TopHat activities

- Please join and indicate that you have done so:

- https://app.tophat.com/login
  Join: **147651**

UNIVERSITY OF LEICESTER

CLASS DIAGRAMS

CO1106 Requirements Engineering and **Professional Practice**

Dr Matthias Heintz , Prof. Shigang Yue
(mmh21@leicester.ac.uk) (sy237@Leicester.ac.uk)

Schedule

Week	Start Date	Monday / Wednesday - Lecture	Thursday / Friday - Surgery	Assessment
26	15/01/2024	Introduction & Why Requirements?	Icebreaker activity for groups & work on Project Description	
27	22/01/2024	Requirements gathering (Quan. & Qual. User Studies)	Work on requirements gathering for Assessment 1	
28	29/01/2024	Functional Requirements	Work on building list of funct. requirements for Assessment 1	
29	05/02/2024	Non-Functional Requirements	Work on building list of non-funct. requirements for Assessment 1	
30	12/02/2024	Overview of UML; Use Case diagrams and descriptions	Work on Use Case diagram and Use Case description for Assessment 1	
31	19/02/2024	Basics of git version control	Checkout and setup group git repository and set up Weekly Log .md file	Assessment 1 (50%)
32	26/02/2024	More advanced git topics	Work on reworked list of functional requirements	
33	04/03/2024	Class Diagrams	Work on Class diagram	
34	11/03/2024	Class Modelling	Rework Class diagram	
35	18/03/2024	Sketching and Lo-fi prototyping	Work on wireframes/lo-fi prototypes	
36	25/03/2024	Software Laws & Professionalism	**none**	Effective use of Git (10%)
37-40	01/04/2024	break	break	
41	29/04/2024	none	none	Blackboard Test (40%)

Matthias
Shigang

Session objectives

- At the end of the lecture you will be able to:
  - *Explain what is a Class model, why they’re useful for design/implementation of an application*
  - *Apply the notational aspects of UML Class Diagrams to create your own*
  - *Recall some examples of Class Diagrams*

Group coursework in CO1106

- Main group project
  - ***Part 1 (50% - due 23rd February)***
    - Project description (10%)
    - Quantitative and qualitative studies (10%)
    - Written requirements (20%)
    - Use Case UML Diagram and Use Case Description (10%)
  - ***Part 2 (10% - due 27th March)***
    - Effective usage of git version control

Group Coursework Part 2 – Effective use of git version control

- Your group will utilise a git repository in order to manage/submit any files produced as part of the second part of the group project for CO1106 (details of how to access the repository will be provided to you in the tutorial of Week 6).
- Groups should **make frequent usage** of their group repository - any shared files that you work on (for example, the .md files containing functional requirements and use case descriptions) should be added to the repository as soon as they are made, with regular changes being committed by group members until that particular file is finished. Each group will be responsible for coordinating their git usage.

Group Coursework Part 2 – Instructions

- A maximum of 3 marks are available, depending on **how effectively your group used git**. We will decide the number of marks you receive by inspecting the contents of your repository as well as the commit history of your repository:
  - For 1 mark, at least one member of your group needs to make a commit each week; no advanced features (i.e., branching/merging) have been used, and commit descriptions (included when you made the commit) may be non-descriptive and not give a good idea of the changes included in a particular commit.
  - For 2 marks, multiple commits should be made each week, and an initial (possibly incomplete) version of the artefact worked on during each tutorial session needs to be submitted in the week it was worked on. Commit messages must be descriptive (but succinct) and give a good idea of the changes that have been committed.
  - For 3 marks, you must satisfy all points from the previous two bullets, and it should be apparent from your commit log that all members of the group have

Group Coursework Part 2 – Weekly log

- Each group must also produce **a 'Weekly Log' (.md format)** that is **updated** with the following contents in **each** of the **week**s 6-10 (5 weeks in total):
  - A 'beginning of week' entry containing a summary of the work that the group plans to complete during that week, along with a breakdown of which members will complete which tasks. The beginning of week entry for each week should be produced by the group; for example, during your first groupwork meeting of that week.
  - An 'end of week' entry which lists the tasks that each member of the group has completed; any outstanding work; and any other additional information that your group feel is relevant to add .

Group Coursework Part 2 – Weekly log (continued)

- Each group will be responsible for **designing the Markdown structure** of their Weekly Log, ensuring that it is easy to read and maintained properly. The entries in the Weekly Log will be checked on a weekly basis (the entry for Week X will be checked by us during Week X+1). For the entries of Week X, there is a maximum of 0.75 marks available (up to a total of 3 marks for weekly updates):
  - 0.25 marks depending on whether both the 'beginning of week' and 'end of week' entries have actually been added to the log (if either one is missing, you receive 0 marks for that week)
  - 0.5 marks will be awarded depending on the quality of the entry (is it descriptive enough? does it contain all the information listed above?)
  - An additional mark out of 4 will be awarded based on the readability/quality of the Weekly Log document.

Marking rubrics

Markdown Usage, Marking Rubric					
Fail	Poor	Requires Improvements	Satisfactory	Good	Excellent
0	0.5	1	2	3	4
No serious attempt is made.	Required information is contained in the document but no Markdown syntax has been utilised in order to improve readability.	A limited amount of basic Markdown syntax has been used to produce a document that is readable but not aesthetically pleasing or easy to navigate. There may be some syntactical errors in the Markdown usage that cause the document to look untidy.	Basic Markdown syntax has been used to reasonable effect. The produced document is navigable and its contents is clearly displayed. Most audiences should have little to no problem with understanding the document, but it is not particularly aesthetically pleasing.	Same as Satisfactory, but with a number of more advanced Markdown features used in order to improve the readability of the document. Any member of the intended audience would be able to navigate the document easily.	Same as Good, but a clear effort to make the document aesthetically pleasing as well as easily navigable has been made.

Refresher:
A Class Diagram models what type of view?

A system model is an '***abstract representation***' of a system that takes a particular perspective of that system and **ignores irrelevant details.**

A ***view*** is a perspective of a system, and ***view models*** allow us to focus on:

- What a system **does** for its users/stakeholders (**User** view)
- How a system is **structured** internally (**Structural** view)
- How a system **reacts** to external influences and how its components **interact** with each other (**Behavioural** view)

https://app.tophat.com/login
Join: **147651**

Unified Modelling Language (UML): Diagram types

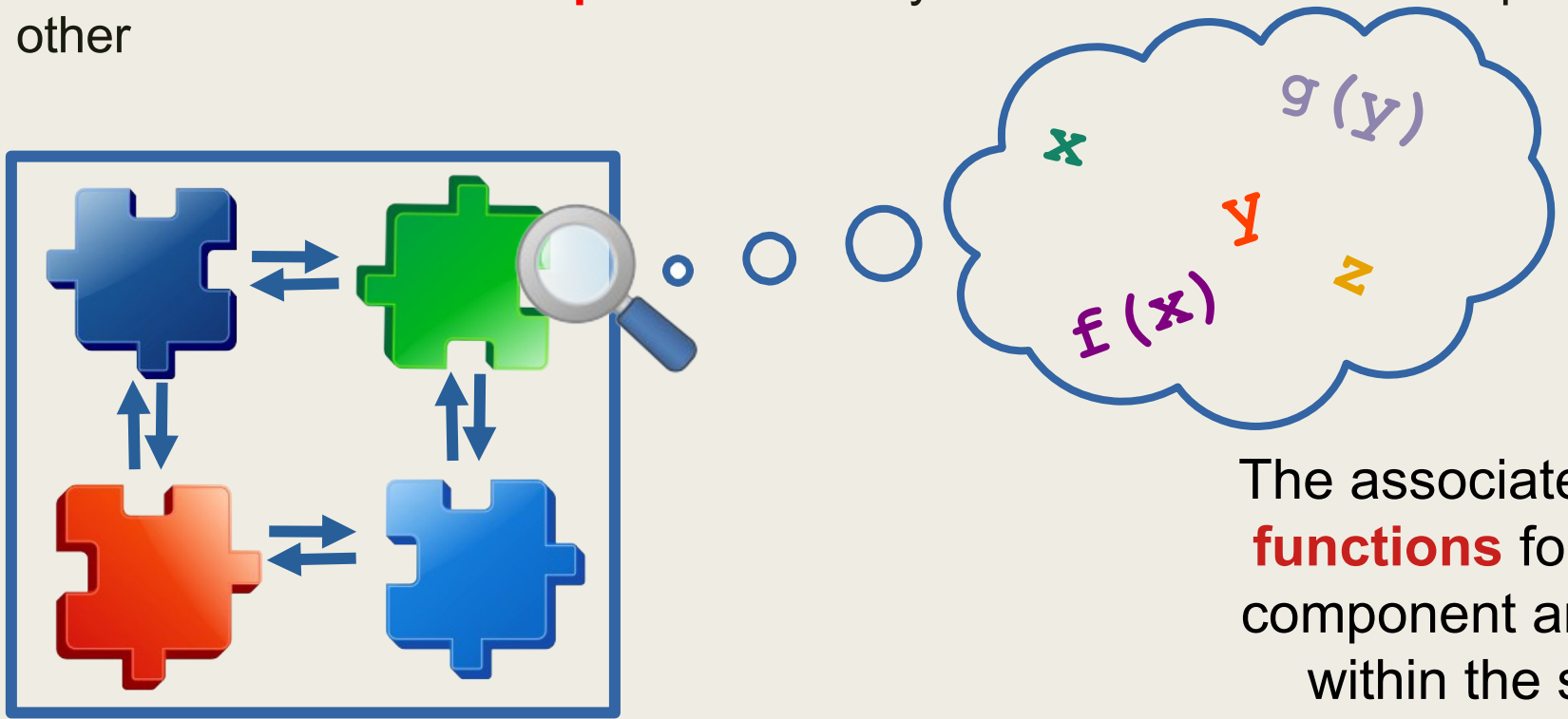
UML diagram types generally belong to one of three different categories:

