiplicity of these relationships.
* **Step 4: Create Class Boxes:**
  + Draw a rectangle (class box) for each class identified. Place the class name in the top compartment of the box. Divide the box into compartments for attributes and methods.
* **Step 5: Add Attributes and Methods:**
  + Inside each class box, list the attributes and methods in their respective compartments. Use visibility notations (+ for public, – for private, # for protected, ~ for package/default).
* **Step 6: Draw Relationships:**
  + Draw lines to represent relationships between classes. Use arrows to indicate the direction of associations or dependencies. Different line types or notations may be used for various relationships.
* **Step 7: Label Relationships:**
  + Label the relationships with multiplicity and role names if needed. Multiplicity indicates the number of instances involved in the relationship, and role names clarify the role of each class in the relationship.
* **Step 8: Review and Refine:**
  + Review your class diagram to ensure it accurately represents the system’s structure and relationships. Refine the diagram as needed based on feedback and requirements.

**Object Diagrams**

In Java and Object-Oriented Programming (OOP), object diagrams visually represent the structure and relationships between objects at a specific moment in time, particularly showing how different objects are connected to each other and the values they hold. These diagrams are often used to model and understand the dynamic aspects of a system, especially for illustrating how objects interact at runtime.

While class diagrams define the blueprint of the system (i.e., the structure of classes, methods, and relationships), object diagrams focus on showing specific instances (objects) of those classes and their relationships at a particular point in time.

**1. What are Object Diagrams?**

An Object Diagram can be referred to as a screenshot of the instances in a system and the relationship that exists between them.

* An object diagram in UML is useful because it provides a clear and visual representation of specific instances of classes and their relationships at a particular point in time, aiding in understanding and communicating the structure and interactions within a system.

*Object diagrams in UML are depicted using a simple and intuitive notations to show a snapshot of a system at a specific point in time, displaying instances of classes and their relationships.*

**What is an Object?**

An object refers to a specific instance of a class within a system. A class is a blueprint or template that defines the common attributes and behaviors shared by a group of objects. An object, on the other hand, is a concrete and individual occurrence of that class, possessing unique values for its attributes.

**What is a Classifier?**

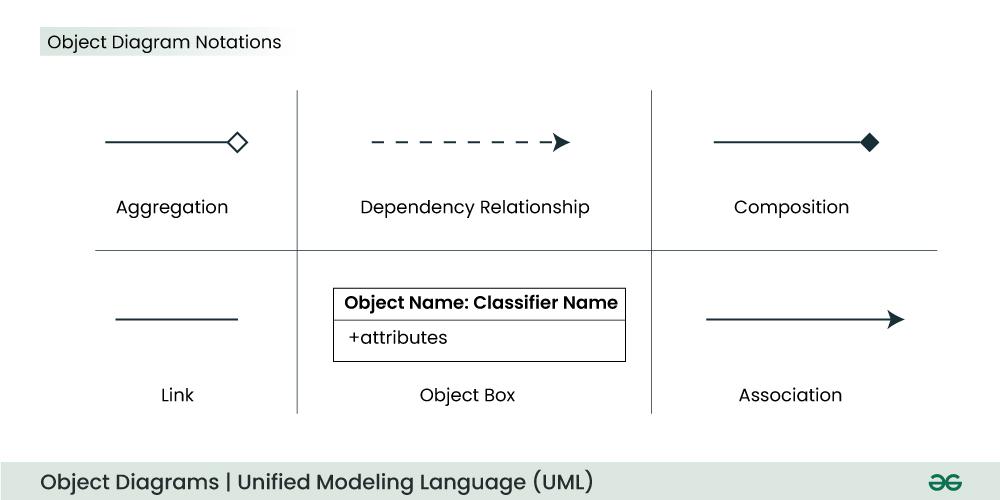
In UML a classifier refers to a group of elements that have some common features like methods, attributes and operations. A classifier can be thought of as an abstract metaclass which draws a boundary for a group of instances having common static and dynamic features.

