When you work on a project, you might need a proper visual representation of a system structure. If you can divide your project into a set of objects and use the Object-Oriented Paradigm (OOP), you can use a **Class Diagram**. It would help you to describe the components of your application and better understand the connections between them.

If you are working on an application that uses OOP, a Class Diagram will define your project structure and will make it apparent to the developers.

**What is a class diagram?**

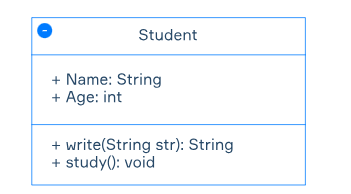
In the Unified Model Language (UML), **Class Diagram** is a visual representation of an object-oriented structure. To describe said structure the diagram uses the following elements:

* Classes;
* Class attributes;
* Class methods;
* Class relationships.

These elements form a set of classes and relationships between them. Such structure provides a better understanding of connections within a system and allows us to easily demonstrate the contents of an object.

**Description of a class**

A class is a representation of an object that describes its methods and attributes. Simply put, a class can be seen as a blueprint for an object. When the system creates a specimen of an object, it will be created according to a template that was described by the class. Here's an example of a class, called "*Student"*:



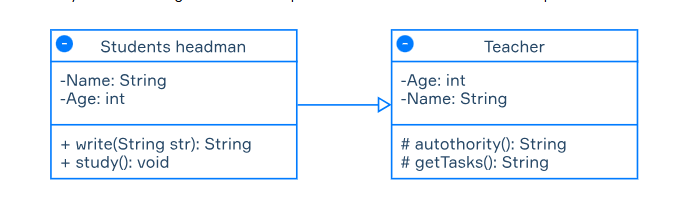
*Student*has 2 attributes: *Name*defined by String type, and *Age* defined by int type. The class also has 2 methods: *write()* which returns String type and *study()* which returns nothing. Methods also have brackets that can be filled in with some parameters. In this example, we have the parameter *String str*that is required to activate method write*()*.

Each element of a class has a visibility option, defined before the name of the element. In this example, all the elements have a "+" before their names. That means that all the visibility options of these elements are set to **public**. Let's elaborate on these visibility options:

* + (**Public**) – element can be accessed by any class in the system;
* - (**Private**) – element can be accessed only by a class that owns it;
* # (**Protected**) – element can be accessed by classes that have a generalization (or inheritance) relationship with its class;
* ~ (**Package**) – element can be accessed by classes that are located in the same package.

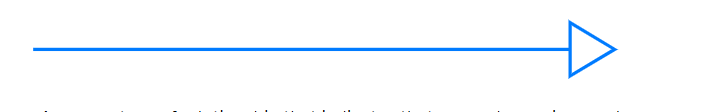
**Description of class relationships**

*Class relationships* are a concept that defines connections between classes. For example, we have 2 classes in our applications: *Student headman* and *Teacher*. We define their connection by establishing a relationship of some sort. Here's an example:



In this example, we use the type of relationship called**generalization (or inheritance)**. Let's elaborate on the relationship types:

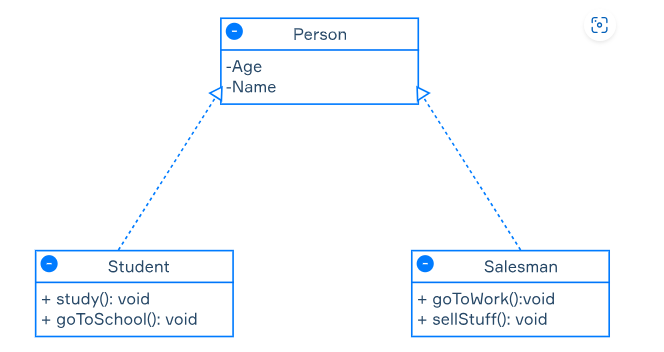
* **Generalization (inheritance) –**a type of relationship where one class could be described as a child class which assumes and could use methods of a parent class.In our case *Student headman*is the child class of *Teacher.* This relationship could be visualized by this arrow below:



* **Dependency –**a type of relationship that indicates that some change in one class can affect another class. Here's an example of *Student* depending on *School*:



* **Realization** – the relationship between the blueprint class and the object containing its respective implementation level details. For example, we have a class *Person*which describes the basic attributes of a person. It's a blueprint that can be made into an object that represents a specific person like a *Student*, or a*Salesman*. Our *Student* class can study and go to school, and our *Salesman* class can go to work and sell stuff, but they are people, so they will have *Age* and *Name* attributes. Here's how you could visualize it:

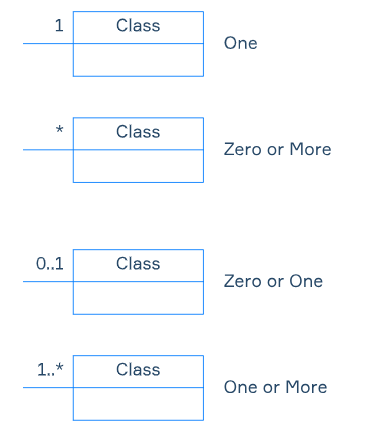


**Description of association relationships**

**Association –**a type of relationship that indicates that instances of one class are connected to instances of another. For example, *Teacher* teaches *Student.* This relationship can be visualized by a straight line:



Association relationships could also include a **cardinality**attribute. Simply put, this attribute defines the number of instances of a class that could exist in this relationship. Here's how it can be visualized for different situations:

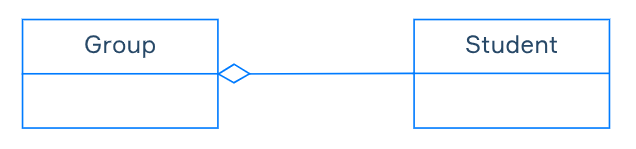


A basic example of this concept is this: class *Teacher*and class*Student* are connected by an association relationship. It means that *Teacher*teaches *Student*. One student can have one or multiple teachers, and the same can be said about teachers. So the diagram that describes this relationship would look like this:



Association relationships also have two special types:

* **Aggregation**– a special type of association relation that describes one class as a part of the other. Classes in this relation have a separate lifespan. If we come back to our *Student*example, we can describe class *Student* as part of *a Group*. This relation can be represented in this way:



* **Composition**– aspecial type of aggregation, where classes share lifespan. If the main class (*School*) stops its functions, the class that is a part of it (*Group*) will stop functioning too. This relationship can be visualized like this:



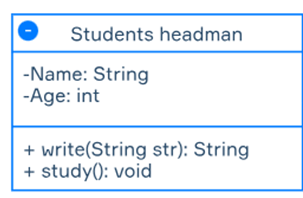
**Conclusion**

Class Diagram could be seen as too complicated at first. But when you start working on your first project, making your first framework, and connecting all the components into one structure, you may realize that this way of presenting project structure is really helpful. It makes all the application components look more apparent, and it also helps to set boundaries of their contents.

Imagine you need to make a class attribute invisible for all other classes. What symbol do you need to use in the visibility parameter? – ( minus or dash in front the class or the attribute) like

Hint by

We need "private" visibility here



In which of these situations is a class diagram helpful?

When you need to visually represent relations within your program

Which relation in class diagrams describes one class as a part of the other, while classes have a separate lifespan?

Aggregates are things like Reddit. Everything from Everyone is posted, all different accounts. It's an "aggregate" of people of different ages.

