upGrad

#LifeKoKaroLift

Object-Oriented Analysis and Design

upGrad

Course: Object-Oriented Analysis, Design and Programming

Lecture On: Object-Oriented Analysis and Design

Instructor: Sayan Nayak

upGrad

Topics covered in the previous class...

1. Introduction to Version Control System and Git
2. How to start with Git and GitHub
3. Git commands to start with Git - git init and git config
4. Git commands for local changes - git status, git add, git commit, git log
5. Git commands for branching in Git - git branch, git checkout, git merge
6. Git commands to sync with Central Repo - git clone, git push, git pull, git fetch
7. The .gitignore file

Poll 1 (15 sec)

Which of the following Git commands help you move changes from the local repo to remote repo?

1. git move
2. git commit
3. git push
4. git fetch

Poll 1 (15 sec)

Which of the following Git commands help you move changes from the local repo to remote repo?

1. git move
2. git commit
3. **git push**
4. git fetch

upGrad

Homework Discussion

Home Work

upGrad

Please follow the steps given below to complete the Homework:

1. Create a new folder using ***mkdir*** command and name it 'upgrad'.
2. Change to the 'upgrad' folder using ***cd*** command.
3. Initialise this directory under version control using the ***git init*** command.
4. Create a new file 'courses.txt' using the ***vim*** command (vim <filename>).
5. Type 'DSA' in the file and save and exit (Esc + :wq).
6. Check status of the repo using ***git status*** command.
7. Add the changes to the staging area using the ***git add .*** command.
8. Set up the git using the ***git config --global user.email "you@example.com"*** and ***git config --global user.name "Your Name"*** commands.

Home Work

upGrad

9. Commit the changes using the ***git commit*** command with message 'added DSA course'.
10. Create a new branch 'dev' using ***git branch*** command and check-it-out using ***git checkout*** command (Or you can do both using ***git checkout -b*** command).
11. Update the 'courses.txt' file using the ***vim*** command and add 'Java' in the new line. So now 'courses.txt' file would contain three lines, first line would have 'DSA', second line would have 'Java' and third line would be empty.
12. Check the status, stage the changes and commit the new changes with message 'added Java course'.
13. So 'master' branch should have just one commit corresponding to DSA and 'dev' branch should have two commits corresponding to DSA and Java.

upGrad

Today's Agenda

- **Design Inventory Management System**
  - Introduction to OOAD and UML diagrams
  - Project overview
  - Start with UML class diagram
  - Understand class, its attributes and methods
  - Draw a class diagram using diagrams.net
  - Different characteristics of class attributes and methods
  - Different relationships between classes

upGrad

Introduction to Object-Oriented Analysis and Design (OOAD)

Question: What is the first step to make complex objects?

Suppose you want to make a new mobile phone. How would you start? What would be the first step? If you have all the raw materials required, can you really build a mobile phone?

Question: What is the first step to make complex objects?

Ans: Before you can start making a new mobile phone, you have to perform the following two tasks:

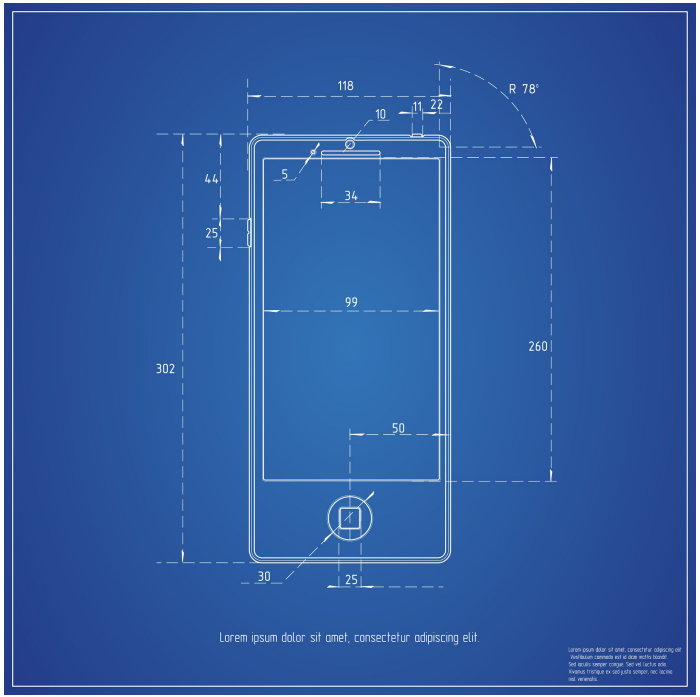
1. ***Analysis***: You have to first determine the requirements for the mobile phone and then list down all the features that it would have.
2. ***Design***: Once you have listed down all the features, you have to start designing its blueprint.

Introduction to OOAD

upGrad

- To make any complex system, such as a mobile phone, we first need to answer two questions in the following order:
  a. ***WHAT-*** What is the requirement for this product? This is called the ***Analysis*** phase.
  b. ***HOW-*** How will it be designed? This is called the ***Design*** phase.

- In the analysis phase, we prepare a requirement doc. And in the design phase, we design the blueprint for the product.



Internal design of a phone

- The blueprint is not only helpful in designing the product, but it also helps in the following:
  - *Keeping all the Stakeholders on the same page*, as they know how other people are imagining the end product
  - *Evolving the Product*, as it gives both the high-level picture and information on the minute details, thus making it possible to take calculated decisions
  - *Knowledge Transfer*, as you can use it to explain how the product works, without the need to disassemble it

Poll 2 (15 sec)

What is the first step to build any complex system?

1. Analysis
2. Design
3. Implementation
4. Deployment

Poll 2 (15 sec)

What is the first step to build any complex system?

1. **Analysis**
2. Design
3. Implementation
4. Deployment

- The same is true while developing big enterprise-level software, such as Amazon or Facebook.
  - ***Analysis:*** Usually not performed by the developers, and they simply get a requirement doc, which lists down all the requirements and the features of the application.
  - ***Design:*** We developers have to design the ***blueprint*** of the application based on the requirement doc.
- ***But how do we design the blueprint of software, which is not tangible?*** To answer that question, we have to first understand what are the different paradigms for software development?

Programming Paradigms

upGrad

- ***Programming paradigm:*** A programming paradigm is a way to structure a software. It tells you how to divide software in smaller parts and write code for them.
- Two of the most widely used programming paradigms are:
  - ***Procedural:*** The application is visualised as a sequence of steps and divided into multiple functions.
  - ***Object-Oriented:*** The application is visualised as a group of objects and divided into multiple classes.

Procedural Programming vs OOP

upGrad

Procedural Programming	Object-Oriented Programming
Application is divided into functions.	Application is divided into classes.
Application is designed in a sequence of steps.	Application is designed in such a way that different objects interact with each other.
Adding new feature is difficult as adding a step in between may break all the steps afterwards.	Adding new features is relatively easy.
No proper way for data hiding.	You can hide data in classes.
Does not imitate real-world objects.	Imitates real-world objects.
Example: C, COBOL, FORTRAN, etc.	**Example:** Java, C++, C#, etc.

Poll 3 (15 sec)

OOP is preferred to Procedural Programming because _____.

1. OOP divides applications into classes.
2. It is easy to add new features in OOP applications.
3. OOP hides data in classes.
4. There are more programming languages in OOP than in Procedural Programming.