**Example:**

*we refer a class, an object, a component, or a deployment node as classifiers in UML since they define a common set of properties. We are able to design object diagrams by instantiating classifiers.*

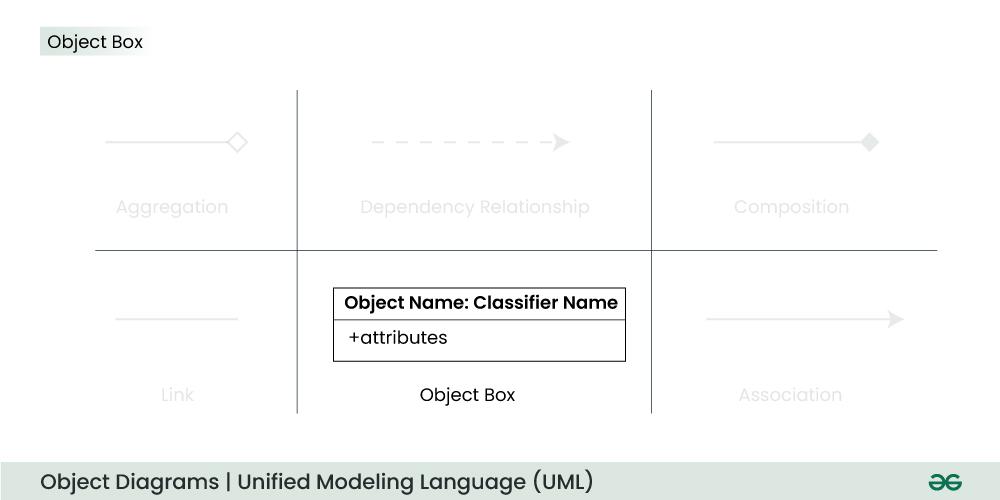
**Object Diagram Notations**

The object diagram in UML uses specific notations to represent instances of classes and their relationships at a particular moment in time.



**1. Objects or Instance specifications**

When we instantiate a classifier in a system, the object we create represents an entity which exists in the system. We can represent the changes in object over time by creating multiple instance specifications. We use a rectangle to represent an object in an object diagram.

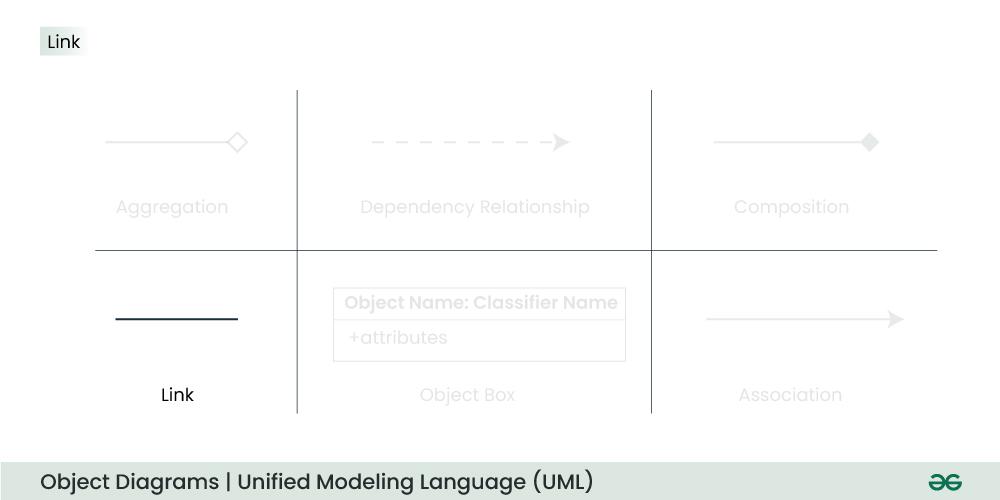


**2. Attributes and Values**

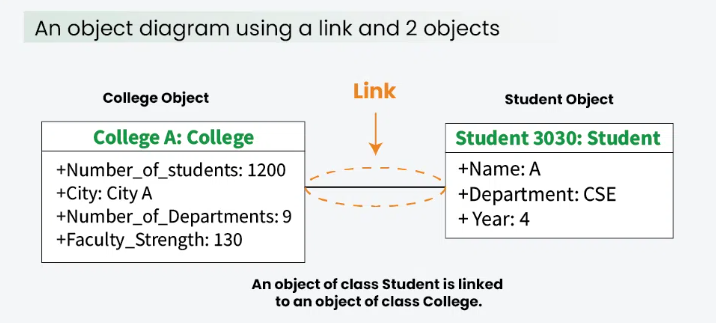
Inside the object box, attributes of the object are listed along with their specific values.

**3. Link**

We use a link to represent a relationship between two objects. We represent the number of participants on the link for each, at the end of the link. The term link is used to specify a relationship between two instance specifications or objects. We use a solid line to represent a link between two objects.