- **Functional:** Concerned with a system's functionality from users' viewpoint (i.e., what does the system provide to a user, or a user provide to it?)
  => Functionalities of the system
  - ***Use Case Diagrams***
- **Static/Structural:** Concerned with objects that exist within a system's boundary and how they relate to one another
  => Static aspects of the system that do not change during the program execution
  - ***Class Diagrams***
- **Dynamic/Behavioural**: Concerned with the behaviour of a system and its components over time
  => Dynamic aspects of the system: What happens during program execution
  - *Sequence Diagrams, Activity Diagrams, State Diagrams*

STRUCTURAL VIEW

Structural view

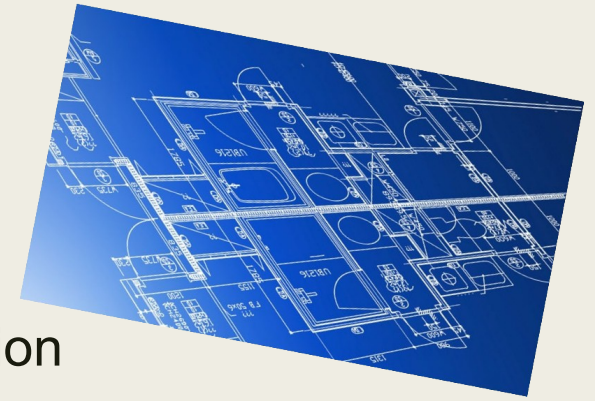
- A transition from "**what**" the system must do (i.e., use- case diagrams) to "**how**" the system will do it
- Concerned with the **components** of a system and their relationships to each other



The associated **variables** and **functions** for each individual component are also important within the structural view

CLASSES

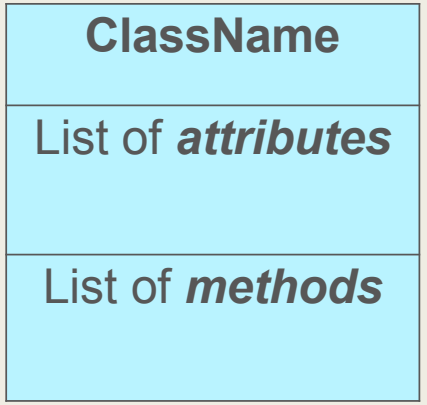
Class models



- A class model is a **blueprint** of the **static structure** of an application – it is closely linked to Object-Oriented style programming
- They define the **kinds of objects** that exist within the system as well as their associations to each other
- They serve as a good starting point for developers to begin developing an application, but **ignore implementation- specific details**
- UML provides the **Class Diagram** notation which can be used to visualise class models

Notational aspects: Classes

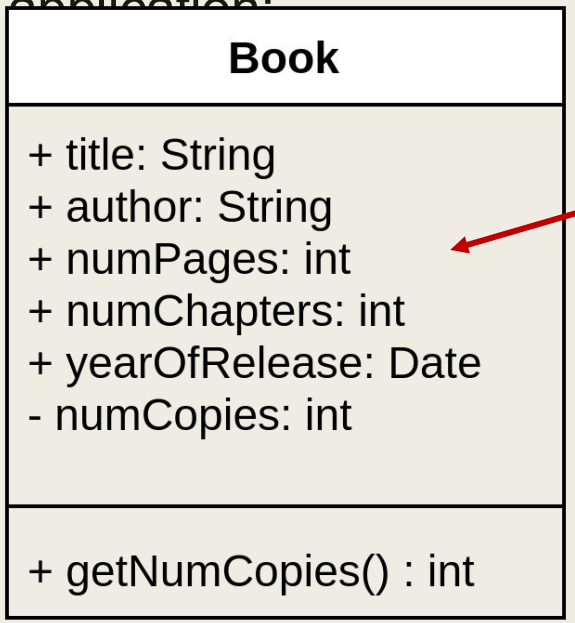
- Each ‘component’ of our application typically translates to a ***Class*** in a Class Diagram of that application
- Each class in a class diagram has a:
  - *Class name*
  - *A (possibly empty) list of associated attributes (variables)*
  - *A (possibly empty) list of associated methods (functions)*



- Each Class represents a ***type of entity*** within the system/application; an ***instance*** of a class is an individual entity that belongs to that class, and has particular values assigned to each of the class attributes

More on classes: Attributes

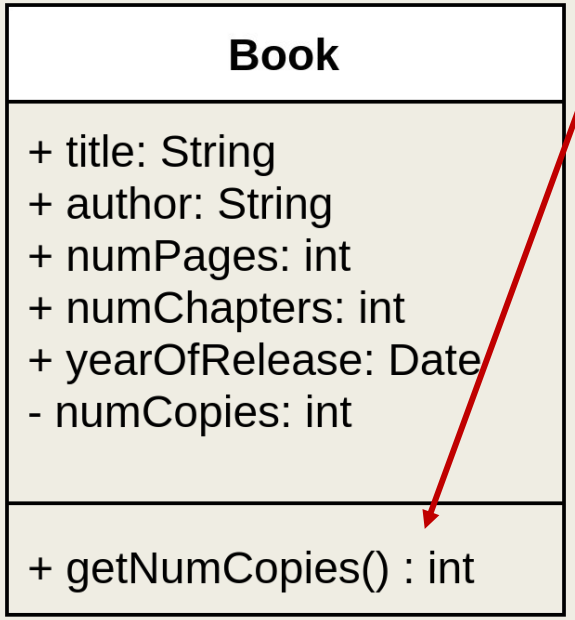
- Classes typically represent ***concepts*** or ***entities*** that comprise the system we are designing, e.g., a 'Book' would be a sensible class in an eLibrary application:



- The collection of ***attributes*** contained in the middle third of the class are pieces of information related to any book in the eLibrary
- The format for each attribute is:
  (accessModifier) attributeName*: *attributeType
  - (accessModifier) is either '+' (i.e., public) if the value for that attribute can be accessed by an instance of any other class, or '-' (i.e., private) otherwise
  - **attributeType** indicates the data-type of the attribute

More on classes: Methods

- Class methods (contained in the final third of a class) are functions that return some value associated with the class, or perform some task related to the class



Note that we use a public method ‘getNumCopies()’ to obtain the value of private attribute ‘numCopies’

- The format for each method is:
  (*accessModifier*) *methodName*: *returnType*
  - (accessModifier) is either ‘+’ (i.e., public) if the method can be executed by other classes, or ‘-’ (i.e., private) otherwise
  - The pair of parentheses ‘()’ indicates that the method should be implemented as function, and can optionally contain arguments
  - ***returnType*** indicates the type of value that is returned by the method (e.g., int, String, void, etc.)

More examples of Classes

BankAccount
+ accountHolder: String - balance: double - interestRate: double
+ getBalance() : double + makeDeposit(amount : double) : void + makeWithdrawl(amount : double) : void + updateInterest(newRate : double) : void

- Typically, we keep attributes whose value depends on the value of other attributes (such as ‘price’) private
- The value of those attributes is then modified only via public methods, e.g., ‘addFood()’ and ‘addDrink()’

- Note that it is assumed that any class method has access to all class attributes, regardless of access modifiers

RestaurantOrder
- food : List - drinks : List - price : double + waiter : String + tableNum : integer
+ addFood(item : String) : void + addDrink(item : String) : void + getPrice() : double + getFoodList() : List + getDrinksList() : List

Class vs. Instance (object)

- A class is like **a template or a contract** that specifies the generic features and actions of a particular type of entity...

Person
+ name : String + gender : String - dateOfBirth: Date + Interests : List
+ getAge(): Integer

VS.

