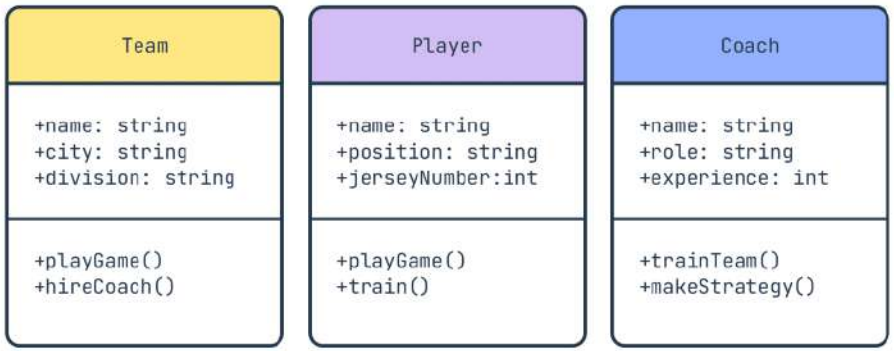
Class Diagram

*A UML Class Diagram is a visual representation that illustrates the **static** structure and relationships within a system. It primarily focuses on **classes, their attributes, methods,** and the **associations** between them.*

Class Diagram Notations:

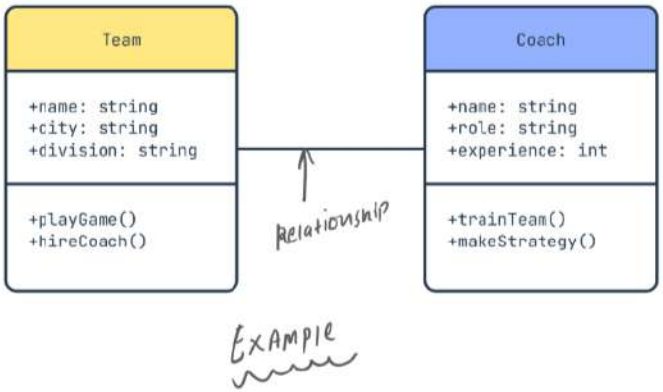
1. *Class*
2. *Interfaces*
3. *Abstract Class*
4. *Enumeration*
5. *Annotations*
6. *Access Modifiers*
7. *Association*
   - *Class Association*
   - *Object Association*



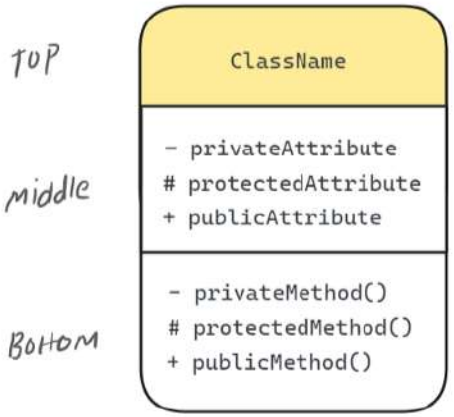
1. Class Notation

A class looks like a box with three parts. The top part has the class's name. The middle part has a list of features, and the bottom part has a list of actions the class can do.

Representation: *A rectangle divided into three compartments.*
Top Compartment: *Class name, usually in bold and centered.*
Middle Compartment: *Attributes (properties or variables).*
Bottom Compartment: *Methods (functions or procedures).*



Example



Top

Middle

Bottom

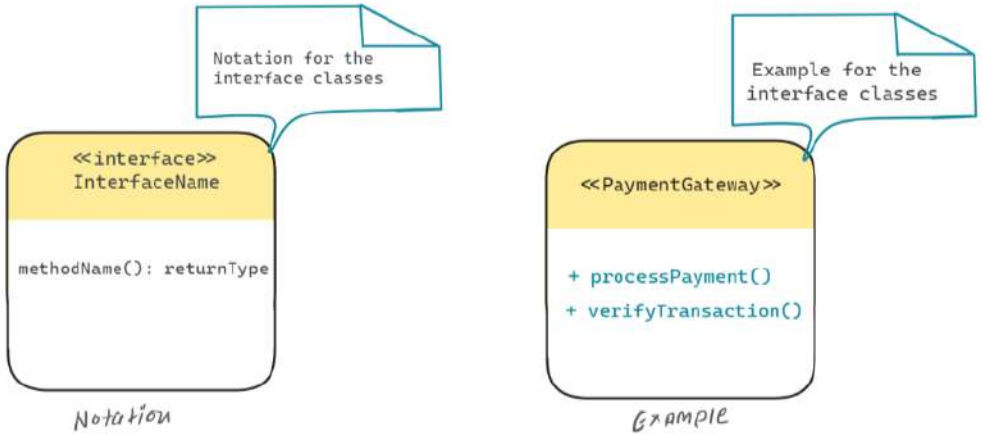
Notation

Class Notation in UML class Diagram

2. Interface Notation

An interface is a contract for a class that specifies what methods the class should implement, without providing the method implementation itself.