The correct answer is: **Aggregation**

### Explanation:

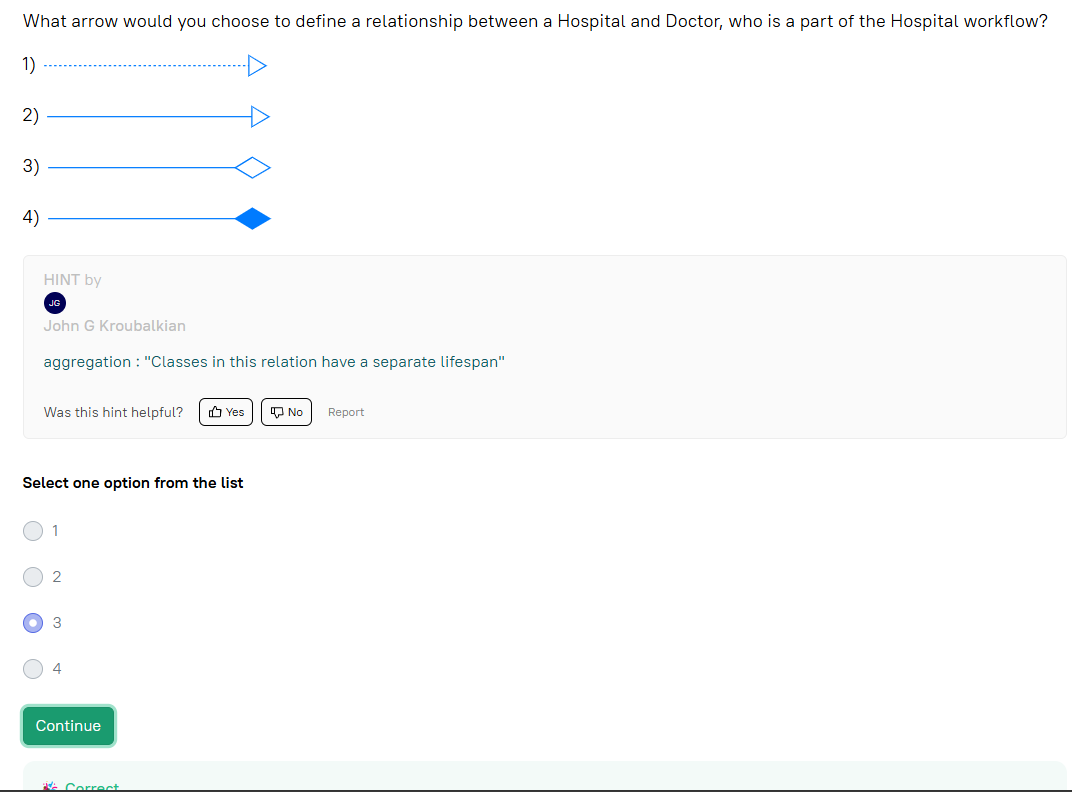
In **Aggregation**, one class is a part of another, but both can have independent lifespans. It's a "has-a" relationship where the contained object can exist without the container.

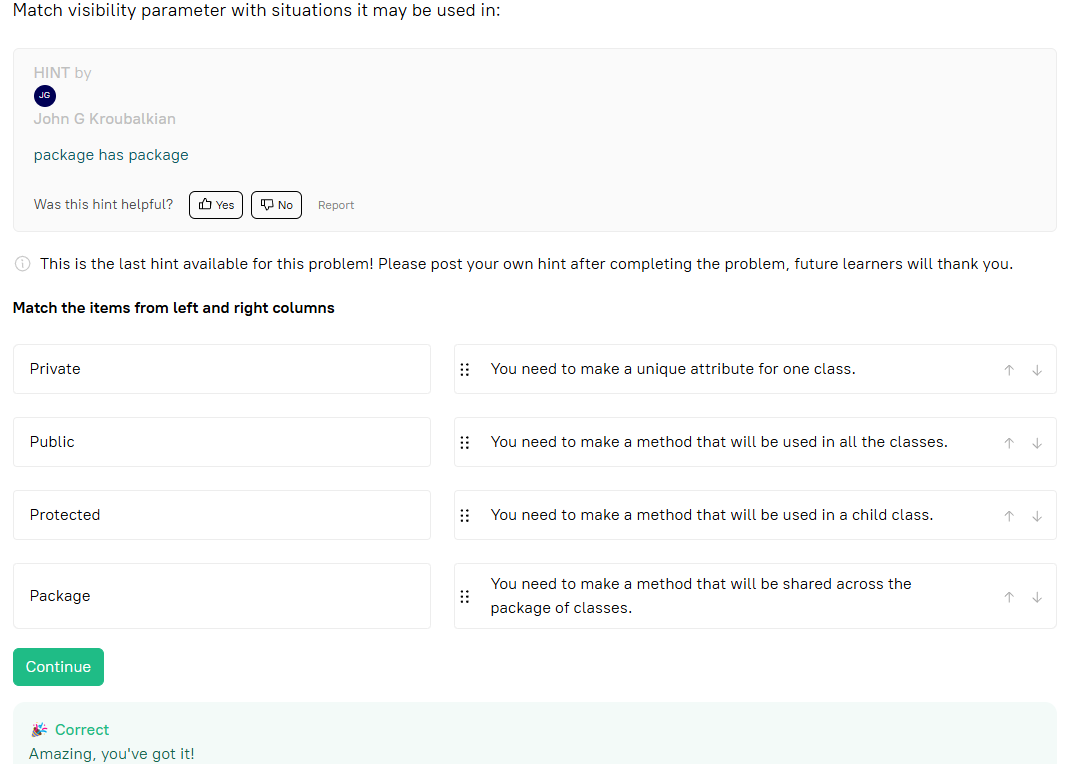
* **Composition** implies ownership and a dependent lifecycle (e.g., a house and its rooms — if the house is destroyed, the rooms are too).
* **Aggregation** is looser — like a university and students. Students can exist even if the university closes.
* **Association** is a general relationship.
* **Realization** is used between interfaces and the classes that implement them.