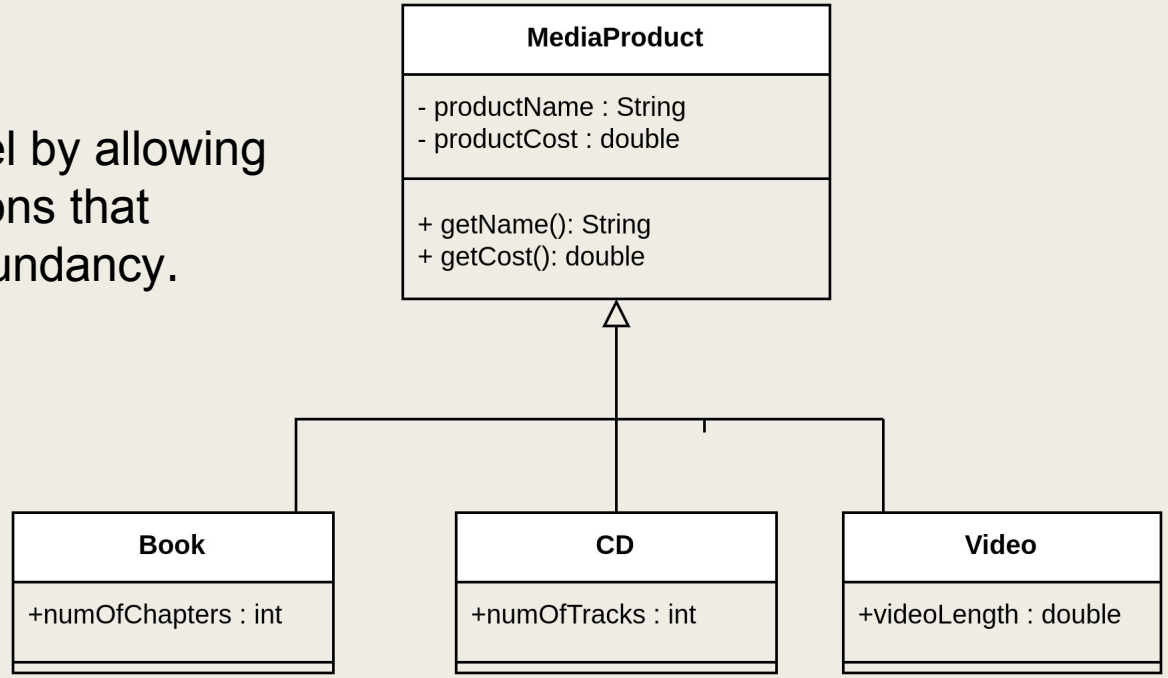
Name: Keith
Gender: Male
DOB: 21/03/1975
Interests:
Photography
Travelling
Football

- An **instance of a class** is a **realisation of that contract**: it is a specific object belonging to the class with concrete values for each attribute, and the ability to perform all associated class operations

Class generalisation

- Class generalisation is used when a certain class should contain the same attributes and operations as another but also has additional functionality
- The **child class** inherits all attributes and operations of the **parent class**:

Generalisation simplifies the model by allowing for reuse of attributes and operations that already exists, i.e., it removes redundancy.



ASSOCIATIONS

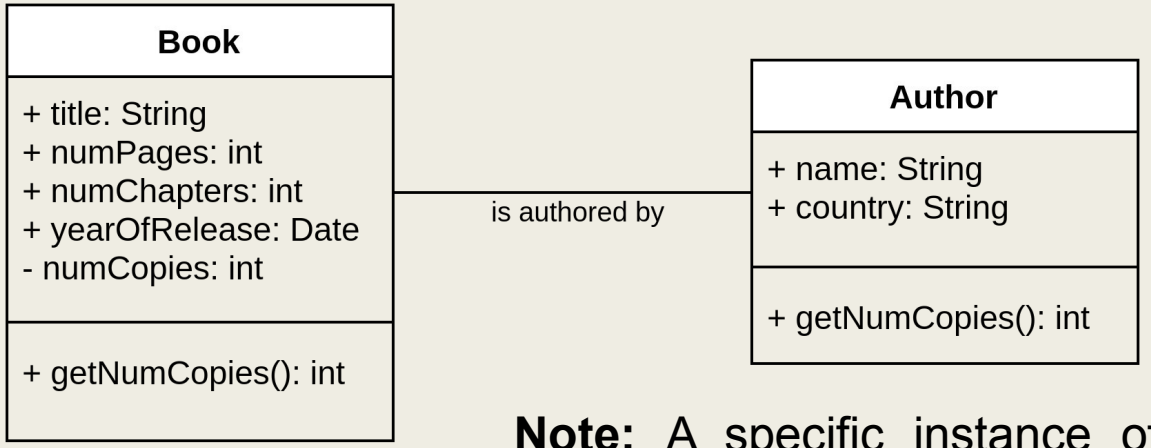
Notational aspects: Class associations

- So far, we've seen only singular classes, but usually a class model consists of **multiple classes that are related in some way**
- Think about the 'Book' class – a book has at least one author, so me may want to record authors in our system as well:

- We represent this relationship using the '**association**' notation: a line between the classes with a sensible annotation

More on associations

- Notice that ‘author’ is no longer listed as an attribute of ‘Book’: the association simply implies that this information should be recorded by our system...
- This could be as a ‘author’ attribute in class Book, or as a list ‘Books’ stored in the Author class: this is an **implementation-specific** design choice which we should worry about when we finally program the system

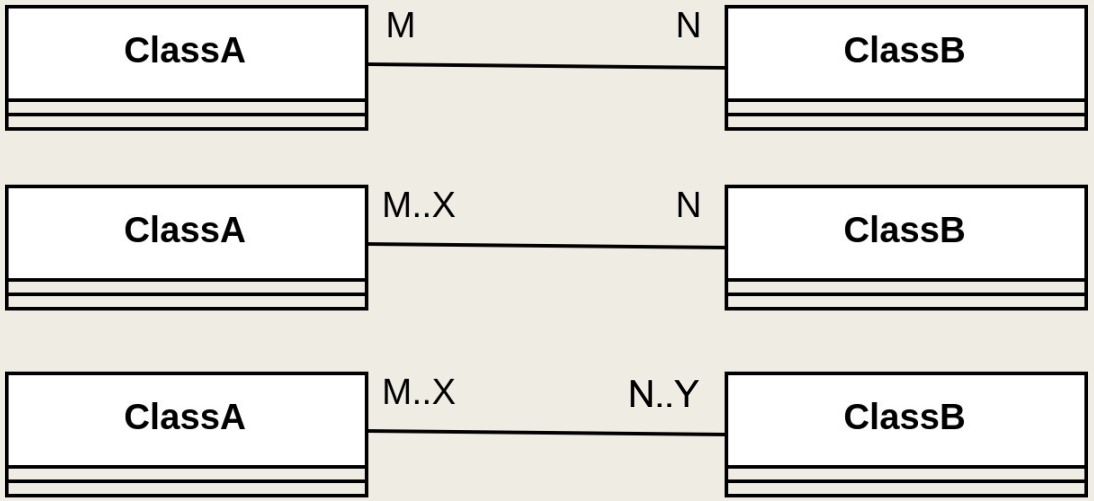


Note: A specific instance of an association between an instance of a Class A and one of Class B is called **a link**

MULTIPLICITIES

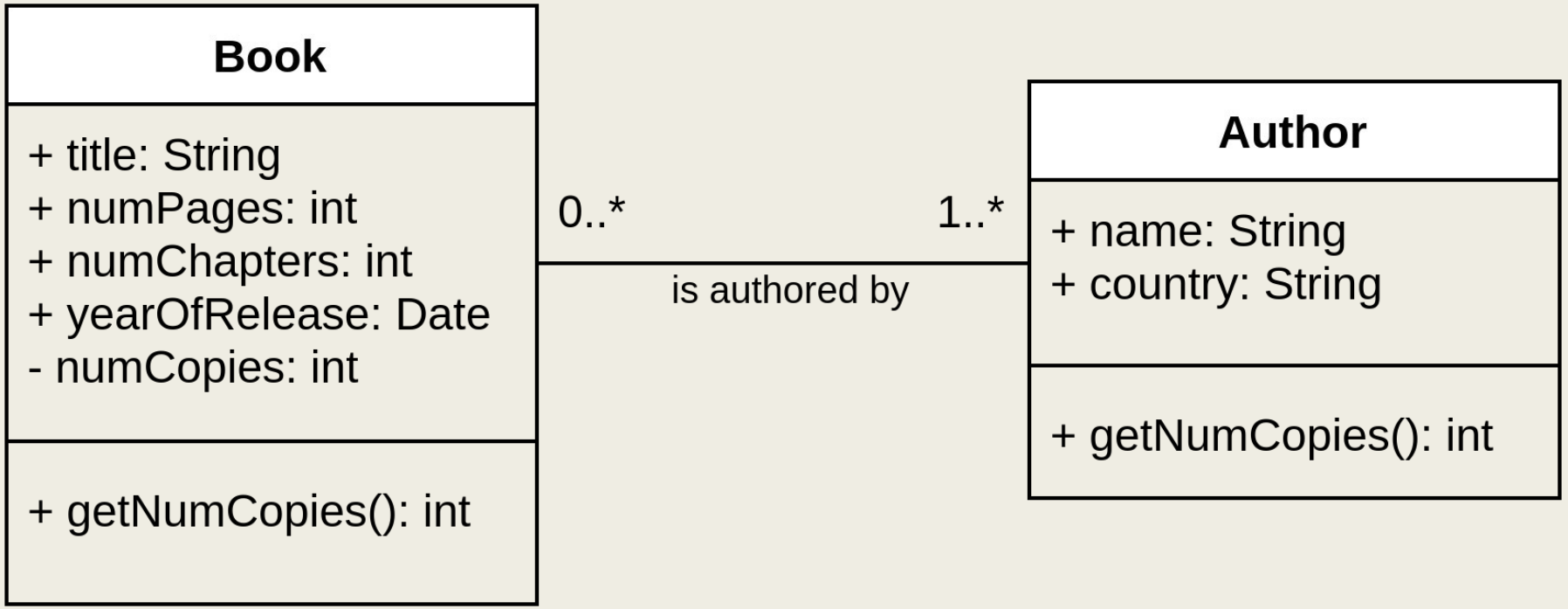
Associations: multiplicities

- If we wish to indicate a precise bound on the number of links that a particular instance of a class participates in, we use **multiplicities**:



- M and N in line one can be replaced with any integer greater than 1;
- X and Y are natural numbers greater than M/N (>=0), or X,Y = * which means that there is no bound on the number of links an instance can participate in.