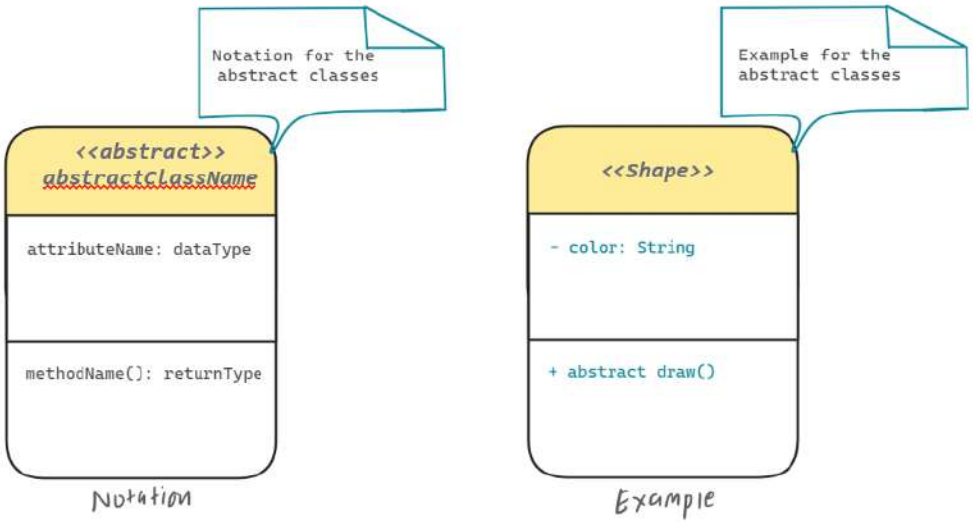
Representation: *A circle with the interface name inside.*
Meaning: *Represents a collection of abstract methods that a class implementing the interface must provide.*



3. Abstract Notation

Abstract classes are classes that cannot be instantiated on their own and are designed to be extended by subclasses that implement the abstract methods contained within.

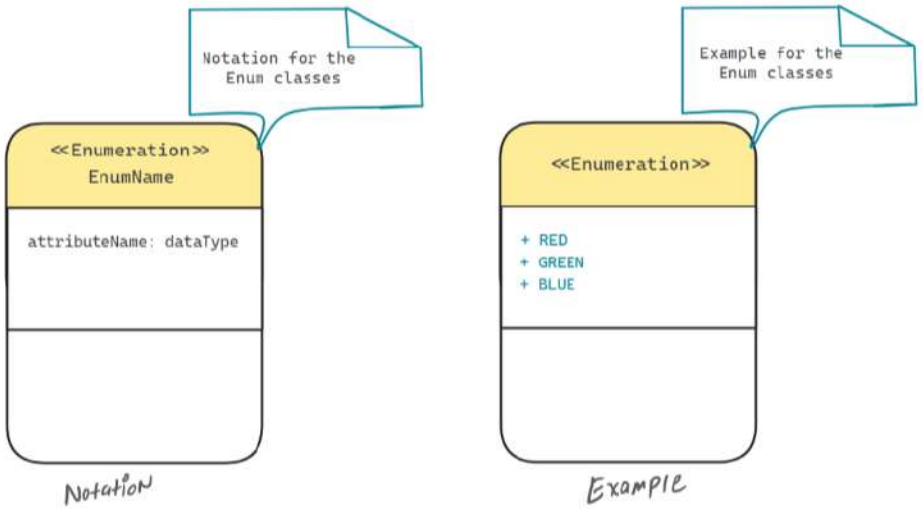
Representation: *Similar to a regular class, but the name is* ***italicized****.*
Meaning: *Represents a class that cannot be instantiated and may have abstract methods.*
It serves as a base class for other classes.