✅ So, **Aggregation** is the correct relation described.

What is Class Relationship?

A concept that defines connections between classes.

****



You are working on an application that has different types of screens. They are all represented by different classes. But there is a class called Reference that acts as a blueprint for other screen classes. Now, you need to depict these classes in a class diagram. Which relationship best describes the connection between Reference and other classes?

### Correct answer: ****Realization****

### Why?

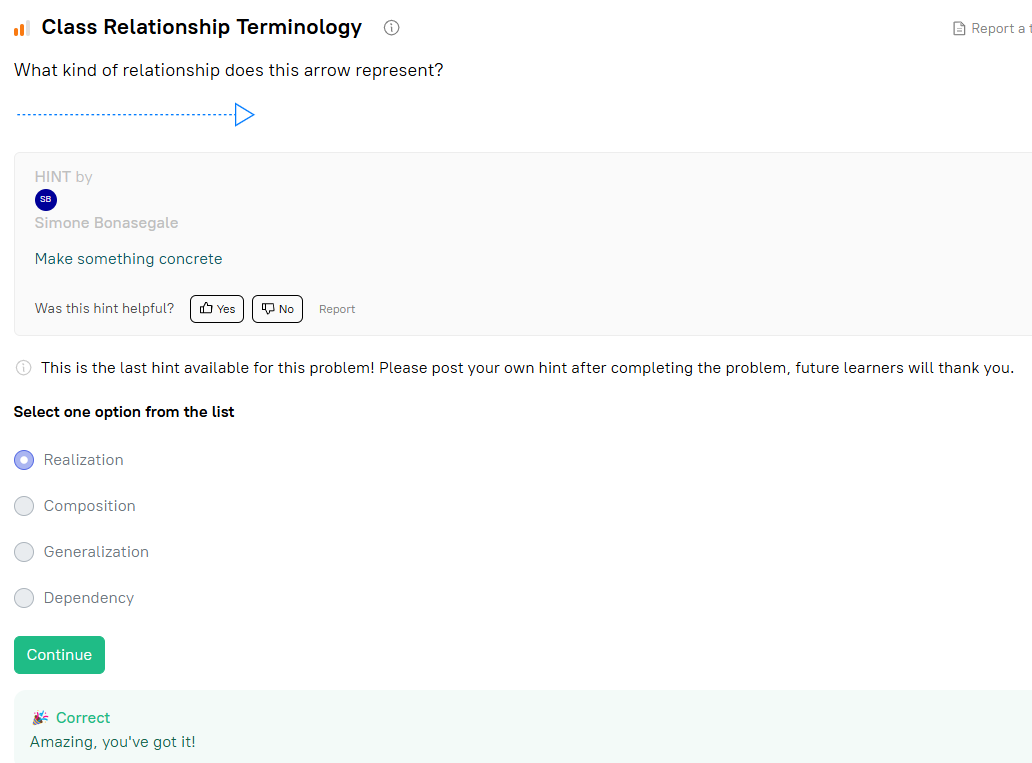
* **Realization** is the relationship between:
  + **An interface or abstract class** (like Reference)
  + And **concrete classes** that implement or extend it (your different screen classes).
* It represents a **"implements"** relationship in UML class diagrams.