Multiplicities: an example

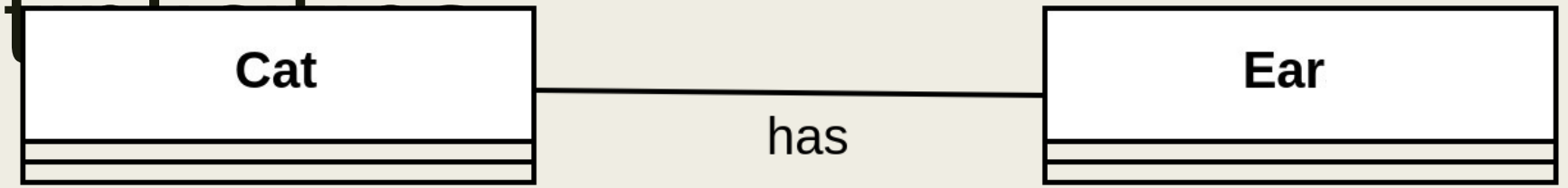


- This indicates that each book has ≥ 1 author, and that an author can have written ≥ 0 (i.e., any natural number of) books (a so-called '**many-to-many' relationship**)

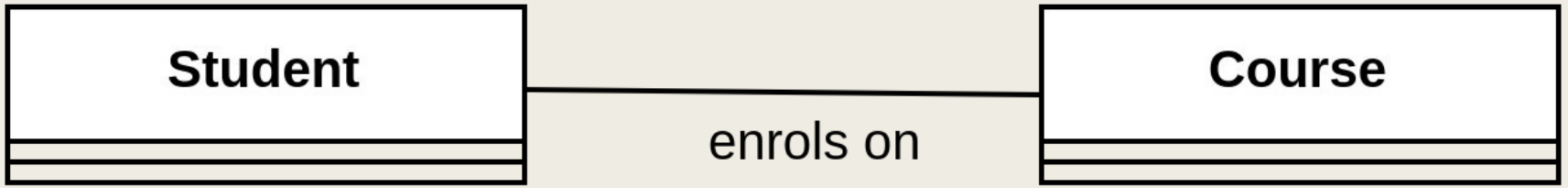
MULTIPLICITIES

Exercise

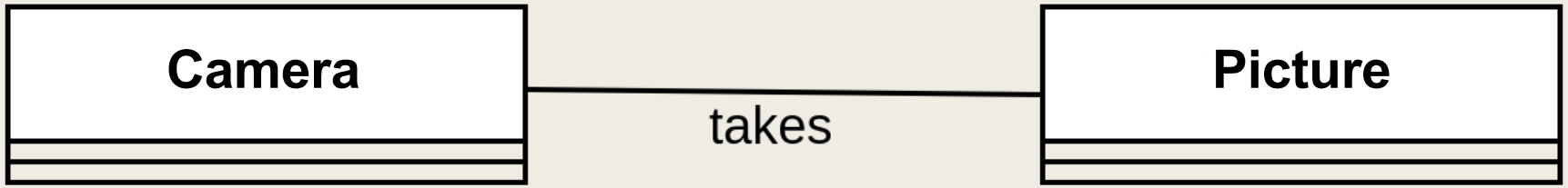
Exercise: Identify the multiplicities



A cat has 2 ears and each ear belongs to exactly 1 cat

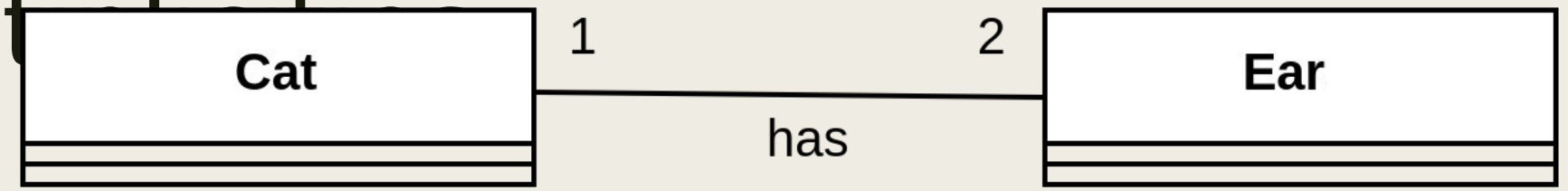


A student enrols on exactly 1 course and each course has at least 1 student

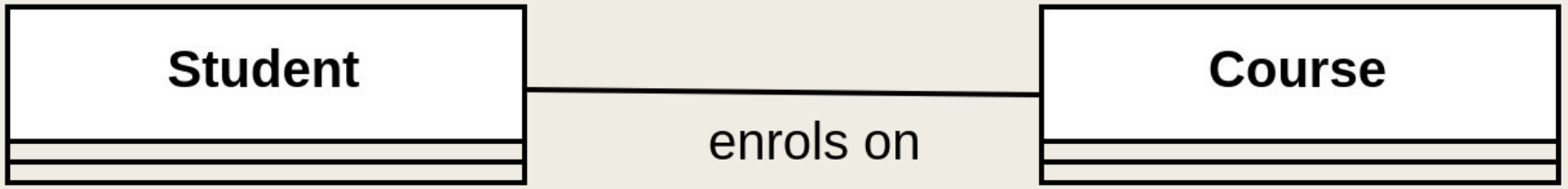


A camera can take 0 or more pictures but each picture is taken by exactly one camera

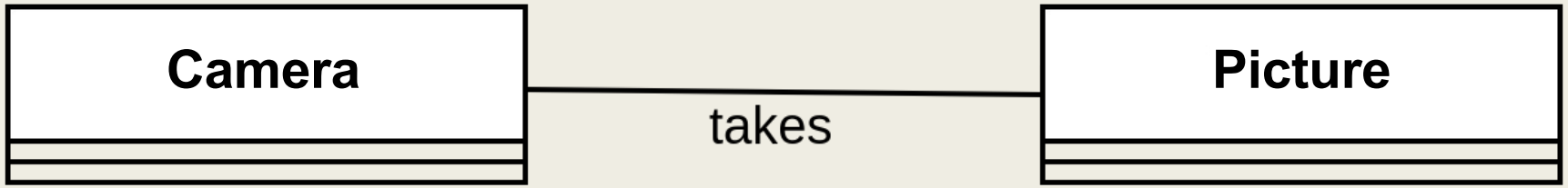
Exercise: Identify the multiplicities



A cat has 2 ears and each ear belongs to exactly 1 cat

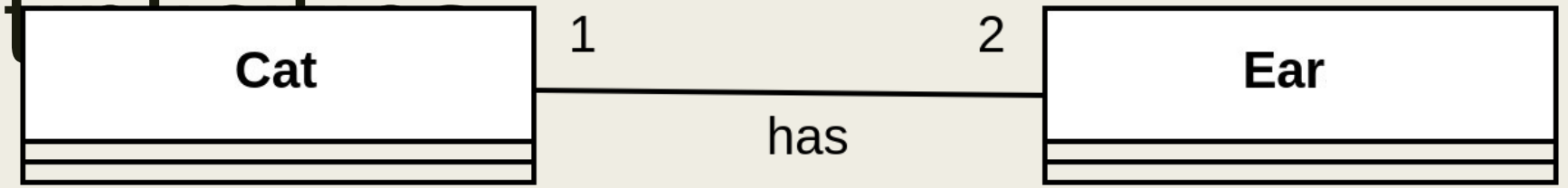


A student enrols on exactly 1 course and each course has at least 1 student

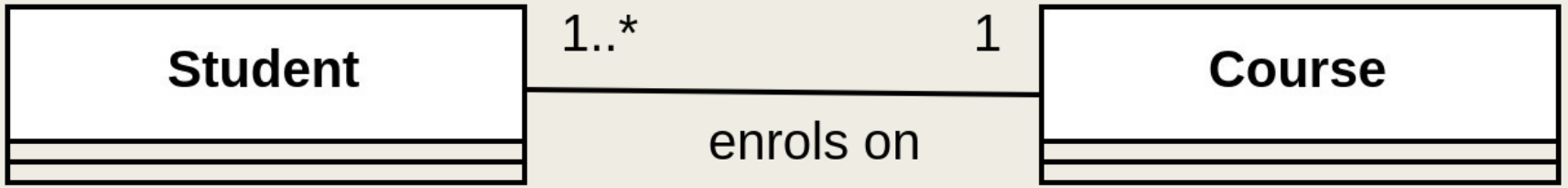


A camera can take 0 or more pictures but each picture is taken by exactly one camera