Poll 3 (15 sec)

OOP is preferred to Procedural Programming because _____.

1. OOP divides applications into classes.
2. **It is easy to add new features in OOP applications.**
3. OOP hides data in classes.
4. There are more programming languages in OOP than in Procedural Programming.

Object-Oriented Programming

upGrad

- In OOP, objects are used to represent real-world entities which have some properties and perform some actions.
- These properties and actions are represented by objects using attributes and methods, respectively.
- For example, if we are creating a Person object in our application, then that person object would have attributes such as *name*, *age* and *height,* and methods such as *walk()*, *eat()* and *speak()*.

- So, let's come back to the earlier question: ***How do we design the blueprint of software, which is not tangible?***
- To build any application:
  - We first have to choose the programming paradigm, which would be the Object-Oriented paradigm for all modern applications.
  - We can then perform the analysis and design based on that.

Introduction to OOAD

upGrad

- For those applications where the Object-Oriented paradigm is employed, the analysis phase is referred to as ***Object-Oriented Analysis***, or ***OOA***, and the design phase is referred to as ***Object-Oriented Design***, or ***OOD***.
- Together, these two are called ***Object-Oriented Analysis and Design***, or ***OOAD***.
- In OOA, we analyse what are the different entities that would be present in the system, what would their properties be, what are the actions that they would perform and how would they interact with each other.
- In OOD, we design the system by drawing the classes (the blueprint for the objects – more on that later) and the relationships between them.

Introduction to UML

upGrad

- So, to design a system, we have to draw the classes and the relationships between them.
- When OOAD started, people were following different conventions to draw classes.
- Because of which the class diagram for the same system looked completely different.
- But now, all of this has been standardised and unified under a common language, ***Unified Modeling Language***, or ***UML***.
- UML is not a programming language but a standard way to design a system and draw its blueprint.

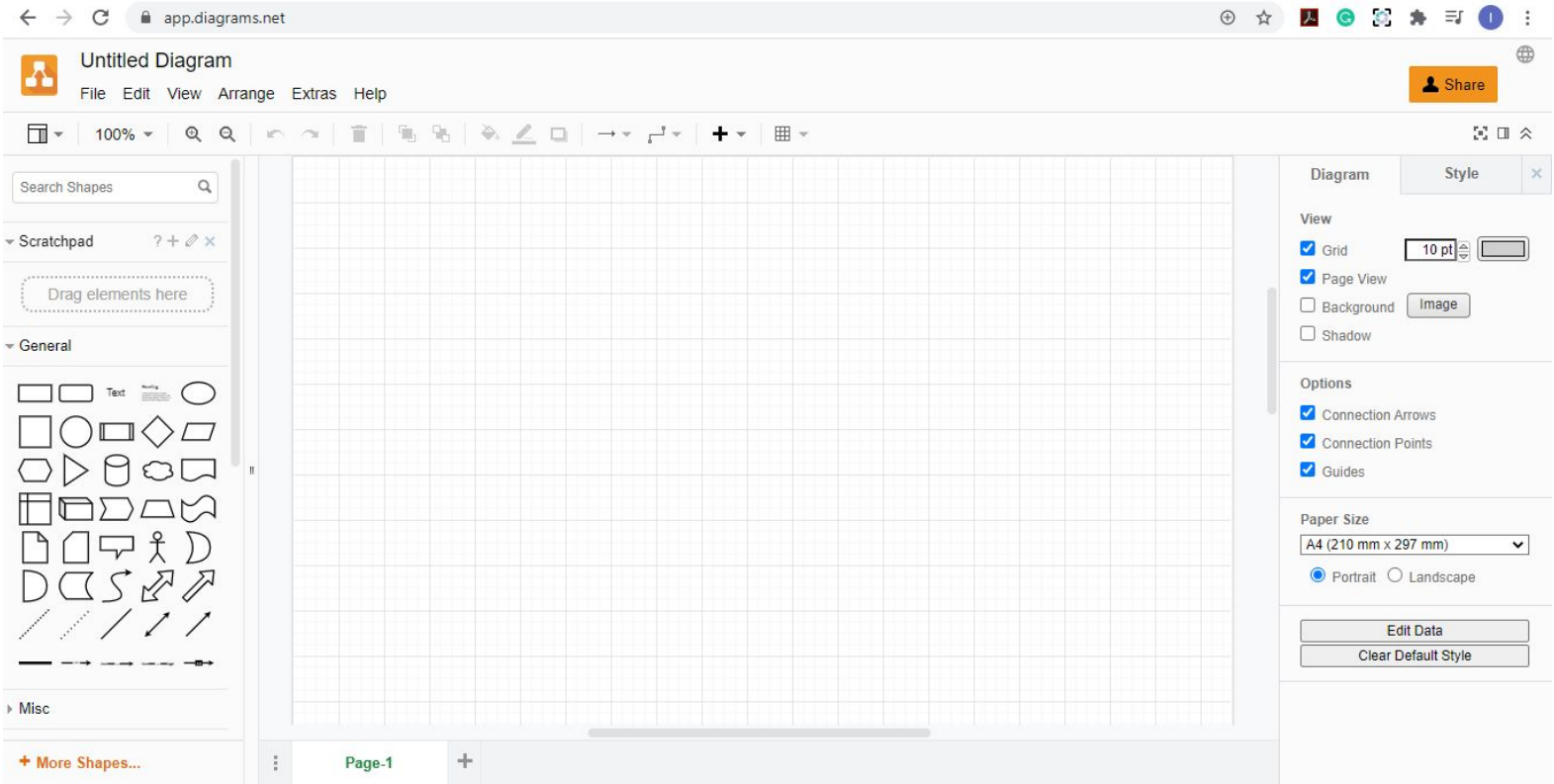
diagrams.net

upGrad

- There are several tools to draw UML diagrams for a system.
- But in this course, you will be using diagrams.net to design the Inventory Management System, or IMS.

diagrams.net

upGrad



Poll 4 (15 sec)

What are the benefits of designing a blueprint? (More than one option maybe correct.)

1. Cost-cutting
2. Easy evolution of products
3. Knowledge transfer
4. Fast deployment

Poll 4 (15 sec)

What are the benefits of designing a blueprint? (More than one option maybe correct.)

1. Cost-cutting
2. **Easy evolution of products**
3. **Knowledge transfer**
4. Fast deployment

upGrad

Project Overview

Project Overview

upGrad

- Let’s start building the Inventory Management System.
- As you know, the first step would be analysis, which is mostly performed by the business team or by project managers.
- They prepare the requirement doc for the application and hand it over to the developers.
- The developer has to start with the design phase and start drawing the class diagram for the application.

IMS Requirement Doc

upGrad

IMS Requirement Doc

Product object

upGrad

- From the requirement doc, we can find out different objects and the corresponding attributes and methods for them.
- Let’s write down all the attributes and methods corresponding to the Product object.
  - **Object:** Product
  - Attributes (using lowerCamelCase)
    - id, name, category, salesPrice, cost, quantity, active
  - Methods (using lowerCamelCase)
    - getProfitOrLoss(), activate(), deactivate(), isBelowThreshold()

Class

upGrad

- In our IMS, there can be many objects to represent to represent real-world entities, such as different mobiles, different laptops and different accessory products.
- But we want to ensure that all of the products follow the same structure. All of them have the required properties (values may differ from product to product) and the required methods.
- This structure is enforced using classes. ***A class is a blueprint for creating different objects of the same type. All the objects of that type provide a particular set of attributes and methods***.

