Class Diagram

Learn the concept of class diagrams and relationships between classes and their notations.

We'll cover the following

Class diagrams are used to depict the system's static perspective. They are used in the design process to show the shared roles and responsibilities of the entities that produce the behavior of the system.

Class diagrams are widely used in the modeling of object-oriented designs because this is the only UML diagram that can be directly transferred to object-oriented programming languages.

Why use class diagrams?

The following are some important purposes of the class diagram:

- Represents the system's static structure.
- Directly maps with object-oriented languages.
- Represents what the system's duties or responsibilities are.
- Uses in both forward and reverse engineering.

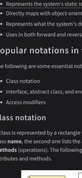
Popular notations in the class diagram

The following are some essential notations of the class diagram:

- Class notation
- Interface, abstract class, and enumeration
- Access modifiers

Class notation

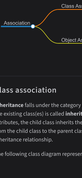
A class is represented by a rectangle with three sections. The first section holds the class **name**, the second one lists the **attributes**, and the third one shows the **methods** (operations). The following is the depiction of a `Movie` class with its attributes and methods.



Notation of a class in a class diagram

Interface, abstract class, and enumeration

We can declare a class as abstract using **abstract** keywords. The class name will be printed in *Italic*. We can use the interface, annotation, and enum keywords too. The illustration below shows how to depict these notations in a class diagram.



Class diagram notation for abstract classes, interfaces, enumerations, and annotations

Access modifiers

You may use character symbols to specify the visibility of the associated object when defining methods or attributes. The most widely used access modifiers are as follows:

Public: A public member can be seen anywhere in the system. It is represented by a `+` symbol.

Private: Members can only be accessible from within the class. It is inaccessible from outside the class. It is represented by a `-` symbol.

Protected: Members are only accessible within the class and derived classes. It is represented by the `#` symbol.

The following images show how to use the access modifiers in the class diagram:

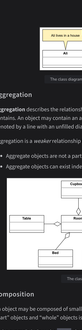
Access modifiers notations

Association

Association provides a mechanism to communicate one object with another object, or one object provides services to another object. Association represents the relationship between classes.

The association can be divided into two categories:

- Class association (Inheritance)
- Object association



Class association

Inheritance falls under the category of class association. Creating a new class from the existing class(es) is called **inheritance**. Apart from its own behaviors and attributes, the child class inherits the characteristics of its parent(s). A solid line leads from the child class to the parent class with a hollow arrowhead representing the inheritance relationship.

The following class diagram represents the inheritance relationship:

Class diagram to represent the multi-level inheritance

Note: We already discussed the different types of inheritance in the previous chapter.

Object association

Object association (relationship between objects) can be divided into the following categories:

1. Simple association
2. Composition
3. Aggregation

Simple association

The weakest connections between objects are made through **simple association**. It is achieved through reference, which one object can inherit from another. The following is an example of a simple association:

The class diagram representing a simple association

Aggregation

Aggregation describes the relationship between the container and the object it contains. An object may contain an aggregate of another object. Aggregation is denoted by a line with an unfilled diamond head towards the container.

Aggregation is a *weaker* relationship because:

- Aggregate objects are not a part of the container.
- Aggregate objects can exist independently.

The class diagram of aggregation

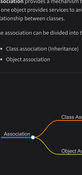
Composition

An object may be composed of smaller objects, and the relationship between the "part" objects and "whole" objects is known as **composition**.

In the example below, the `Chair` class can be composed of other objects of `Leg`, `Seat`, and `Arm` types. Composition is denoted by a line with a filled diamond head at the composer class pointing to the component class.

Composition is a *strong* relationship because:

- The composed object becomes a part of the composer.
- Composed objects can not exist independently.



The class diagram of composition

Some additional types of association

The following are some types of simple associations based on navigation:

- Single-direction navigation is called **one-way association** and is denoted by an arrow toward the server object.

The class diagram of one-way association

- If we navigate in both directions, the association is called a **two-way association** and is denoted by a line between two objects.

The class diagram of two-way association

- Binary, ternary, and *n*-ary associations are based on the number of objects.
  - Association in which two objects are involved is called a **binary association.** The binary association includes one-way or two-way navigation.
  - Association between the objects of exactly three classes is a **ternary association** and is denoted by a diamond with lines connected to associated objects.
  - Association between more than three classes is called **n-ary** association.

The class diagram of ternary association

Dependency

Dependency indicates that one class is dependent on another class(es) for its implementation. Another class may or may not depend on the first class. A dashed arrow denotes dependency.