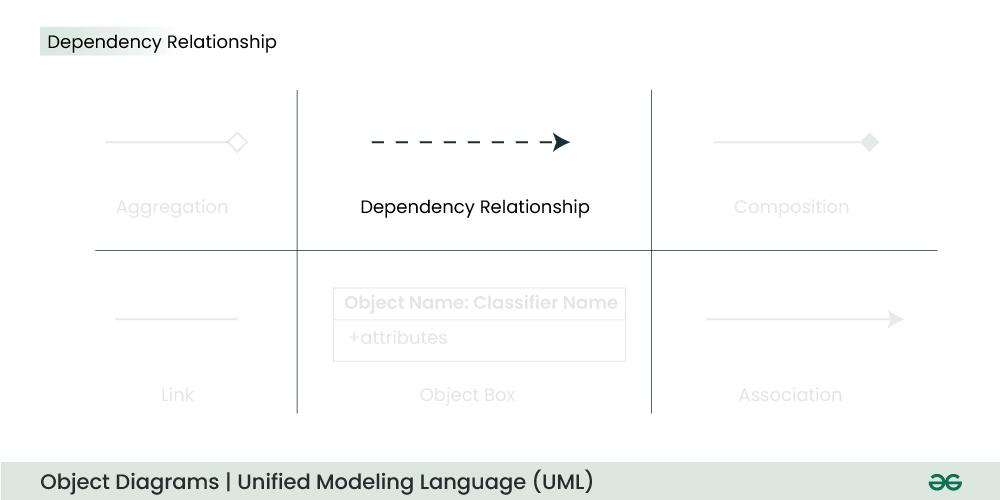
***For Example*** *– In the figure below, an object of class Student is linked to an object of class College.*



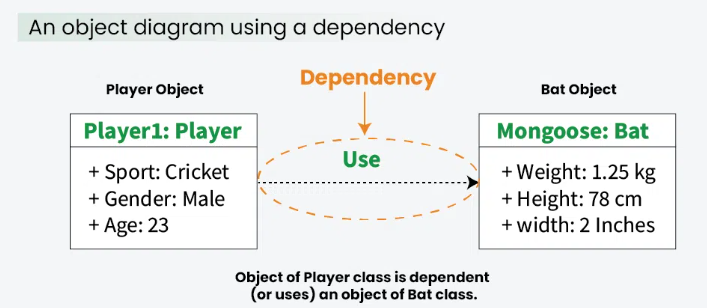
**4. Dependency Relationships**

We use a dependency relationship to show when one element depends on another element. A dependency is used to depict the relationship between dependent and independent entities in the system.

* Any change in the definition or structure of one element may cause changes to the other.
* This is a unidirectional kind of relationship between two objects.
* Dependency relationships are of various types specified with keywords like Abstraction, Binding, Realization, Substitution and Usage are the types of dependency relationships used in UML.