Class

upGrad

- There can be hundreds or thousands of product objects in the IMS application, but there will be only one Product class.
- To create an object, we first have to define a class. This class will then act as a blueprint for creating objects of that type and for enforcing a particular structure for those objects.
- Once we have created an object using a class, we cannot add or remove the attributes or the methods of that object.

- There exist certain ***dynamically*** typed languages, such as JavaScript, which allow you to create objects without first defining a class.
- Because of this, there is no restriction on JavaScript objects to follow a rigid structure.
- You can add or remove attributes and methods whenever you want.
- However, in Java, the rules are simple: If you want to create an object, first provide a class for that. Even if you want to create just one object, you have to first declare a class.

Poll 5 (15 sec)

What is a class? (More than one option maybe correct.)

1. It is a special type of object.
2. It is a blueprint for creating objects.
3. It provides a basic structure for the objects.
4. It is another name for object methods.

Poll 5 (15 sec)

What is a class? (More than one option maybe correct.)

1. It is a special type of object.
2. **It is a blueprint for creating objects.**
3. **It provides a basic structure for the objects.**
4. It is another name for object methods.

Poll 6 (15 sec)

Which of the following statements is true with respect to objects? (More than one option maybe correct.)

1. They represent real-world entities in applications.
2. The object attributes represent the properties of real-world entities.
3. The object methods represent the actions of real-world entities.
4. Objects provide a structure to the classes.

Poll 6 (15 sec)

Which of the following statements is true with respect to objects? (More than one option maybe correct.)

1. **They represent real-world entities in applications.**
2. **The object attributes represent the properties of real-world entities.**
3. **The object methods represent the actions of real-world entities.**
4. Objects provide a structure to the classes.

Hands-On Experience

upGrad

TO-DO:

- Using the requirement doc, find out all the objects, their attributes and methods, and how the different objects interact with each other.

Starting With Class Diagram

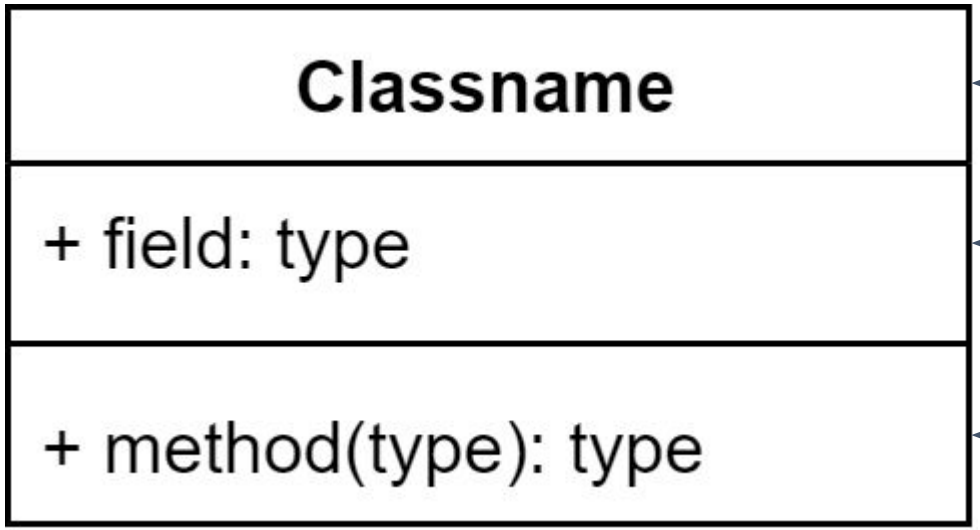
Class Diagram

upGrad

- A class diagram is used to show:
  - The classes in the system
  - The relationships between the different classes
- Once we know about the classes and relationships between them, we can then determine the different types of objects that will be present in the system (using the presence of the classes) and how these objects will interact with each other (using the relationships between the classes).
- Let's start drawing the Product class along with its attributes and methods in diagrams.net.

Class Diagram – Class Syntax

upGrad



Class name (in UpperCamelCase)

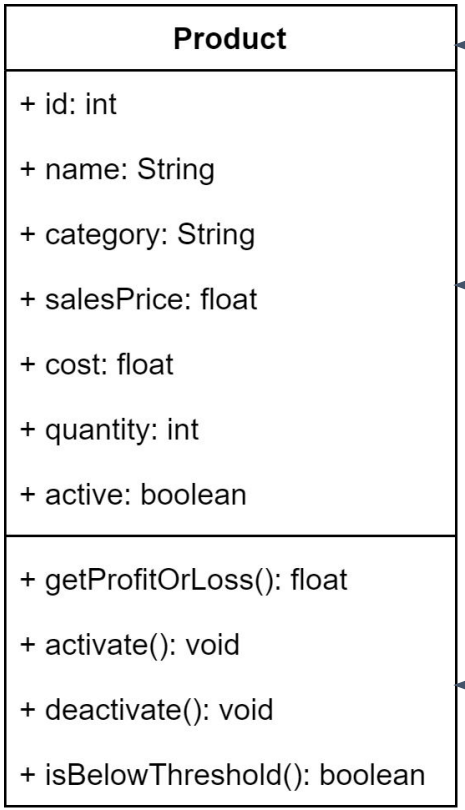
Attributes with their types (in lowerCamelCase)

Methods with their parameters (along with type) and return types (in lowerCamelCase)

For now, you can ignore the plus symbol. We will learn about it later in the session.

Class Diagram – Product Class

upGrad



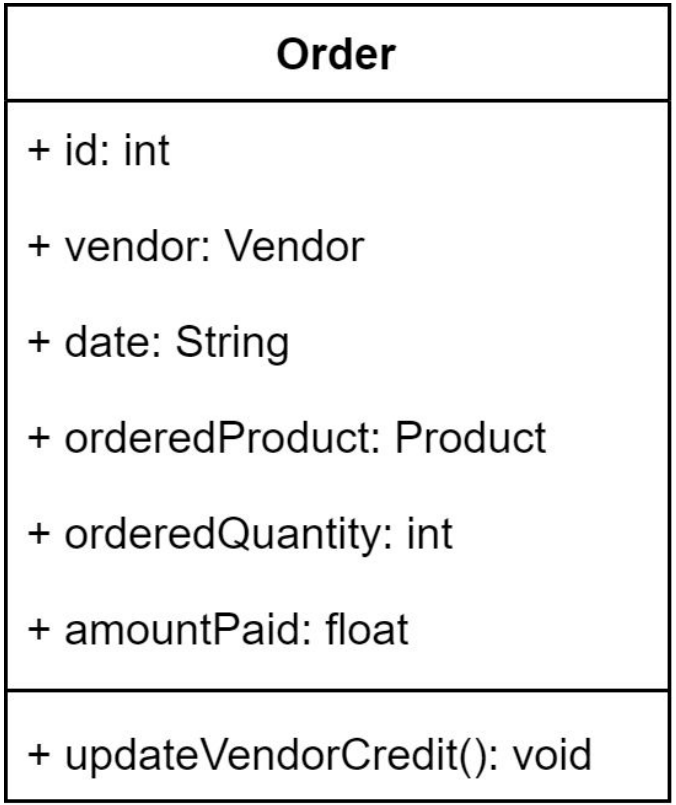
Class name (in UpperCamelCase)

Attributes with their types (in lowerCamelCase)

Methods with their parameters (along with type) and return types (in lowerCamelCase)

Class Diagram - Order Class

upGrad



Class name (in UpperCamelCase)

Attributes with their types (in lowerCamelCase)

Methods with their parameters (along with type) and return types (in lowerCamelCase)

Poll 7 (15 sec)

Which of the following is the correct way to represent a method in class diagram?