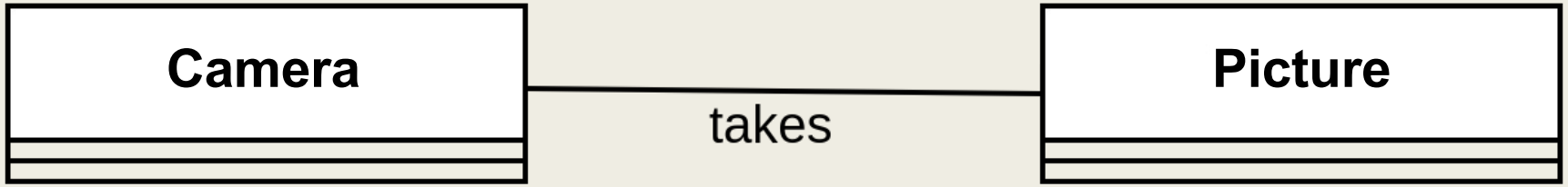
Exercise: Identify the multiplicities



A cat has 2 ears and each ear belongs to exactly 1 cat

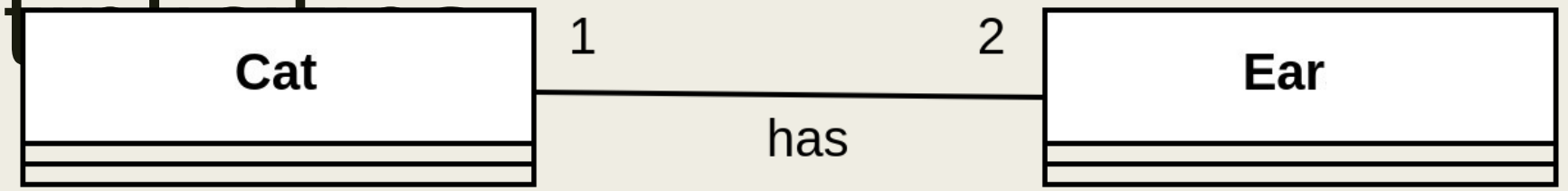


A student enrols on exactly 1 course and each course has at least 1 student

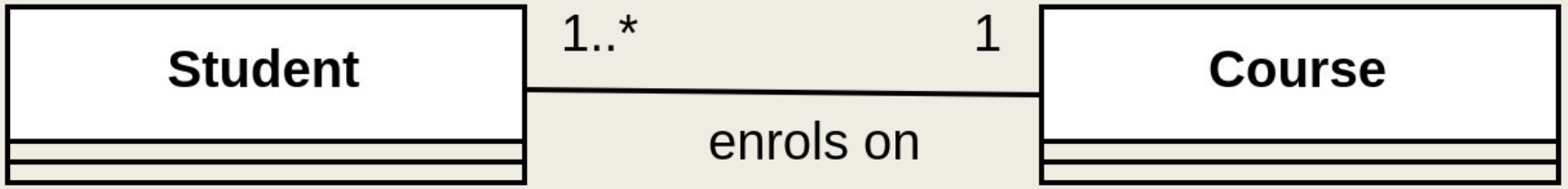


A camera can take 0 or more pictures but each picture is taken by exactly one camera

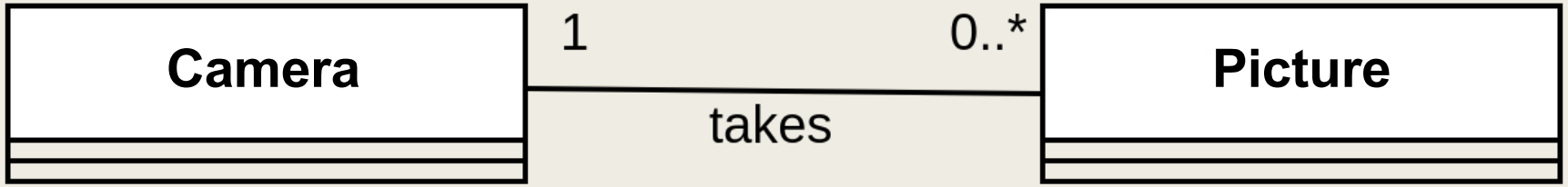
Exercise: Identify the multiplicities



A cat has 2 ears and each ear belongs to exactly 1 cat

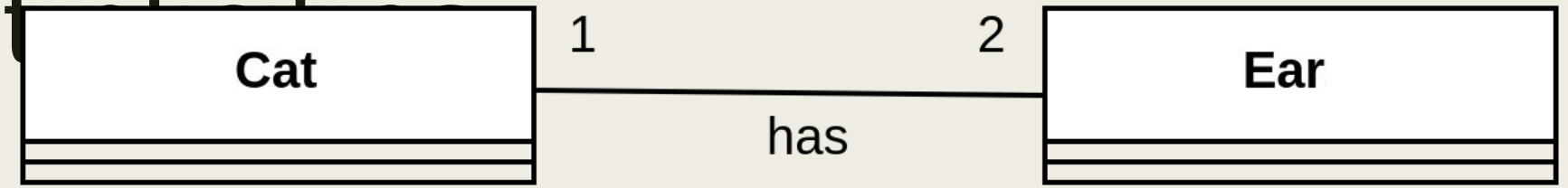


A student enrols on exactly 1 course and each course has at least 1 student

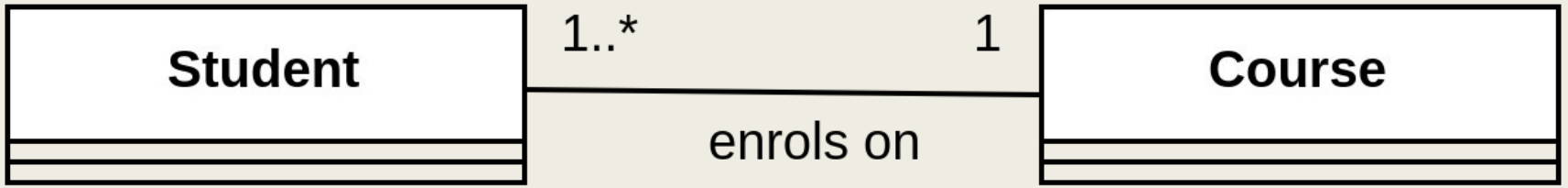


A camera can take 0 or more pictures but each picture is taken by exactly one camera

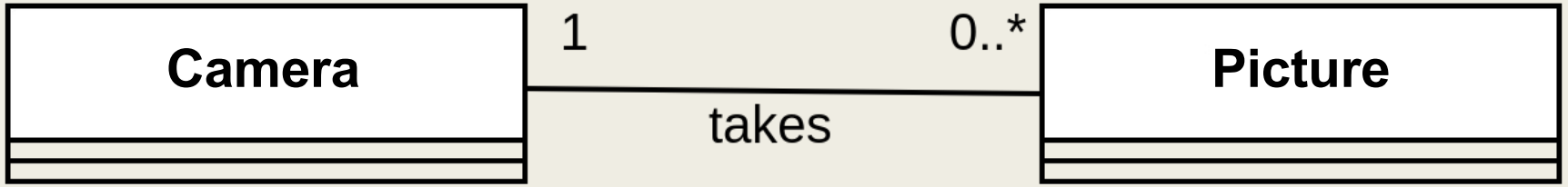
Exercise: Identify the multiplicities



A cat has 2 ears and each ear belongs to exactly 1 cat



A student enrols on exactly 1 course and each course has at least 1 student

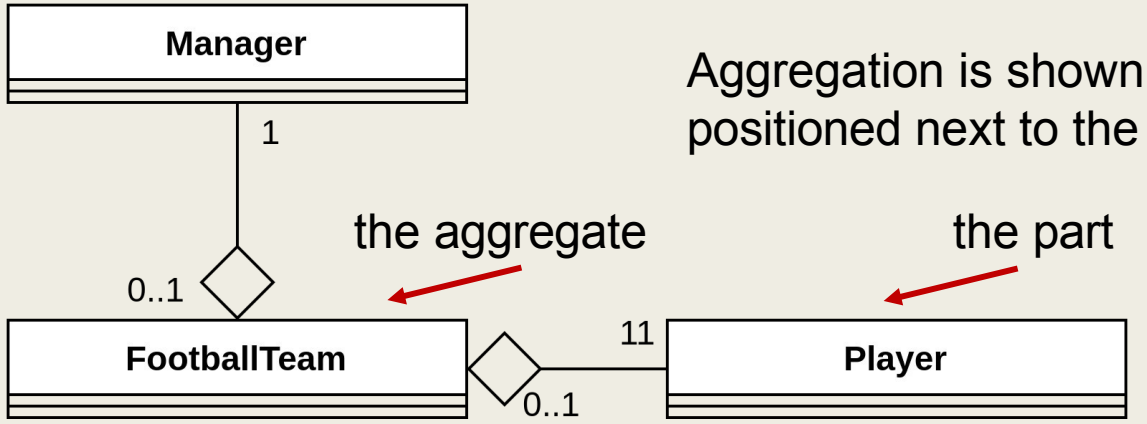


A camera can take 0 or more pictures but each picture is taken by exactly one camera