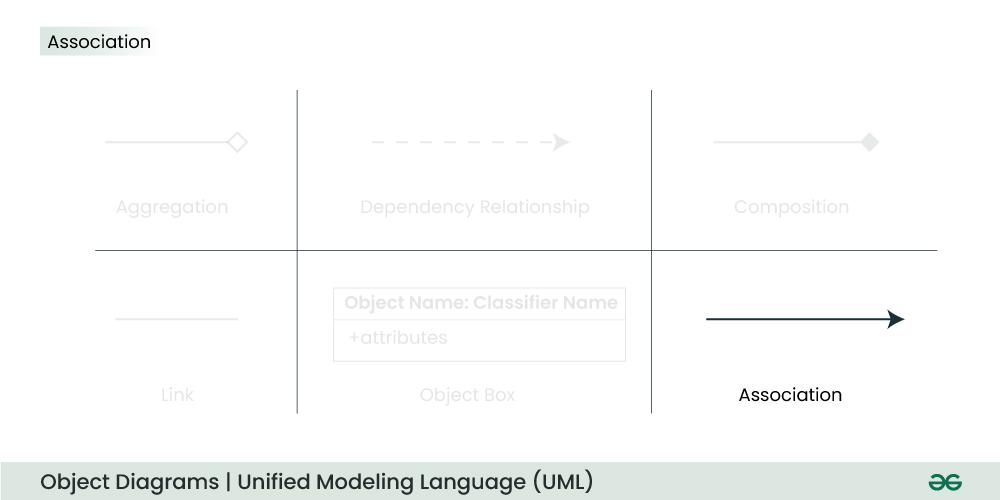


***For example –*** *In the figure below, an object of Player class is dependent (or uses) an object of Bat class.*

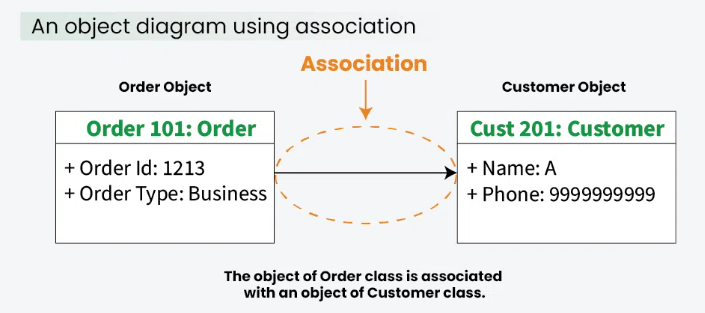


**5.Association**

Association is a reference relationship between two objects (or classes). An association line connects two object boxes, representing a relationship between instances of two classes. We use association when one object references members of the other object. Association can be uni-directional or bi-directional. We use an arrow to represent association.



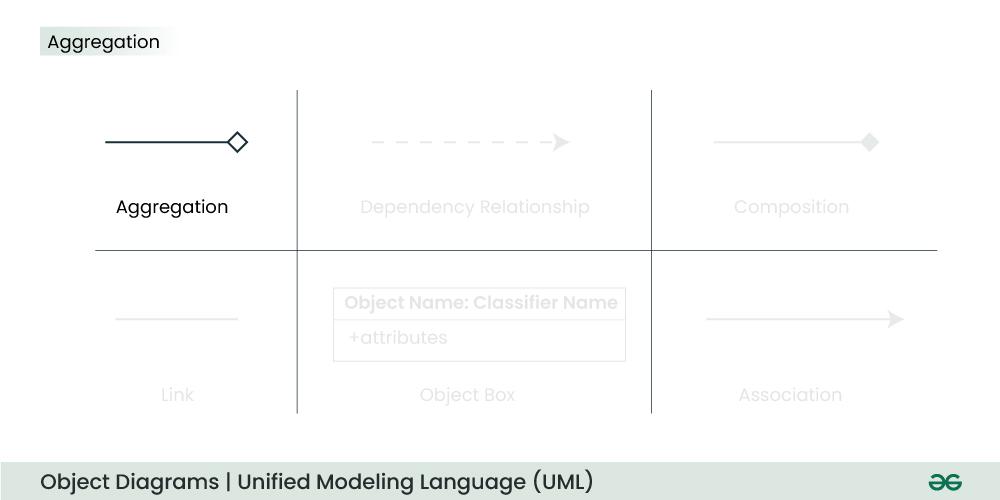
***For example –*** *The object of Order class is associated with an object of Customer class.*

**

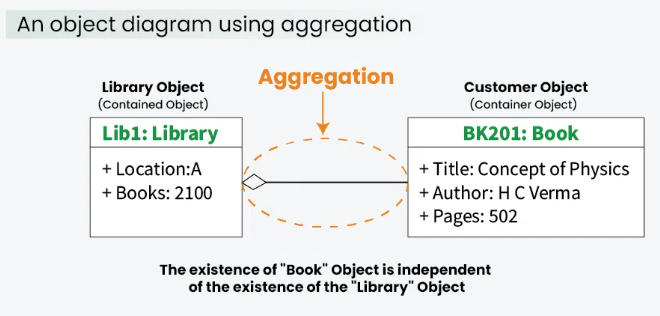
**6. Aggregation**

Aggregation represents a “has a” relationship. We use a hollow diamond on the containing object with a line which joins it to the contained object.

* Aggregation is a specific form of association.
* It is a kind of parent-child relationship however it isn’t inheritance.
* Aggregation occurs when the lifecycle of the contained objects does not strongly depend on the lifecycle of container objects.



***For example –*** *A library has an aggregation relationship with books. Library has books or books are a part of library. The existence of books is independent of the existence of the library.*



**Purpose of Object Diagrams**

The main purpose of using object diagrams is:

* They offer a detailed view of how objects interact with each other in specific scenarios.
* Proper design and analysis of applications can be faster and efficient.
* Object diagrams are beneficial during the implementation phase of software development.
* Promoting a shared understanding of specific instances and their relationships, facilitating collaboration among team members.