1. int add(num1: int, num2: int)
2. add(num1: int, num2: int): int
3. int add(int num1, int num2)
4. add(int num1, int num2): int

Poll 7 (15 sec)

Which of the following is the correct way to represent a method in class diagram?

1. int add(num1: int, num2: int)
2. **add(num1: int, num2: int): int**
3. int add(int num1, int num2)
4. add(int num1, int num2): int

Hands-On Experience

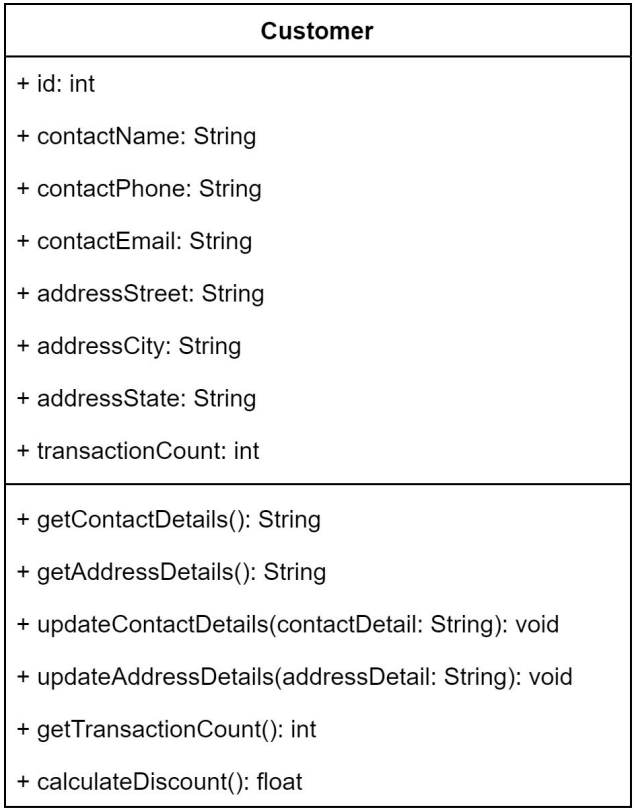
upGrad

TO-DO:

- Using the requirement doc, draw the class for the Customer objects.

Hands-On Experience

upGrad



Visibility and Multiplicity

Visibility

upGrad

- Visibility is basically of three types. We will learn about two of them here, and the third one while learning about class relationship.
  - ***public (+)***: The class members are accessible from within or outside the class.
  - ***private (-)***: The class members are accessible only from within the class.
- Visibility is also called access specifier since it specifies how the attributes and methods of a class will be accessed by different classes.

Visibility

upGrad

- Public methods provide an interface for an object to interact with that object.
- Private methods are used for the internal working of the objects, or the methods that are quite volatile (changing frequently by making changes to the source code), are marked private.
- So, even if we make any changes to the private methods or remove them, we can be sure that the code will not break.

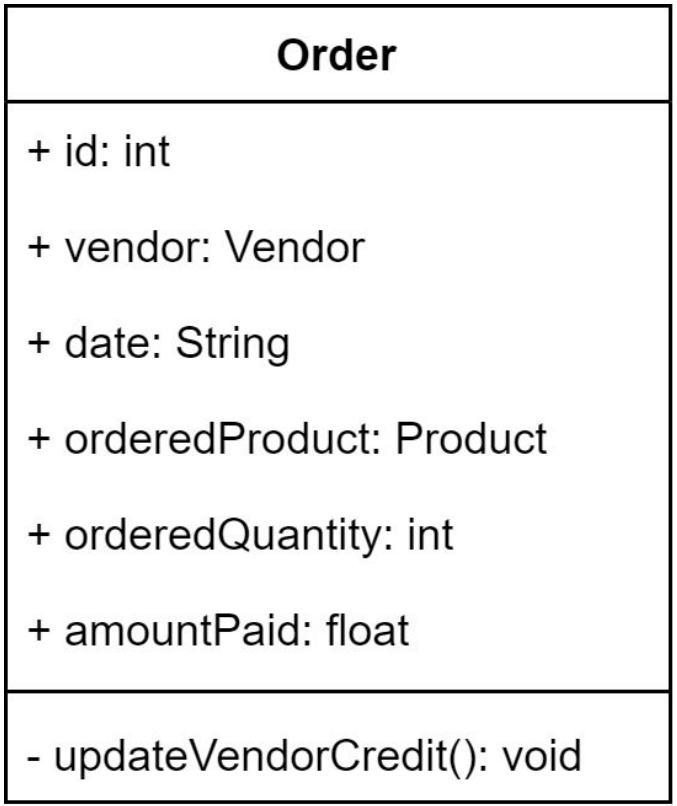
Visibility

upGrad

- As a best practice under data encapsulation, all the attributes of the class are generally made private and we provide public getter and setter methods to interact with the attributes.
- For a private method, let's take the example of the updateVendorCredit() method inside the Order class.
- Whenever we create a new order, this method will be called automatically by the Order object to update the vendor credit.
- No other object needs to call this method and, hence, no other object needs access to this method. This method is purely for the internal working of the Order class and, thus, it can be made private.

Visibility

upGrad



Multiplicity

upGrad

- Let’s draw the class for the Vendor objects which has almost the same attributes and methods as the Customer class.
- The attributes and methods that are different from the Customer class can also be represented very easily, except the list of products that are sold by the vendor.

Multiplicity

upGrad

- To represent the collection of objects or the array of objects, we have to use the concept of multiplicity as shown below:
  +/- variableName: ClassName [min...max]
- Here, min and max represent, respectively, the minimum and maximum number of elements that can be present in the collection.
- For example, if the vendor can sell a minimum of 0 products and a maximum of 100 products, then it can be represented in the class diagram as shown below:
  + products: Product [0...100]

Multiplicity

upGrad

Poll 8 (15 sec)

Which of the following is the correct way to declare a variable?

1. = customers: Customer[0..100]
2. + customers: Customer[100..0]
3. - customers: Customer[0..100]
4. - customers: Customer[100]

Poll 8 (15 sec)

Which of the following is the correct way to declare a variable?