4. Enumeration Notation

*An Enum is a special data type that enables for a variable to be a set of predefined **constants**, improving type safety and readability.*

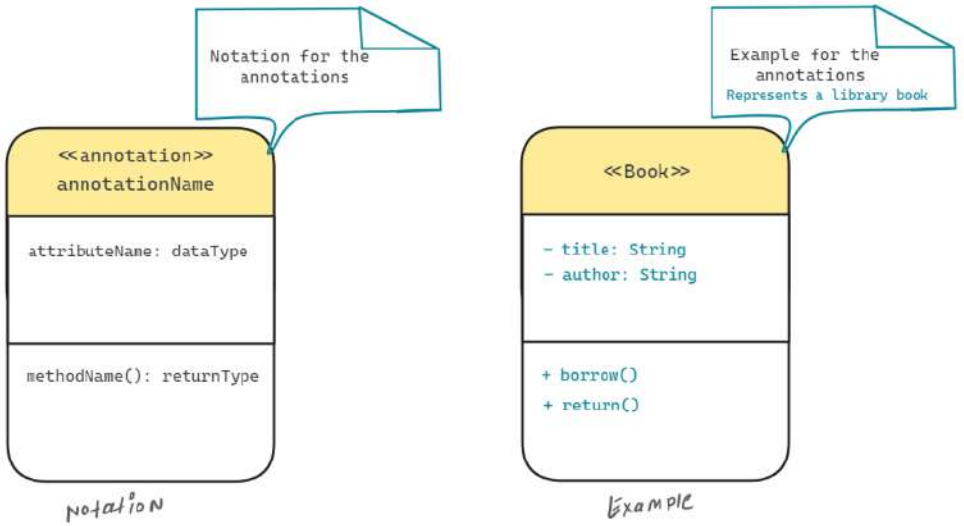
***Representation:** A rectangle with the enumeration name at the top, followed by its literals (enumeration values).*
***Meaning:** Represents a data type consisting of a set of named values.*



5. Annotations Notation

*Annotations are a form of **metadata** that provide information about the code but do not change the code itself; they are used to give additional information to the compiler or to be used through reflection at runtime.*

***Representation**: Note-like shapes attached to classes or relationships with additional information.*
***Meaning**: Provides additional comments or explanations to enhance understanding.*



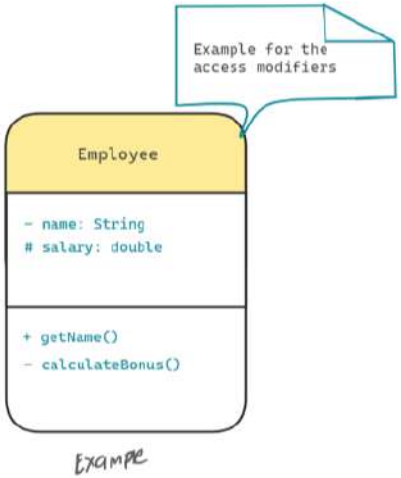
6. Access Modifiers Notation

When you create methods or attributes in programming, you can use special symbols to show who can see or use them. Here's what the symbols mean:

Representation: *Symbols like **+** for public, **-** for private, and **#** for protected, placed before attributes or methods in the class.*

- *The "+" symbol means everyone can see it. This is called "**public**".*
- *The "-" symbol means only the code inside the same class can see it. This is called "**private**".*
- *The "#" symbol means only the class it's in and classes that are based on it can see it. This is called "**protected**".*

Meaning: *Defines the visibility or accessibility of attributes and methods.*



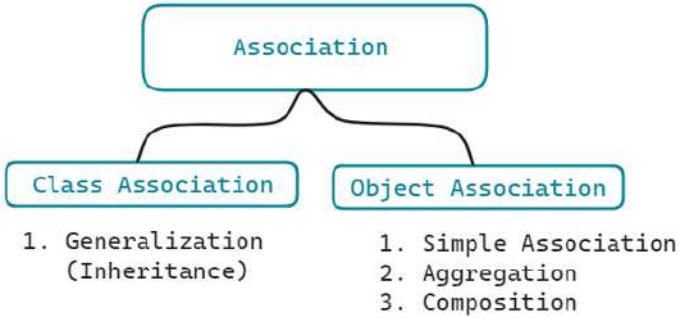
Example

7. Association Notation

*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:

1. ***Class association:*** *Generalization (Inheritance)*
2. ***Object association:*** *Simple Association, Aggregation & Composition*