**How to draw an Object Diagram?**

1. **Identify Classes:** Determine the classes relevant to the scenario you want to depict. Classes are the blueprints that define the attributes and behaviors shared by their instances.
2. **Identify Objects:** Identify specific instances or objects of each class that you want to include in the diagram. These represent the actual things in your system.
3. **Create Object Boxes:** Draw rectangles to represent the specific instances or objects of each class. Write the name of each object inside the box.
4. **Add Attributes and Values:** Inside each object box, list the attributes of that object along with their specific values.
5. **Draw Relationships:** Connect the object boxes with lines to represent relationships or associations between instances. Use arrows to indicate the direction of the association if necessary.
6. **Label Relationships:** Label the relationships with multiplicity and role names if needed. Label the association lines with a verb or phrase to describe the nature of the relationship.
7. **Review and Refine:** Review your Object diagram to ensure it accurately represents the system’s structure and relationships. Refine the diagram as needed based on feedback and requirements.
8. **Use Tools for Digital Drawing:** While you can draw class diagrams on paper, using digital tools can provide more flexibility and ease of modification. UML modeling tools, drawing software, or even specialized diagramming tools can be helpful.

**Use Case Diagram**

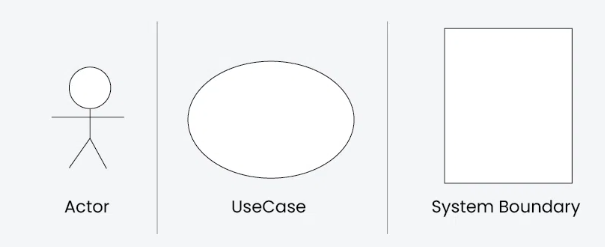
Use case diagrams are part of **Unified Modeling Language (UML)**, which is widely used for modeling software systems. These diagrams represent the functional requirements of a system by showing the interactions between the system and external entities (users or other systems). They help visualize what the system will do without going into the details of how it will be done.

**Basic Components of Use Case Diagrams:**

1. **Actors**: These are the external entities (users, systems, or hardware) that interact with the system. An actor is typically represented by a stick figure.
2. **Use Cases**: These represent the specific functionality or behavior that the system will provide. They are usually depicted as ovals.
3. **System Boundary**: This defines the boundary between the system being designed and the outside world. It is often drawn as a rectangle that contains all the use cases.
4. **Relationships**: These represent interactions between actors and use cases, or between different use cases. Some common relationships include:
   * **Association**: A solid line connecting an actor to a use case, representing an interaction.
   * **Include**: A dotted line with an arrow that indicates that a use case includes the behavior of another use case.
   * **Extend**: A dotted line with an arrow that indicates that a use case can extend the behavior of another use case under certain conditions.

It typically represents the interaction between actors (users or external systems) and a system under consideration to accomplish specific goals. It provides a high-level view of the system's functionality by illustrating the various ways users can interact with it.

**Use Case Diagram Notations:**



**1. Actors**

They are represented by stick figures.

**2. Use Cases**

Use cases are like scenes in the play. They represent specific things your system can do. In the online shopping system, examples of use cases could be "Place Order," "Track Delivery," or "Update Product Information".Use cases are represented by ovals.

**3. System Boundary**

The system boundary is a visual representation of the scope or limits of the system you are modeling. It defines what is inside the system and what is outside. The boundary helps to establish a clear distinction between the elements that are part of the system and those that are external to it. The system boundary is typically represented by a rectangular box that surrounds all the use cases of the system.

*The purpose of system boundary is to clearly outlines the boundaries of the system, indicating which components are internal to the system and which are external actors or entities interacting with the system.*

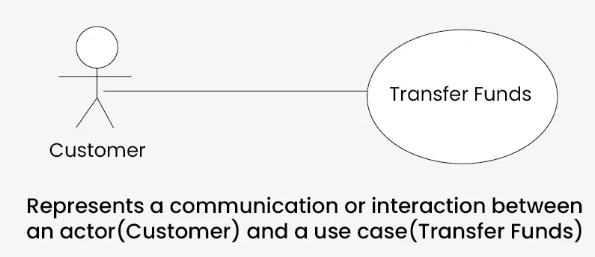
**4. Relationships**

**1. Association Relationship**

TheAssociation Relationship represents a communication or interaction between an actor and a use case. It is depicted by a line connecting the actor to the use case. This relationship signifies that the actor is involved in the functionality described by the use case.