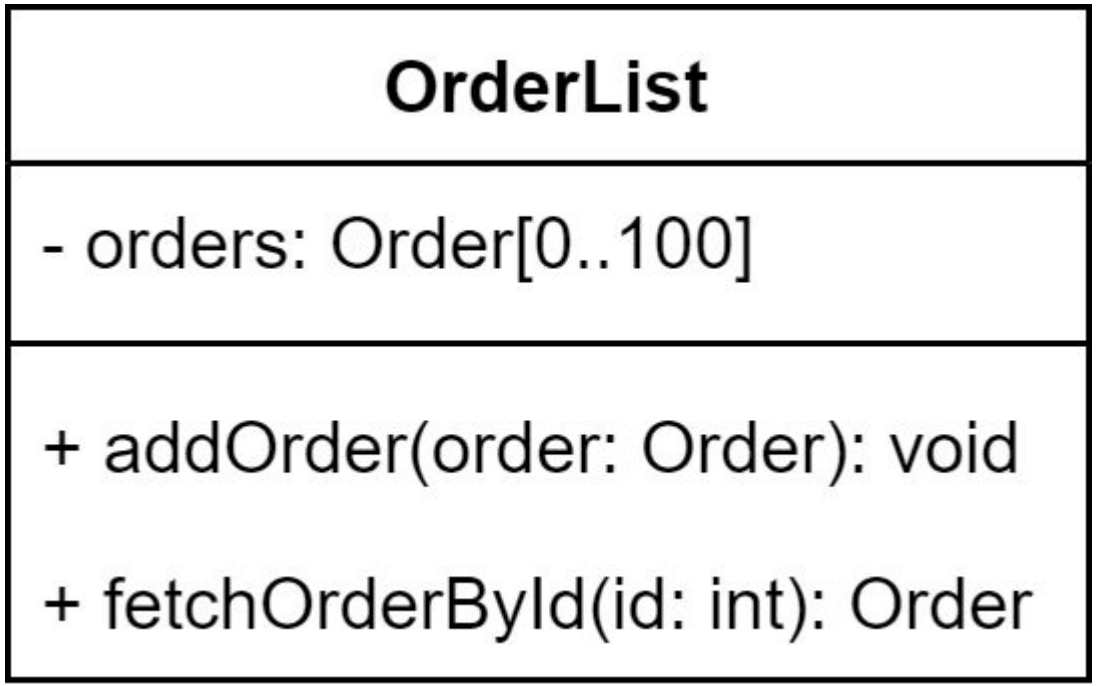
1. = customers: Customer[0..100]
2. + customers: Customer[100..0]
3. **- customers: Customer[0..100]**
4. - customers: Customer[100]

Hands-On Experience

upGrad

TO-DO:

- Draw an OrderList class. This class should have one attribute, orders, which is a collection of Order.
- This collection can have at least 0 elements and at max 100 elements.
- This collection should not be visible outside the class.
- Provide the following two methods:
    - To add a new Order to the collection and return void.
    - To get an Order from the collection using order id.


OrderList
- orders: Order[0..100]
+ addOrder(order: Order): void
+ fetchOrderById(id: int): Order

upGrad

Class Relationship

Class Relationship

upGrad

- The objects in the system, as well as real-life entities, never work in isolation. They always interact with each other to perform some work.
- For example, the Vendor object will be interacting with the Product object to contain a list of products, which the vendor sells.
- The Order object will also interact with the Product object to specify for which product that order was placed.
- So, how do we ensure that the objects interact with each other in the desired manner?

Class Relationship

upGrad

- We can enforce object interaction using class relationship.
- If we establish a relationship between the two classes in some particular manner, then all the objects of those two classes will interact with each other in that particular manner only.
- For example, if we declare a collection of Product objects in the Vendor class, then the Vendor object will have a collection of product objects.

Class Relationship

upGrad

- The relationship between classes can be of the following four types:
  - Association
  - Generalisation
  - Aggregation
  - Composition

Association Relationship

upGrad

- Association relationship is also called link or collaboration.
- It is the general form of the other three relationships.
- During the development of a system, we first identify how the different objects will interact with each other and then draw a link or an association relationship between the classes for those objects.
- Later during the development process, you make decisions about the exact relationship between the classes and then refine the association relationship into one of the other three relationships.

Association Relationship

upGrad

- Based on the direction, association relationship can be of the following two types:
  - Unidirectional relationship
  - Bidirectional relationship

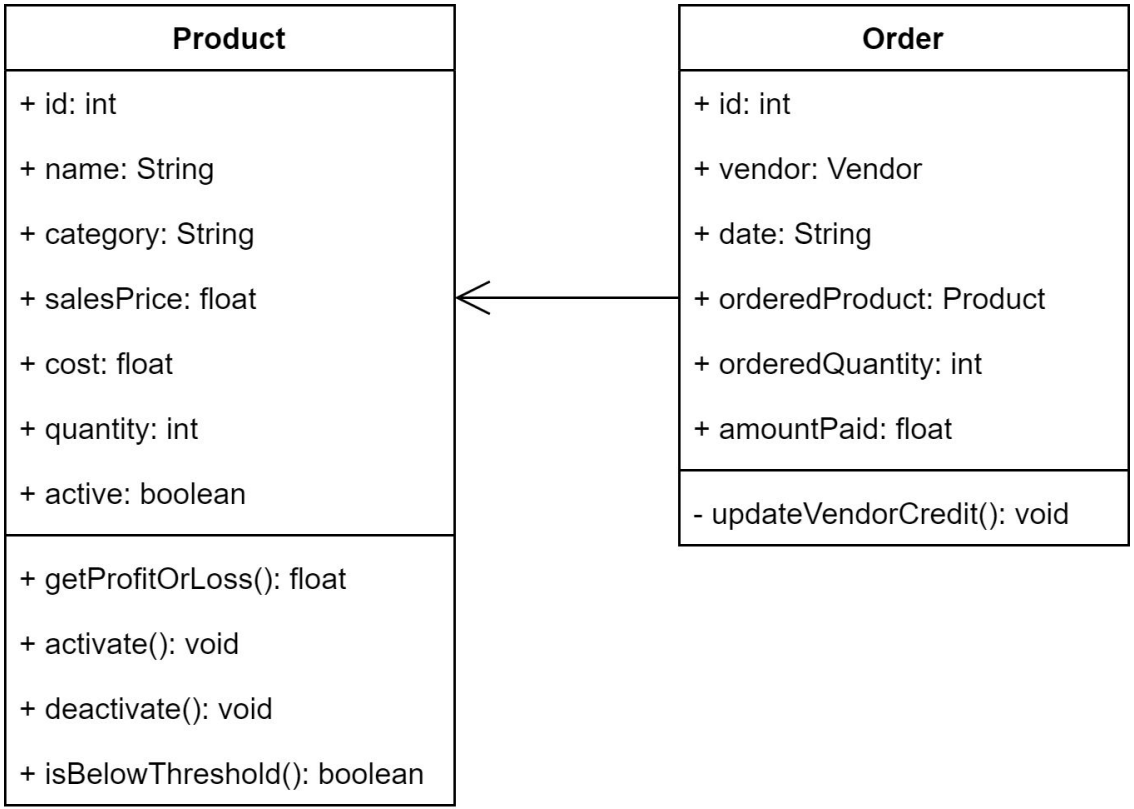
Unidirectional Association Relationship

upGrad

- This relationship is established between those classes where only one of the interacting objects can access the other interacting object, and not the opposite.
- For example, consider the relationship between the Order and Product objects where only the Order object needs to access the attributes and methods of the Product object.
- In unidirectional relationship, only one of the related classes will have an attribute of another class. For example, only the Order class has an attribute of Product class.
- Unidirectional relationship is drawn using an arrow, which starts from the source class (which has the attribute of the other class) and points towards the target class (which does not have the attribute).