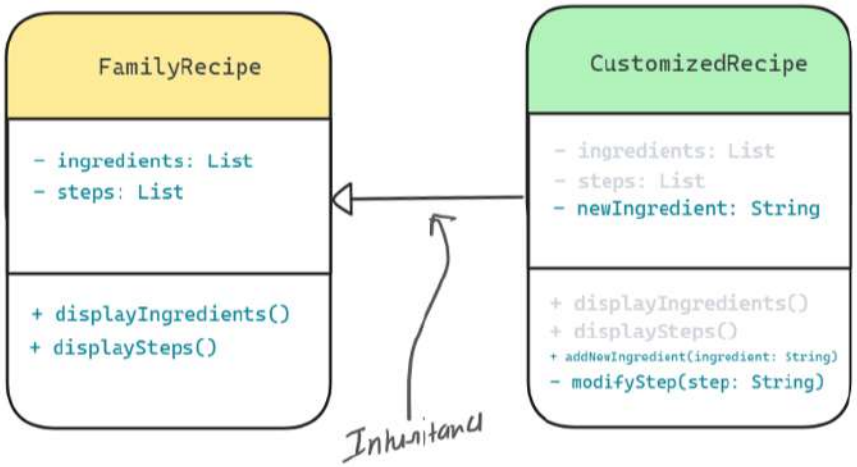


1. Class association: Generalization (Inheritance)

Inheritance is like making a new version of something that already exists.

*Imagine you have a recipe from your family **(the original class: FamilyRecipe)** and you decide to add your own twist to it **(creating a new class: CustomizedRecipe)**. This new recipe **(the child class)** still keeps all the original ingredients and steps **(characteristics of the parent class)**, but you might add something new or do something a bit differently. In pictures that show how classes are related, we draw a straight line with an empty arrow pointing from the new version **(child class)** back to the original **(parent class)** to show this connection.*

Example 1



Class Diagram Explanation:

FamilyRecipe Class:

1. **Attributes:**
   - **ingredients: List:** Represents the list of original ingredients in the family recipe.
   - **steps: List:** Represents the list of original steps or instructions in the family recipe.

2. **Methods:**
   - **displayIngredients():** Displays the list of ingredients in the family recipe.
   - **displaySteps():** Displays the list of steps or instructions in the family recipe.

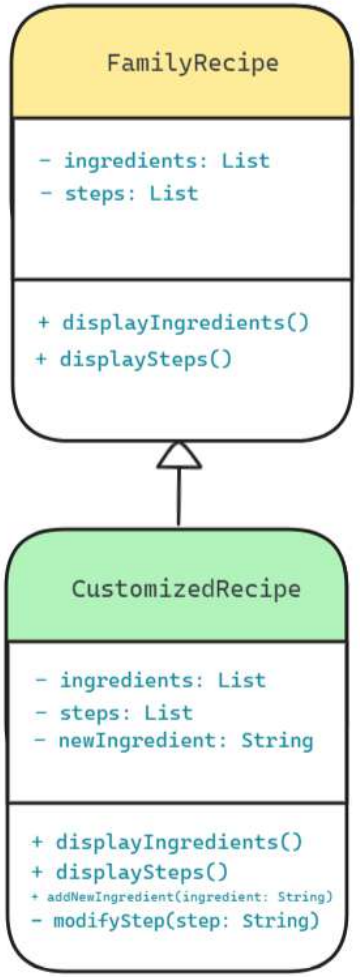
The **FamilyRecipe** class serves as the original or parent class, encapsulating the common features of a family recipe.

CustomizedRecipe Class:

1. **Attributes:**
   - **ingredients:** List: Inherits from the FamilyRecipe class, representing the original ingredients.
   - **steps: List:** Inherits from the FamilyRecipe class, representing the original steps.
   - **newIngredient:** String: Represents the additional ingredient in the customized recipe.

2. **Methods:**
   - **displayIngredients():** Overrides the method from the parent class to display customized ingredients.
   - **displaySteps():** Overrides the method from the parent class to display customized steps.
   - **modifyStep(step: String):** Modifies or adds a step to the customized recipe.
   - **addNewIngredient(ingredient: String):** Adds a new ingredient to the customized recipe.

The **CustomizedRecipe** class is the child class that inherits from **FamilyRecipe**. It introduces new functionality and represents a customized version of the family recipe.