So in your case:

* Reference defines **what** should be done.
* Other screen classes define **how** it’s done.

What is Class Diagram?

A UML diagram that visually represents object-oriented structure.

****

**Realization** is used when a class implements an interface or an abstract class, typically represented with a dashed line and a closed arrowhead in UML diagrams.

### Generalization:

* Represents inheritance.
* **A subclass ("child")** inherits from a **superclass ("parent")**.
* The subclass "is a" type of the superclass.

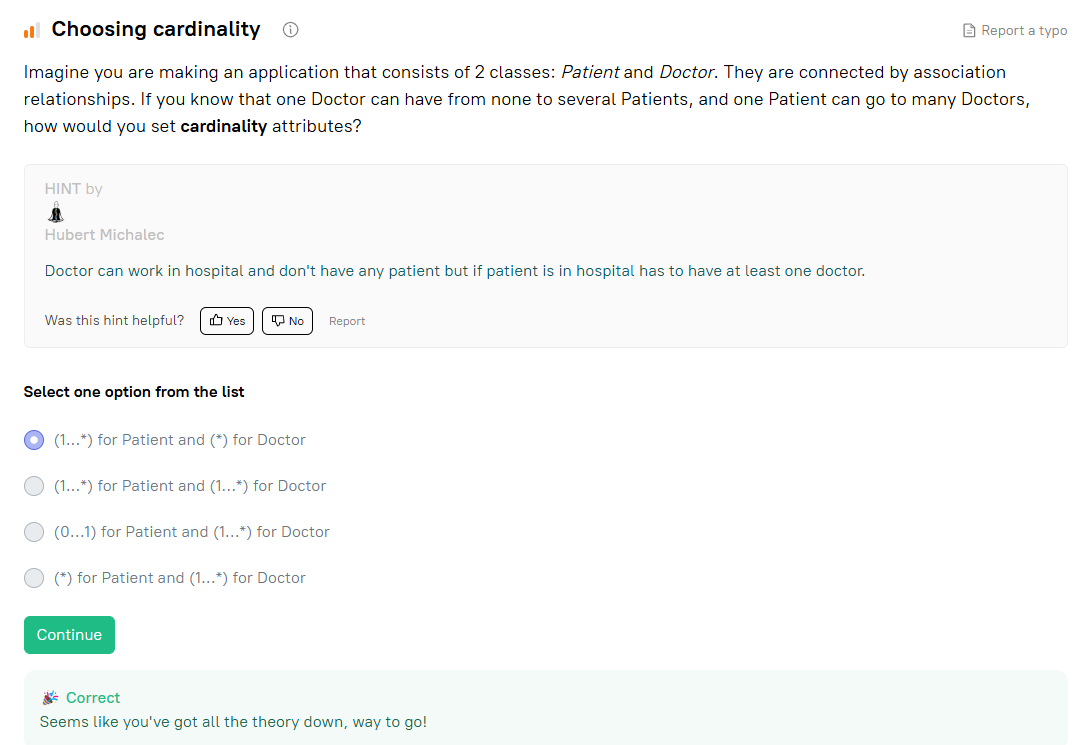
### Dependency:

* A weaker relationship.
* One class uses another class, typically as a parameter, local variable, or return type.
* It's temporary; if the dependent class disappears, it affects the dependent class, but not always critically.

### Realization:

* Relates to interfaces.
* A class implements an interface, providing concrete behavior for the methods defined in the interface.

So, **Generalization** is inheritance, **Dependency** is a loose coupling, and **Realization** is implementing an interface.

****