***Example: Online Banking System***

* **Actor:** Customer
* **Use Case:** Transfer Funds
* **Association:** A line connecting the "Customer" actor to the "Transfer Funds" use case, indicating the customer's involvement in the funds transfer process.

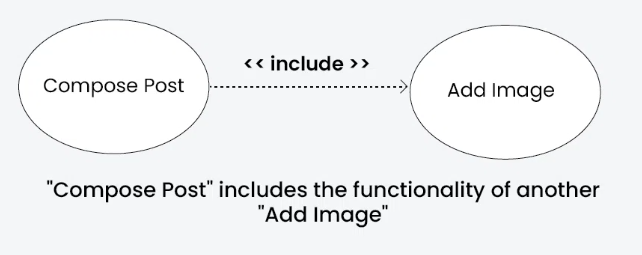


**2. Include Relationship**

The Include Relationship indicates that a use case includes the functionality of another use case. It is denoted by a dashed arrow pointing from the including use case to the included use case. This relationship promotes modular and reusable design.

***Example: Social Media Posting***

* **Use Cases:** Compose Post, Add Image
* **Include Relationship:** The "Compose Post" use case includes the functionality of "Add Image." Therefore, composing a post includes the action of adding an image.

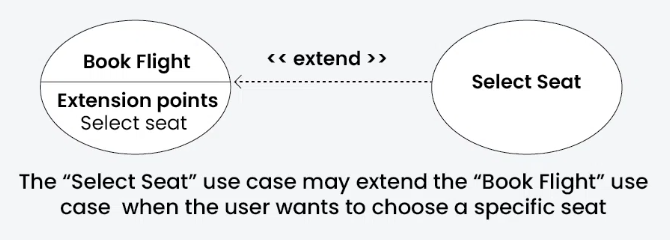


**3. Extend Relationship**

The Extend Relationship illustrates that a use case can be extended by another use case under specific conditions. It is represented by a dashed arrow with the keyword "extend." This relationship is useful for handling optional or exceptional behavior.

***Example: Flight Booking System***

* **Use Cases:** Book Flight, Select Seat
* **Extend Relationship:** The "Select Seat" use case may extend the "Book Flight" use case when the user wants to choose a specific seat, but it is an optional step.

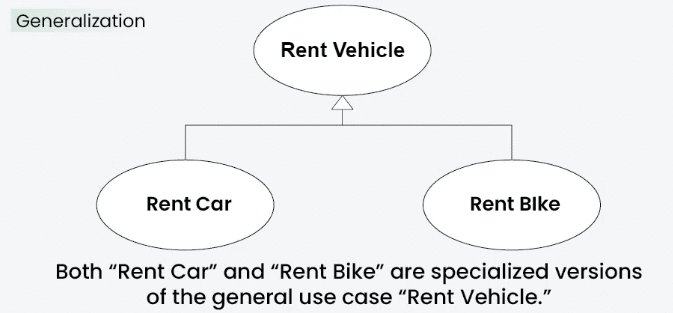


**4. Generalization Relationship**

The Generalization Relationship establishes an "is-a" connection between two use cases, indicating that one use case is a specialized version of another. It is represented by an arrow pointing from the specialized use case to the general use case.

***Example: Vehicle Rental System***

* **Use Cases:** Rent Car, Rent Bike
* **Generalization Relationship:** Both "Rent Car" and "Rent Bike" are specialized versions of the general use case "Rent Vehicle."



**How to draw a Use Case diagram in UML?**

Below are the main steps to draw use case diagram in UML:

* **Step 1: Identify Actors**: Determine who or what interacts with the system. These are your actors. They can be users, other systems, or external entities.
* **Step 2: Identify Use Cases**: Identify the main functionalities or actions the system must perform. These are your use cases. Each use case should represent a specific piece of functionality.
* **Step 3: Connect Actors and Use Cases**: Draw lines (associations) between actors and the use cases they are involved in. This represents the interactions between actors and the system.
* **Step 4: Add System Boundary**: Draw a box around the actors and use cases to represent the system boundary. This defines the scope of your system.
* **Step 5: Define Relationships**: If certain use cases are related or if one use case is an extension of another, you can indicate these relationships with appropriate notations.
* **Step 6: Review and Refine**: Step back and review your diagram. Ensure that it accurately represents the interactions and relationships in your system. Refine as needed.
* **Step 7: Validate**: Share your use case diagram with stakeholders and gather feedback. Ensure that it aligns with their understanding of the system's functionality.