Explanation:

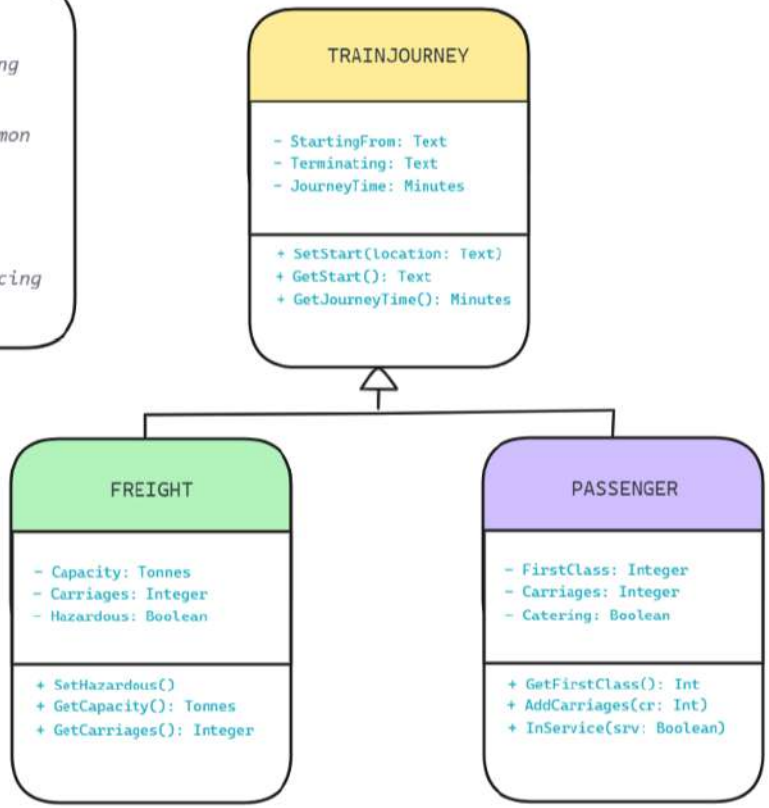
*1. The **TRAIN JOURNEY** class serves as the base class with common attributes and methods that are shared among different types of train journeys.*

*2. Both **FREIGHT** and **PASSENGER** classes inherit from the **TRAIN JOURNEY** class, indicating that they share common features but also have their specific attributes and methods.*

*3. The arrows in the diagram indicate the direction of inheritance, showing that **FREIGHT** and **PASSENGER** are specialized versions of the **TRAIN JOURNEY**.*

4. Inheritance allows the subclasses to reuse and extend the functionality of the base class while introducing additional features specific to each type of train journey.

Example-2



Class Diagram Explanation:

TRAINJOURNEY Class:

1. *Attributes:*
   - ***StartingFrom: Text:*** *Represents the starting location of the train journey.*
   - ***Terminating: Text:*** *Represents the terminating location of the train journey.*
   - ***JourneyTime: Minutes:*** *Represents the duration of the train journey in minutes.*

2. *Methods:*
   - ***SetStart(location: Text):*** *Sets the starting location of the train journey.*
   - ***GetStart(): Text:*** *Gets the starting location of the train journey.*
   - ***GetJourneyTime(): Minutes:*** *Gets the duration of the train journey.*

FREIGHT Class (Subclass of TRAIN JOURNEY):

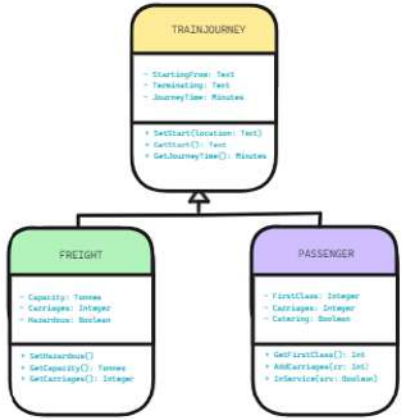
1. *Additional Attributes:*
   - ***Capacity: Tonnes:*** *Represents the capacity of the freight train in tonnes.*
   - ***Carriages: Integer:*** *Represents the number of carriages in the freight train.*
   - ***Hazardous: Boolean:*** *Indicates whether the freight train carries hazardous materials.*

2. *Additional Methods:*
   - ***SetHazardous():*** *Sets the hazardous flag for the freight train.*
   - ***GetCapacity():*** *Tonnes: Gets the capacity of the freight train.*
   - ***GetCarriages():*** *Integer: To retrieve the number of carriages in the freight train.*

PASSENGER Class (Subclass of TRAIN JOURNEY):

1. *Additional Attributes:*
   - ***FirstClass: Integer:*** *Represents the number of first-class carriages in the passenger train.*
   - ***Carriages: Integer:*** *Represents the total number of carriages in the passenger train.*
   - ***Catering: Boolean:*** *Indicates whether catering services are available in the passenger train.*