ASSOCIATIONS

Other association types: Aggregation

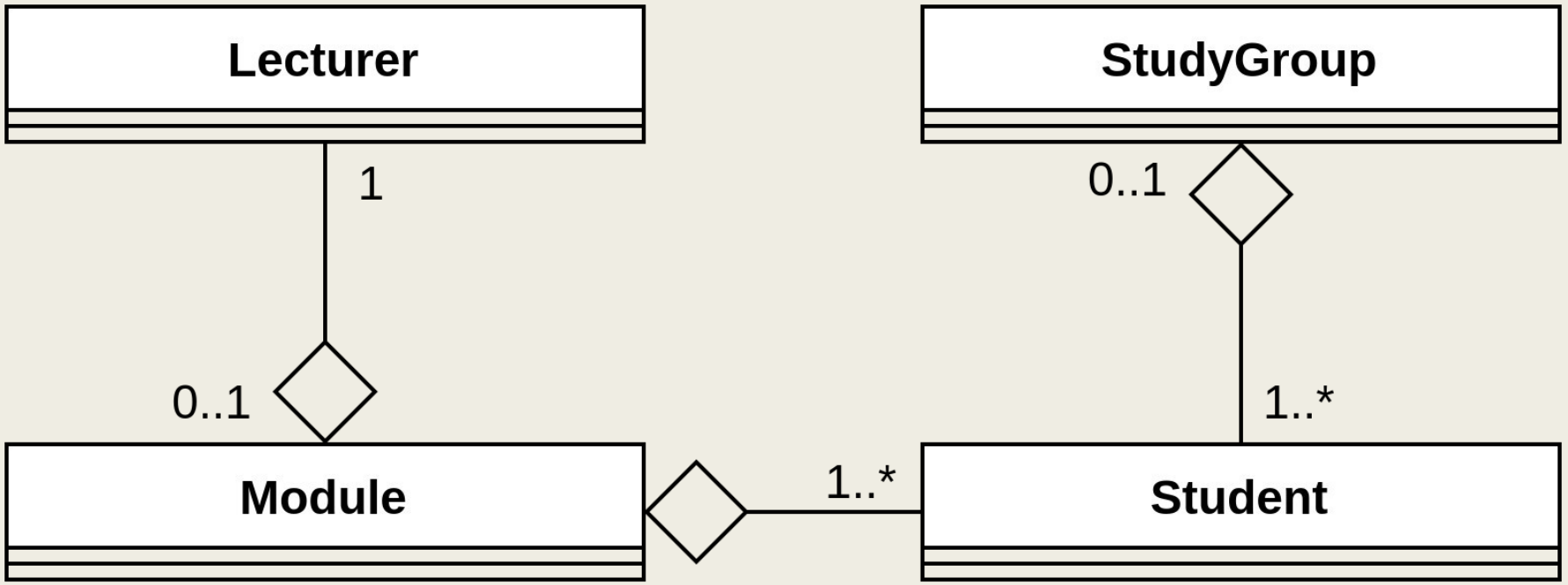
- Sometimes, we want to explicitly show that a class B is **an integral part of** a class A
- The first way is **aggregation**, which implies that instances of class B can exist without being associated to an instance of class A, but not the other way round



Aggregation is shown by a **white diamond** positioned next to the aggregate class

Note that a class can be a part of **more than 1 other class**, since the aggregate class **does not strongly own** the part class

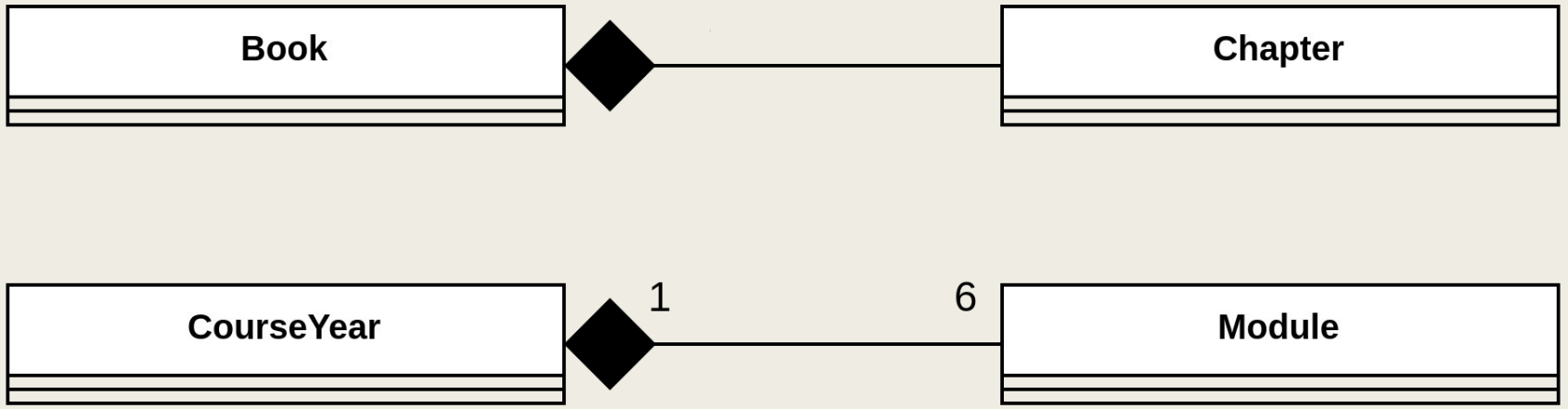
Another aggregation example:



- A **Module** needs **exactly 1 Lecturer** and **≥ 1** enrolled **Student** to exist
- Each **Student** can be enrolled in 0 to 6 **Modules** at a given time
- A **Lecturer** convenes **0 or 1** modules at a given time
- A **StudyGroup** needs at **≥ 1 student** and a student can belong to at most 1 study group

Other association types: Composition

- **Composition** is the second way of showing one class is part of another, but this time, the part **cannot exist** without the aggregate
- It is represented by a **black diamond next to the parent class**
- **Examples:**



- **Key idea:** if we delete an instance of the parent class, then the composite instances of the child class is also deleted...

ASSOCIATIONS

Exercise

Exercise (5 min): identifying association types

- Draw the aggregation/compositions based on the descriptions below
- **Aggregation:** An **Album** is an aggregation of an **Artist** and **Song**

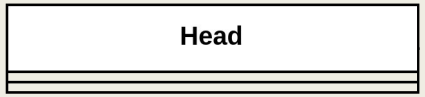
Each album must have exactly 1 artist, and each artist can have written 0 or more albums.

Each album must consist of 1 or more songs, and each song may belong to 0 or more albums.

- **Composition:** A **Dog** is a composition of various body parts

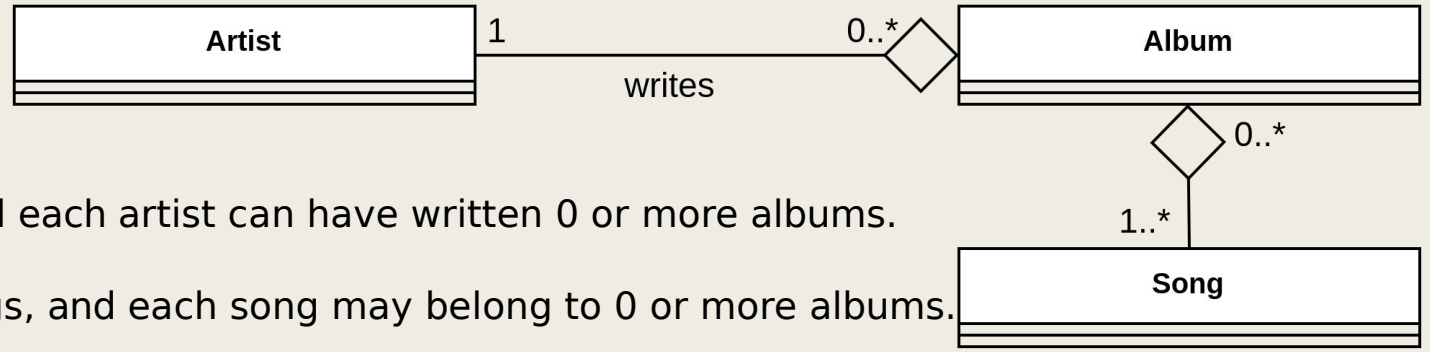
A dog must have 1 tail,

1 head and 4 legs



Exercise (5 min): identifying association types

- Draw the aggregation/compositions based on the descriptions below
- **Aggregation:** An **Album** is an aggregation of an **Artist** and **Song**