Unidirectional Association Relationship

upGrad



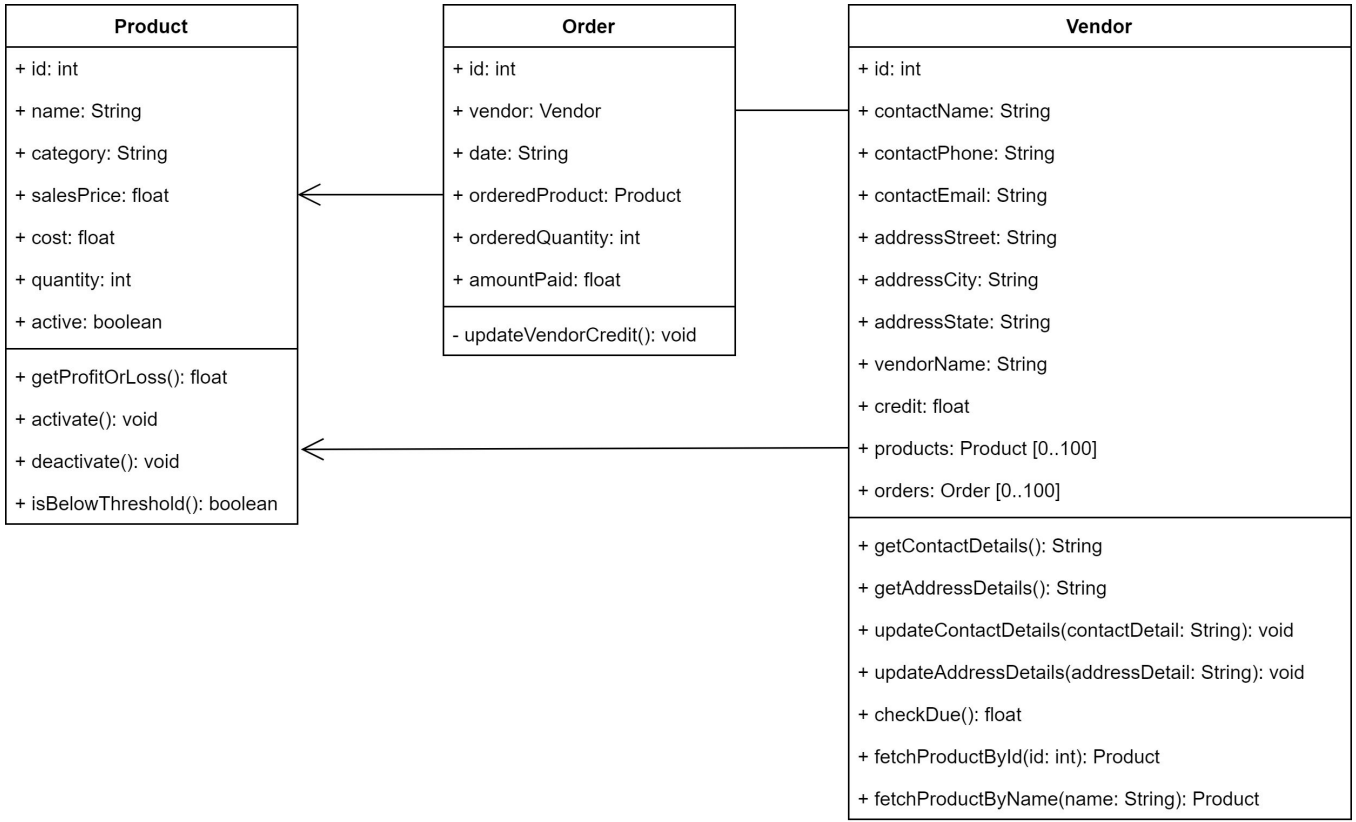
Bidirectional Association Relationship

upGrad

- This relationship is established between those classes where both the interacting objects can access one another.
- For example, the relationship between the Order class and the Vendor class.
- Bidirectional relationship is drawn using a solid line between the two classes.

Bidirectional Association Relationship

upGrad



Association Relationship

upGrad

- Based on multiplicity, association relationships are of the following three types:
  - ***One-to-one***: When both the sides interact with just one object of each other. *Example:* Product and Order
  - ***One-to-many***: When one side interacts with many objects of another side, but the other side interacts with only one object of that side. *Example:* Vendor and Order
  - ***Many-to-many***: When both the sides interact with many objects of each other. *Example:* Product and Vendor.
- You should always specify multiplicity for an Association relationship. You should write ‘1’ for one side and ‘*’ for many sides.

Poll 9 (15 sec)

A battalion can consist of many soldiers, but one soldier can belong to only one battalion. Also, both these objects contain attribute to refer to each other. Which type of relationship is this?

1. One-to-One Unidirectional
2. One-to-Many Bidirectional
3. One-to-Many Unidirectional
4. Many-to-Many Bidirectional

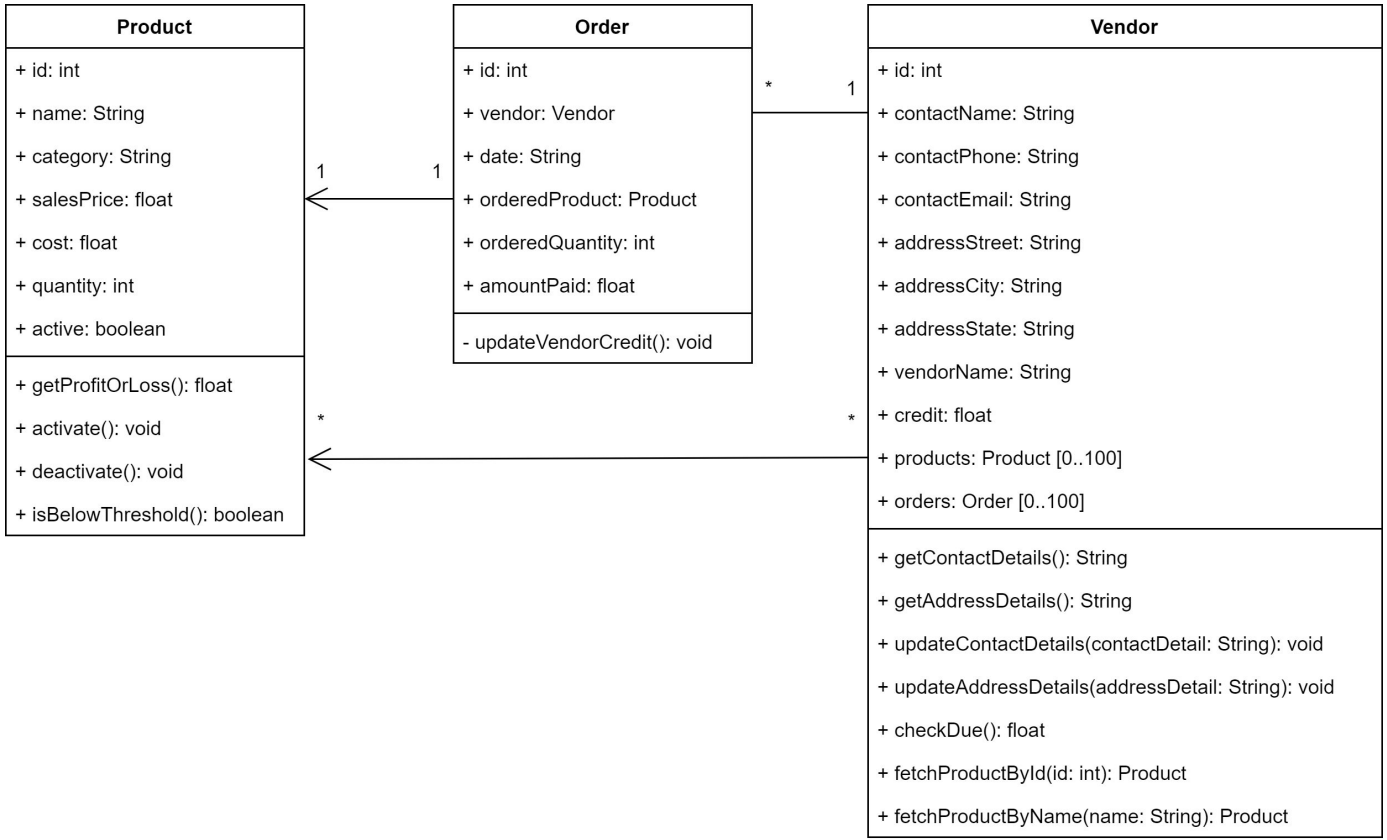
Poll 9 (15 sec)