2. *Additional Methods:*
   - ***GetFirstClass(): Integer:*** *Gets the number of first-class carriages.*
   - ***AddCarriages(cr: Integer):*** *Adds carriages to the passenger train.*
   - ***InService(srv: Boolean):*** *Checks if the passenger train is in service.*



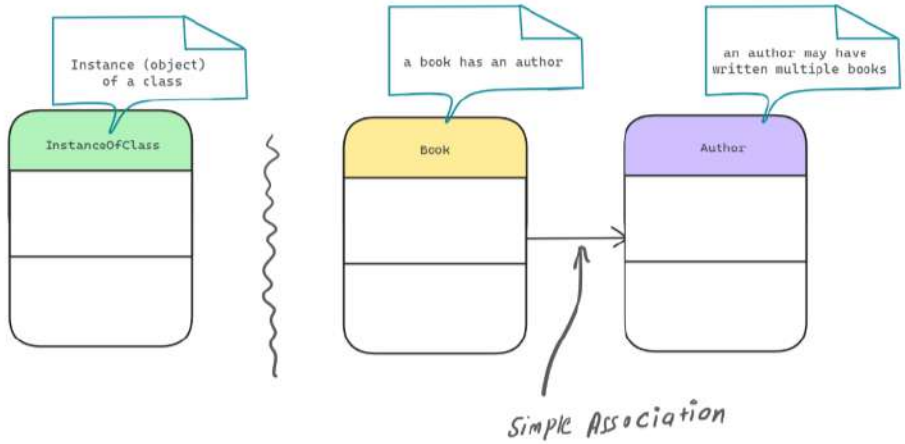
2. Object/Simple association (Has-A Relationship)

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:
a simple UML class diagram with a simple association between two classes, **"Book"** and **"Author"**.

Explanation:

- The "Book" class has attributes such as **"title"** and **"genre"**.
- The "Author" class has attributes such as **"name"** and **"email"**.
- There is a simple association between the **"Book"** and **"Author"** classes, represented by a line connecting them. This indicates that a book has an association with its author, but there is no specific ownership or dependency implied.
- The association is **one-directional**, meaning that a book has an author, but an author may have written multiple books (represented by the multiplicity notation).



Note: an object may reference data members or methods of its class, it doesn't directly contain the class itself.

More Examples on Simple/Object Association:

Object association, or simple association, is a concept where two or more objects are connected or related to each other in a straightforward manner. Here are five examples of simple associations:

1. Pen and Paper: *This is a classic example where the pen is used to write or draw on the paper. The association is based on the function of the pen and the utility of the paper.*

2. Key and Lock: *A key is associated with a lock because the key is designed to open or close the lock. This relationship is direct and is based on the functionality of the key matching the lock mechanism.*

3. Shoes and Laces: *Shoes and laces are associated because laces are used to secure shoes on the feet. The laces go through the eyelets of the shoes and can be tied to adjust the fit of the shoe.*

4. Charger and Smartphone: *The charger is associated with a smartphone because it is used to recharge the smartphone's battery. This is a straightforward relationship where the charger's output matches the smartphone's charging requirements.*

5. Teapot and Tea Cup: *A teapot is associated with a tea cup because the teapot is used to hold and pour tea into the cup. This association is based on the complementary functions of serving (teapot) and receiving (tea cup) the tea.*

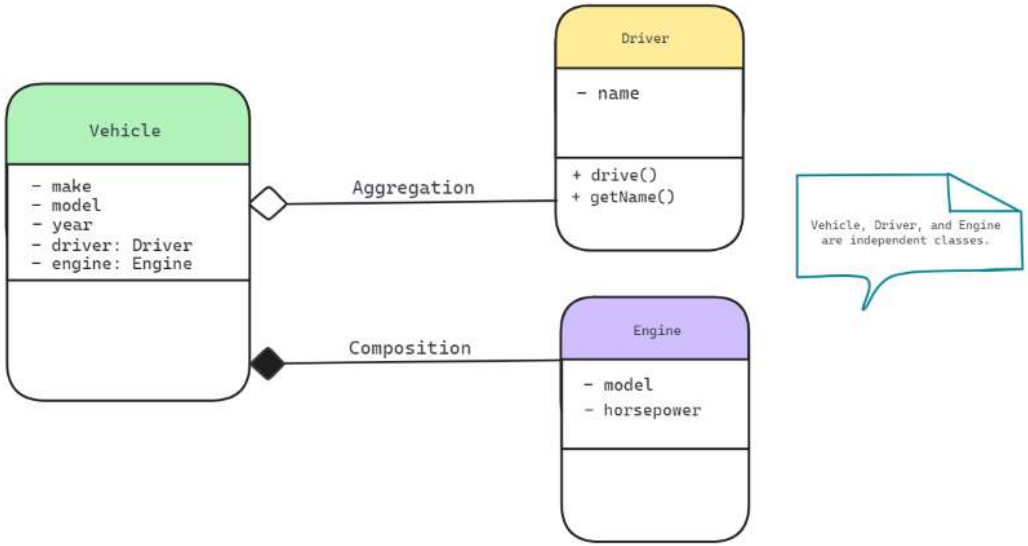
*2. **Object Aggregation** (Not A Strong "Has-A" Relationship)*