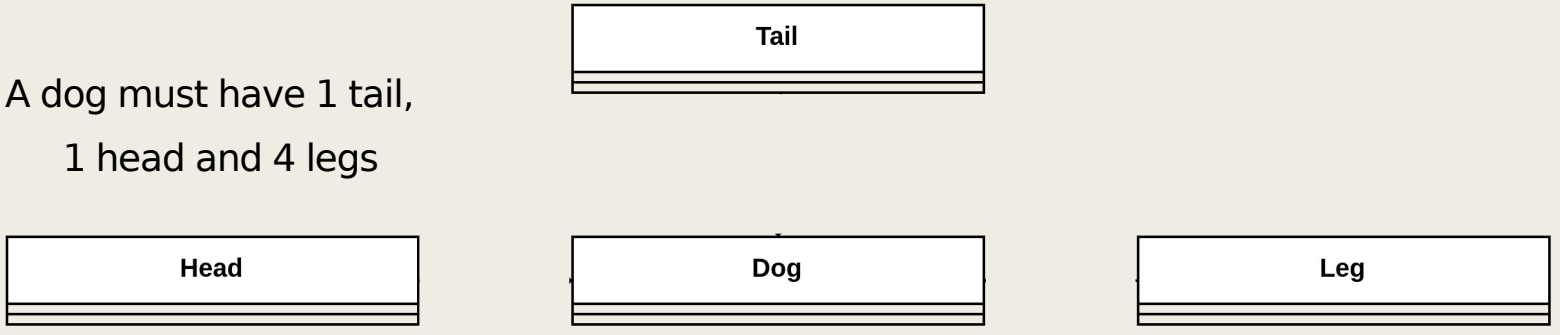
Each album must have exactly 1 artist, and each artist can have written 0 or more albums.

Each album must consist of 1 or more songs, and each song may belong to 0 or more albums.

- **Composition:** A **Dog** is a composition of various body parts

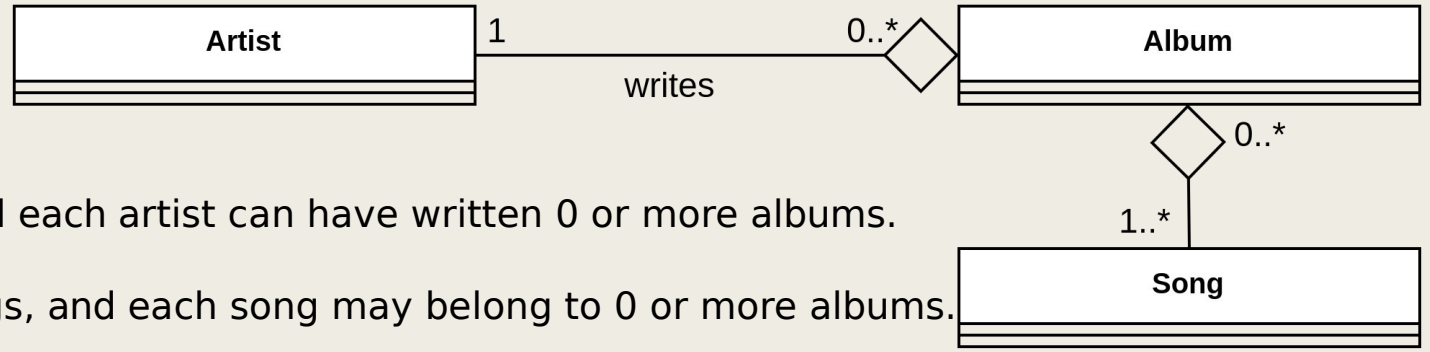
A dog must have 1 tail,

1 head and 4 legs



Exercise (5 min): identifying association types

- Draw the aggregation/compositions based on the descriptions below
- **Aggregation:** An **Album** is an aggregation of an **Artist** and **Song**



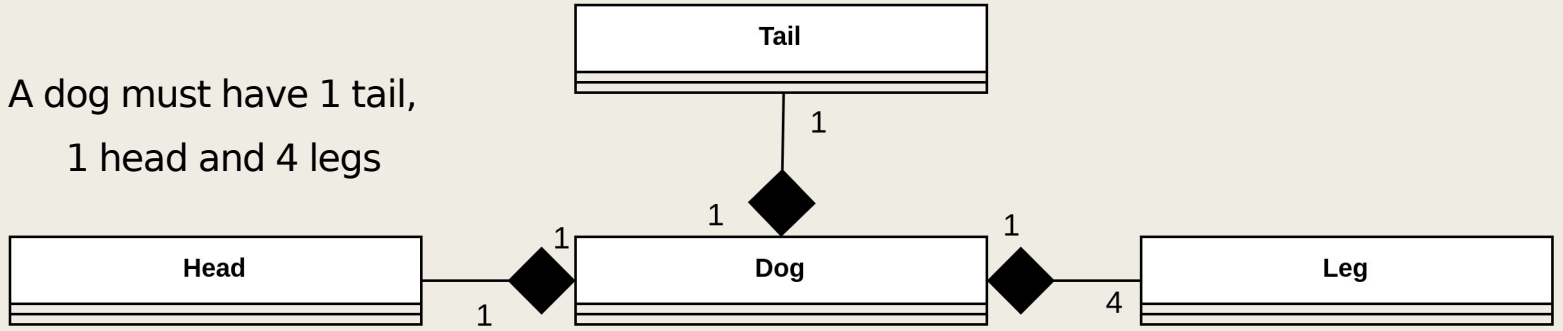
Each album must have exactly 1 artist, and each artist can have written 0 or more albums.

Each album must consist of 1 or more songs, and each song may belong to 0 or more albums.

- **Composition:** A **Dog** is a composition of various body parts

A dog must have 1 tail,

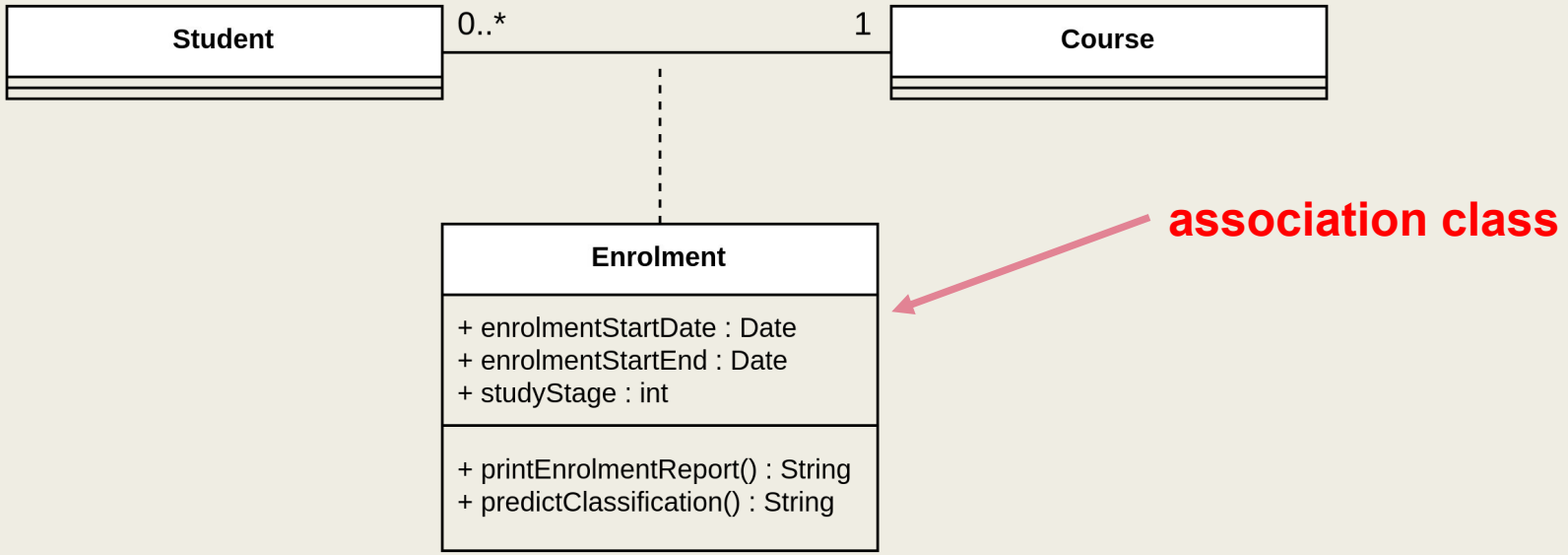
1 head and 4 legs



ASSOCIATION CLASSES

Association classes

- In UML diagrams, an **association class** is a class that is part of an association relationship between two other classes