A battalion can consist of many soldiers, but one soldier can belong to only one battalion. Also, both these objects contain attribute to refer to each other. Which type of relationship is this?

1. One-to-One Unidirectional
2. **One-to-Many Bidirectional**
3. One-to-Many Unidirectional
4. Many-to-Many Bidirectional

Association Relationship

upGrad



Generalisation Relationship

upGrad

- If you observe the Vendor and Customer classes, then you will notice that they are quite similar.
- Most of their attributes and methods are the same. If you implement classes in their current form, then there will be a lot of code duplication.
- In real life also, both Customer and Vendor will be your business partners and will help you grow your business.
- So, how do we remove this code duplication and improve code reusability?

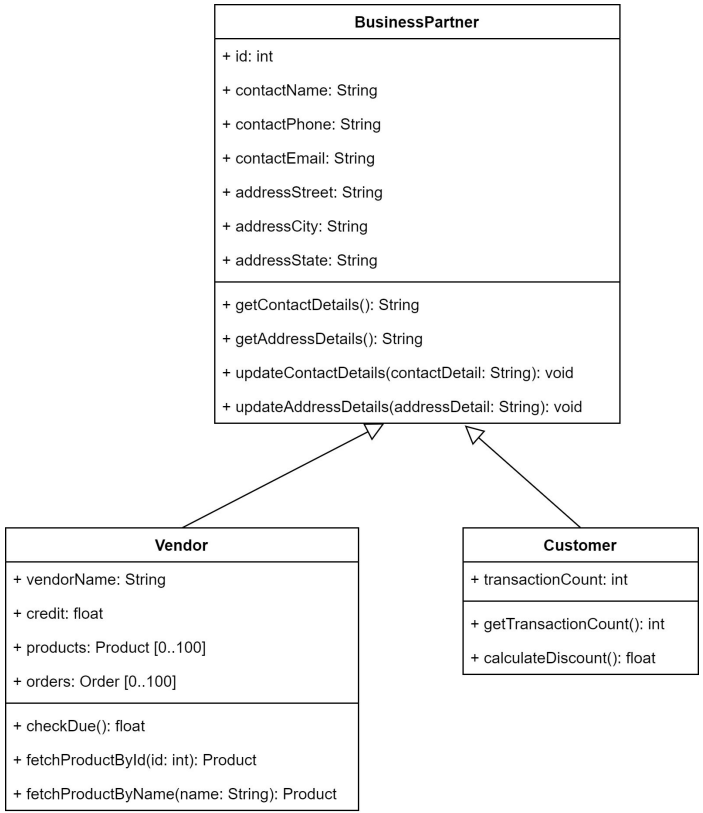
Generalisation Relationship

upGrad

- We can create a general class, called BusinessPartner, and put the common attributes and methods in it.
- Now you can remove the common attributes from the Customer and Vendor classes and establish generalisation relationship between the Customer and BusinessPartner classes, as well as the Vendor and BusinessPartner classes.
- Now the Vendor and Customer classes will inherit all the ***public*** attributes from the BusinessPartner class.
- This reduces code duplication and improves code reusability.
- Remember that only a child class can access the class members of a parent class, not the other way round.

Generalisation Relationship

upGrad



Generalisation Relationship – Protected Visibility

upGrad

- In generalisation relationship, the child classes (from which the generalisation arrow originates) inherit only the public class members of the parent class (to which the generalisation arrow points).
- There is also a third kind of visibility, called ***protected*** visibility. It is represented with the '#' symbol.
- Protected class members are visible only in the same class (same as private) and in the child classes (broader than private) but not anywhere else (narrower than public).

Generalisation Relationship – is a

upGrad

- The generalisation relationship is also called the ‘is a’ relationship.
- For example, we can say that Vendor *is a* BusinessPartner and Customer *is a* BusinessPartner.
- A parent class cannot access the member variables of a child class, nor do child classes of other child classes at the same level.
- But a parent class or other child classes can access the class members of other child classes using other relationships, such as association.

Aggregation and Composition Relationships

upGrad

- So far, we know that:
  - All objects interact with each other.
  - In the unidirectional relationship, one object will have an attribute of another object.
  - In the bidirectional relationship, both the objects will have each other's attributes.
- Most of the time, these interactions are of one of the following two types:
  - When one object ***owns*** another object.
  - When one object ***uses*** another object.

Aggregation and Composition Relationships

upGrad

- Suppose we want to establish a relationship between object A and object B.
- So, we will establish an association relationship between class A and class B.
- Then we have to determine whether:
  - A owns B, which means B has no purpose for existence without A. This is called ***composition*** relationship.
  - A uses B, which means B can exist even without the existence of A. This is called ***aggregation*** relationship.

Aggregation and Composition Relationships

upGrad

- Composition relationship is represented using a ‘filled diamond arrow’, whereas aggregation relationship is represented using an ‘empty diamond arrow’.
- The arrow always point to the owner side of the relationship and marked with the multiplicity.
- If we cannot determine whether the relationship is aggregation or composition, then we leave it as association relationship.

Aggregation and Composition Relationships

upGrad

- For example, suppose we provide separate classes to store Address and Contact information, which is also a good practice.
- Now, Address and BusinessPartner (either Customer or Vendor) will be related to each other and the BusinessPartner object is the owner of this relationship (since business partner contains the attribute of the Address class).
- If the owner of this relationship (the BusinessPartner object) is deleted, then we should also delete the associated Address object because now the existence of the Address object does not make sense.
- Therefore, there is a ***composition*** relationship between BusinessPartner and the Address class.