*Aggregation is like having a **box** where you can put things inside. The box is one object, and the things inside are other objects. We show this relationship by drawing a line with a **diamond shape** at one end that points to the box.*

This relationship is not very strong because:

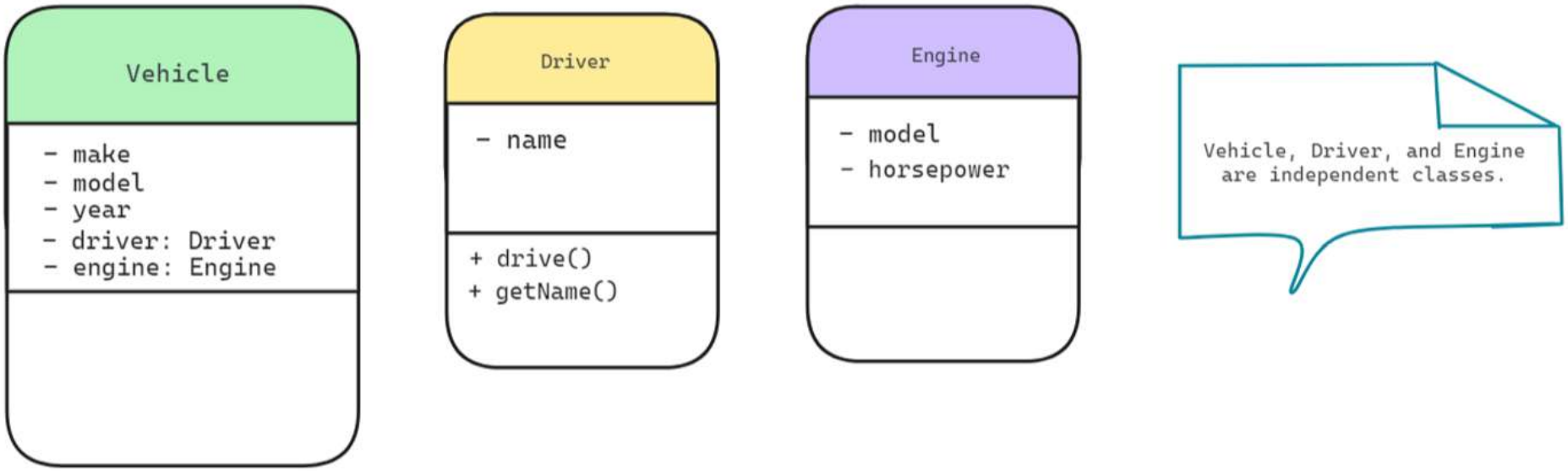
- *The things inside the box are not actually part of the box itself.*
- *The things inside can exist on their own, even if you take them out of the box.*

The following is an example of a aggregation:



Explanation:

- *The **"Vehicle"** class represents a whole object.*
- *The **"Driver"** and **"Engine"** classes represent part objects.*
- *The **"Vehicle"** class has attributes like **"make,"** **"model,"** and **"year,"** along with a reference to a **"Driver"** object and an **"Engine"** object.*
- *The **"Driver"** class has attributes like **"name"** and a method **"drive()"** to control the driver's actions.*
- *The **"Engine"** class has attributes like **"model"** and **"horsepower."***
- *There is an aggregation relationship between the **"Vehicle"** and **"Driver"** classes, and between the **"Vehicle"** and **"Engine"** classes, indicated by the **diamond shape** on the **"Vehicle"** end of the association lines.*
- *This aggregation relationship implies that a **vehicle** contains a **driver** and an **engine**, but they can exist **independently**.*



*They can exist and be used **independently** of each other.*

*However, they can also have a **relationship** where a **Vehicle** contains a **Driver** and an **Engine**.*

This relationship allows for modeling scenarios where a Vehicle is composed of a Driver and an Engine, but both the Driver and Engine can exist independently outside of the context of a specific Vehicle.

More Examples on Object Aggregation:

Aggregation in object association represents a "has-a" relationship where one entity (the whole) is made up of other entities (the parts), but the parts can also exist independently of the whole. Here are five examples of aggregation:

1. Library and Books: *A library contains books, but books can exist outside of the library as well. The library is an aggregation of books and possibly other resources like magazines, DVDs, and so on.*

2. Computer System and Hardware Components: *A computer system is an aggregation of various hardware components like the CPU, RAM, hard drive, motherboard, etc. Each of these components can exist independently of the computer system and can be used in different configurations or systems.*

3. Car and Wheels: *A car has wheels, but wheels are not exclusive to the car and can be used in other applications like motorcycles, carts, etc. The car is an aggregation of wheels along with other parts like the engine, seats, and steering wheel.*

4. University and Departments: *A university is comprised of various departments like Mathematics, English, Engineering, etc. Each department functions within the university structure but can also exist independently as specialized entities or in other university settings.*

5. Flower Bouquet and Flowers: *A flower bouquet is an aggregation of various flowers. Each flower contributes to the overall appearance of the bouquet, but each flower can also exist independently, not bound to the bouquet configuration.*

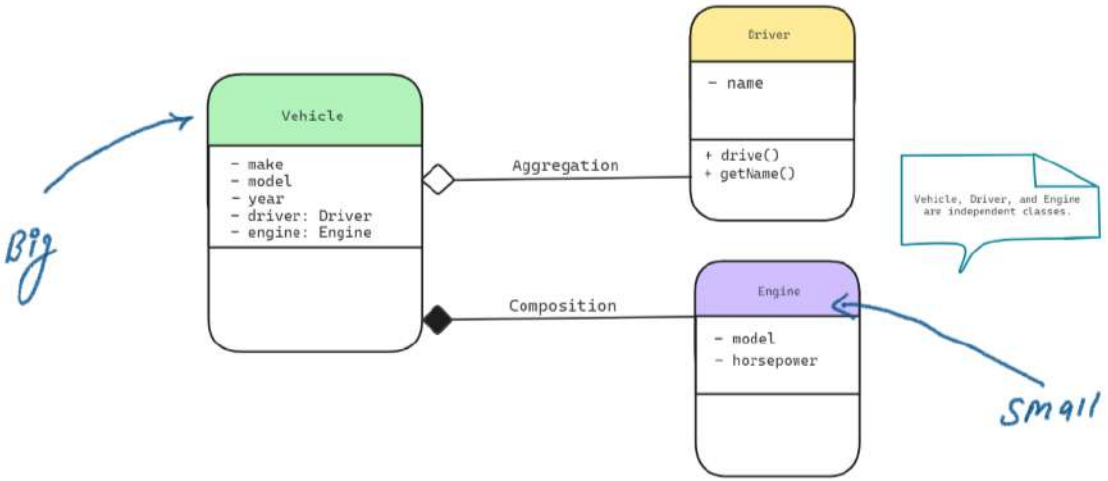
2. Object Composition (A Strong "Has-A" Relationship)

Composition is when a big thing is made up of smaller parts. Think of a chair as the big thing. It's made up of smaller parts like arms, a seat, and legs. We show this by drawing a line with a ***solid diamond shape*** *at one end, pointing from the big thing to its parts.*

This is a strong connection because:

- *The smaller parts become a part of the big thing.*
- *The smaller parts can't be on their own without the big thing.*

The following is an example of a composition:



More Examples on Object Composition:

Composition is a strong form of aggregation where the parts cannot exist independently of the whole; if the whole is destroyed, the parts are destroyed or no longer functional as well. Here are five examples of composition:

1. House and Rooms: *A house is composed of multiple rooms such as the kitchen, living room, bedrooms, and bathrooms. The rooms are defined by their existence within the structure of the house and do not have an independent existence outside of it.*

2. Human Body and Organs: *The human body is a composition of various organs like the heart, lungs, liver, and kidneys. These organs are integral to the body's functioning, and outside of the body, they lose their function and context.*

3. Airplane and Wings: *An airplane is composed of several parts, including wings. The wings are crucial for the airplane's ability to fly and are designed specifically for a particular airplane model. Without the wings, the airplane cannot function as intended, and the wings themselves have no purpose without the airplane.*

4. Book and Pages: *A book is composed of pages. The pages are bound together in a specific order and cannot function as a book if they are separated from the binding. The integrity of the book as an object is dependent on the pages being combined in a fixed arrangement.*