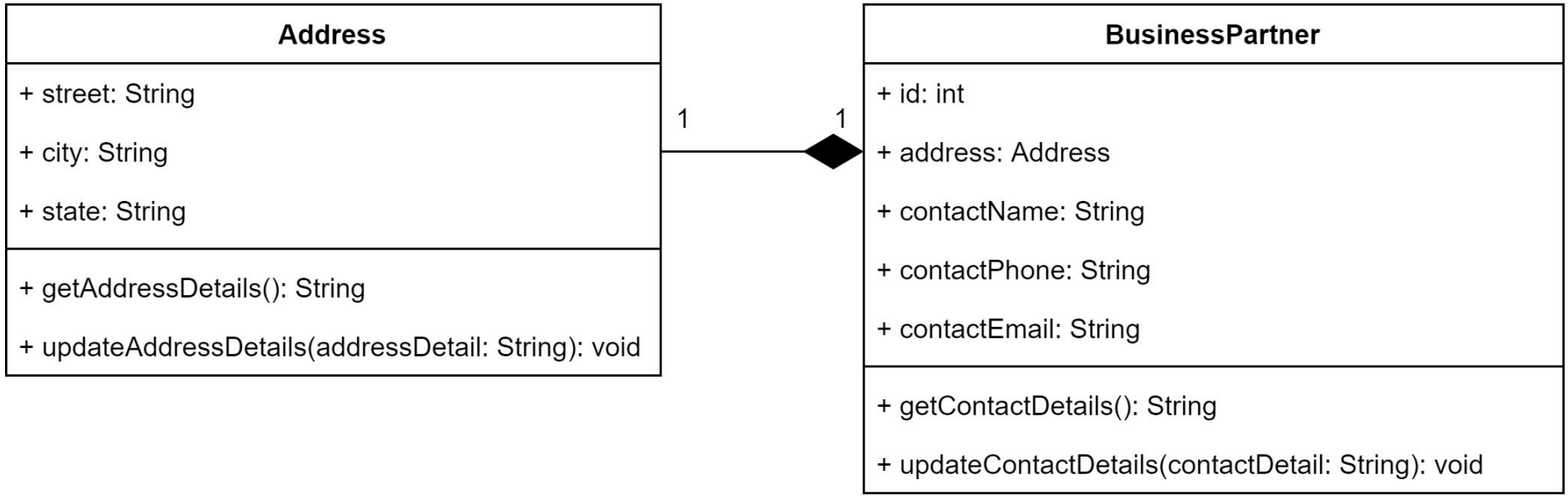
Aggregation and Composition Relationships

upGrad

- But the relationship between Product and Order (the owner of the relationship) is of aggregation type.
- When we delete the Order object, we do not need to delete the associated Product object, since the Product object can exist without the existence of the Order object.

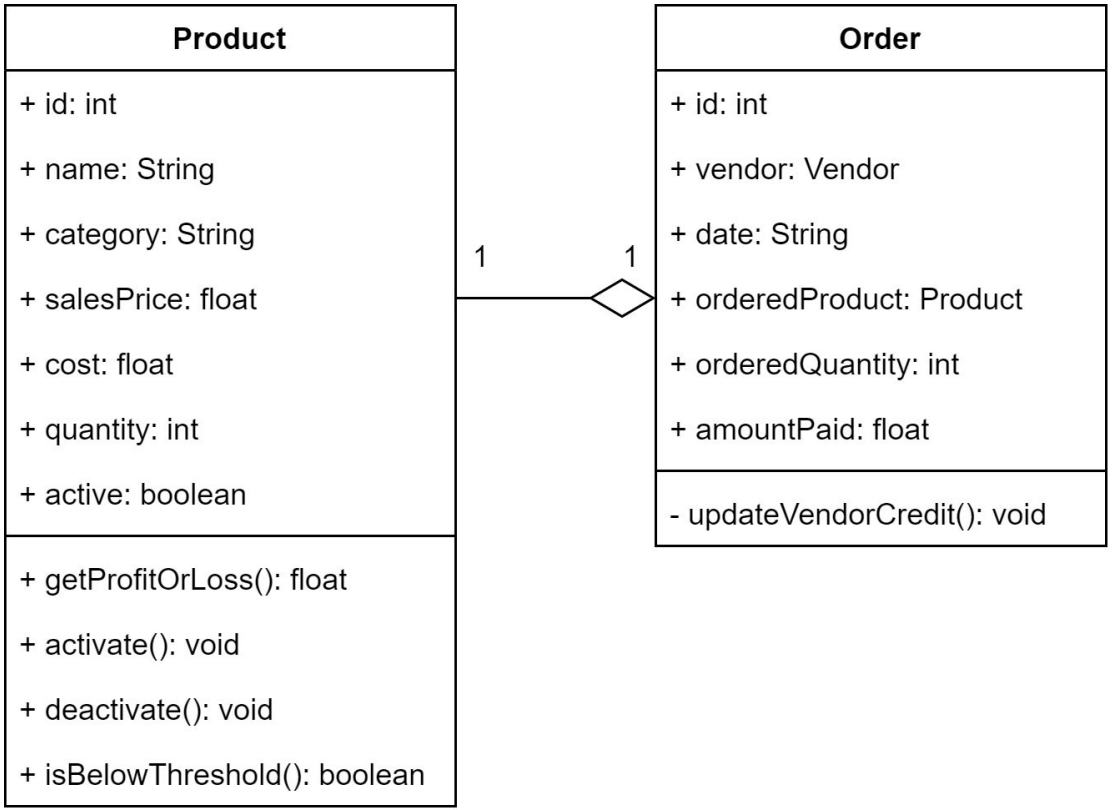
Composition Relationship

upGrad



Aggregation Relationship

upGrad



Poll 10 (15 sec)

Which of the following statements about aggregation and composition relationships is true? (More than one option can be correct.)

1. Aggregation is a ‘uses’ relationship.
2. Composition is an ‘owns’ relationship.
3. In Aggregation, a child object cannot exist without the owner object.
4. In Composition, a child object can exist without the owner object.

Poll 10 (15 sec)

Which of the following statements about aggregation and composition relationships is true? (More than one option can be correct.)

1. **Aggregation is a ‘uses’ relationship.**
2. **Composition is an ‘owns’ relationship.**
3. In Aggregation, a child object cannot exist without the owner object.
4. In Composition, a child object can exist without the owner object.

All the diagrams used in today's session can be found at the link provided below:

https://github.com/ishwar-soni/fsd-ooadp-ims/tree/session2-demo

upGrad

Important Concepts and Questions

Important Questions

upGrad

1. What do you understand by OOA and OOD?
2. Explain Generalisation relationship with a real-world example?
3. Differentiate between Aggregation and Composition using real-world examples? Provide at least two examples for each.
4. Explain the different visibilities of class members along with their UML symbols.
5. What are the uses of private methods?
6. How to identify whether the association relationship between the two classes is unidirectional or bidirectional?
7. Explain one-to-one, one-to-many and many-to-many relationships between classes.

upGrad

Doubt Clearance Window

upGrad

Today, we learnt about the following:

1. What is OOAD and why it is needed
2. Procedural Programming vs Object-Oriented Programming
3. What are Object-Oriented Analysis and Object-Oriented Design
4. Introduction to Unified Modeling Language
5. How to read a requirement doc
6. How to identify objects, their attributes and methods
7. What is a class and how to draw one using diagrams.net
8. What are visibility and multiplicity for class members
9. Different class relationships: Association, Generalisation, Aggregation and Composition

Homework

upGrad

In today's session, we have designed the Inventory Management System, which we will be developing in future sessions. But the diagram is still not complete.

Please use the requirement doc and complete the diagram. You do not have to start from scratch. You can build over the existing diagram.

upGrad

Tasks to Complete After Today's Session

MCQs
Homework
Project Checkpoint 1

upGrad

#RahoAmbitious

Thank You!