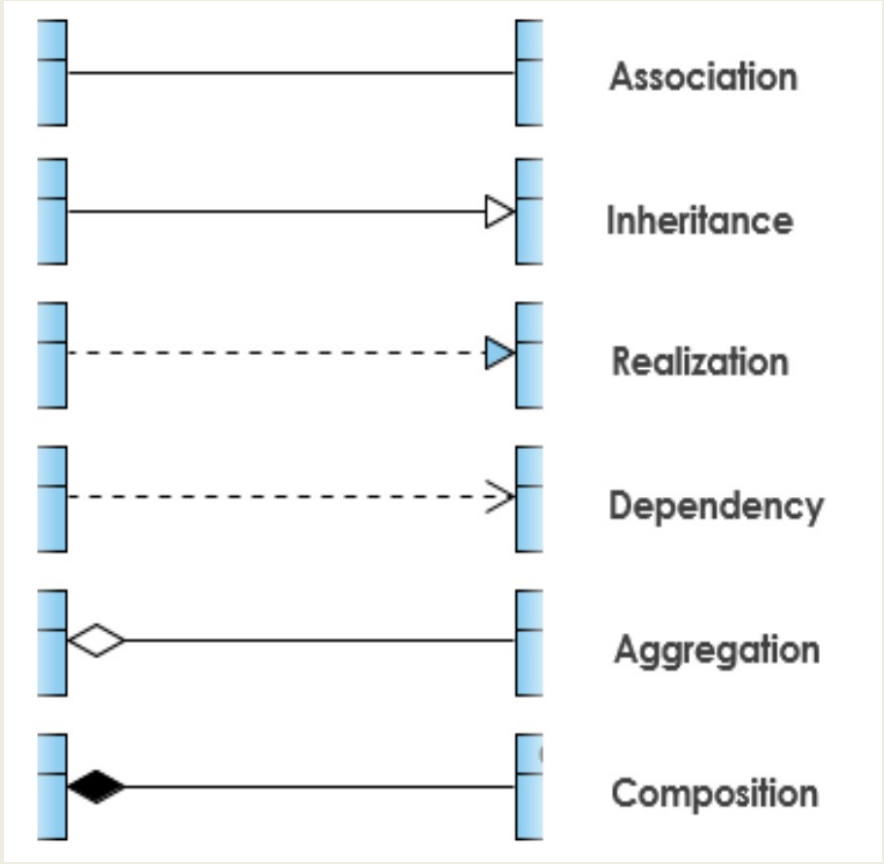


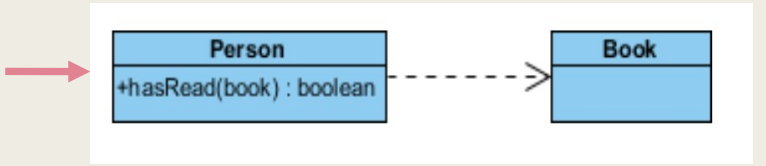
- Additional information can be provided if attach an association class to the association relationship, by a dotted line (**at most one instance** of an association class for every pair of class instances linked by that association)

RELATIONSHIPS - SUMMARY

Typical Relationships between classes

- Ideally, UML precisely conveys designers' intention on how code should be implemented from diagrams. It is important to know those typical type of relationships

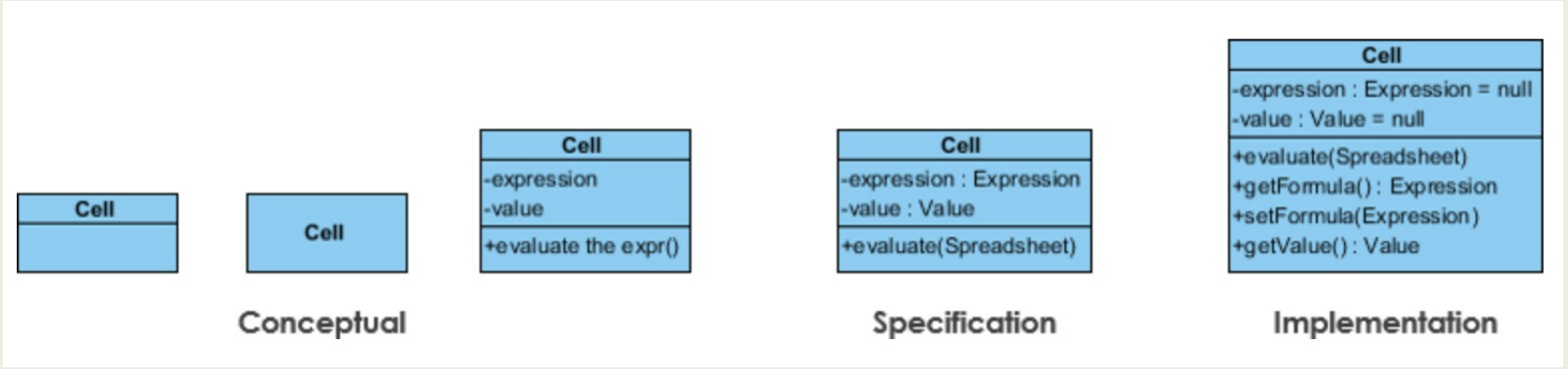




https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/

Perspectives of Class Diagram

- A diagram can be interpreted from various perspectives:
  - ***Conceptual****: represents the concepts in the domain*
  - ***Specification****: focus is on the interfaces of Abstract Data Type (ADTs) in the software*
  - ***Implementation****: describes how classes will implement their interfaces*

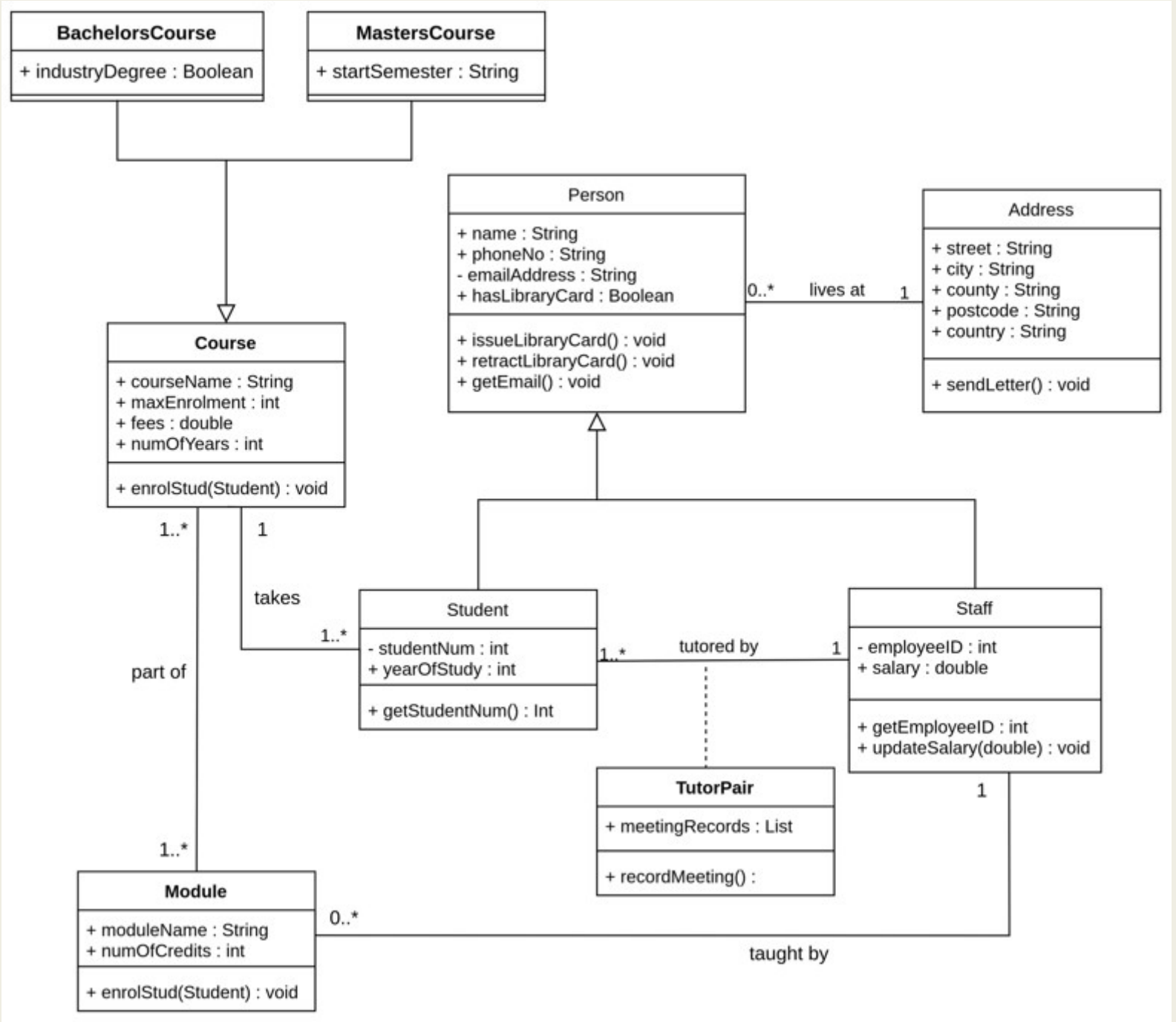


https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/

- The perspective affects the amount of detail to be supplied and the kinds of relationships worth presenting. In UML diagrams, the class name is the only mandatory information

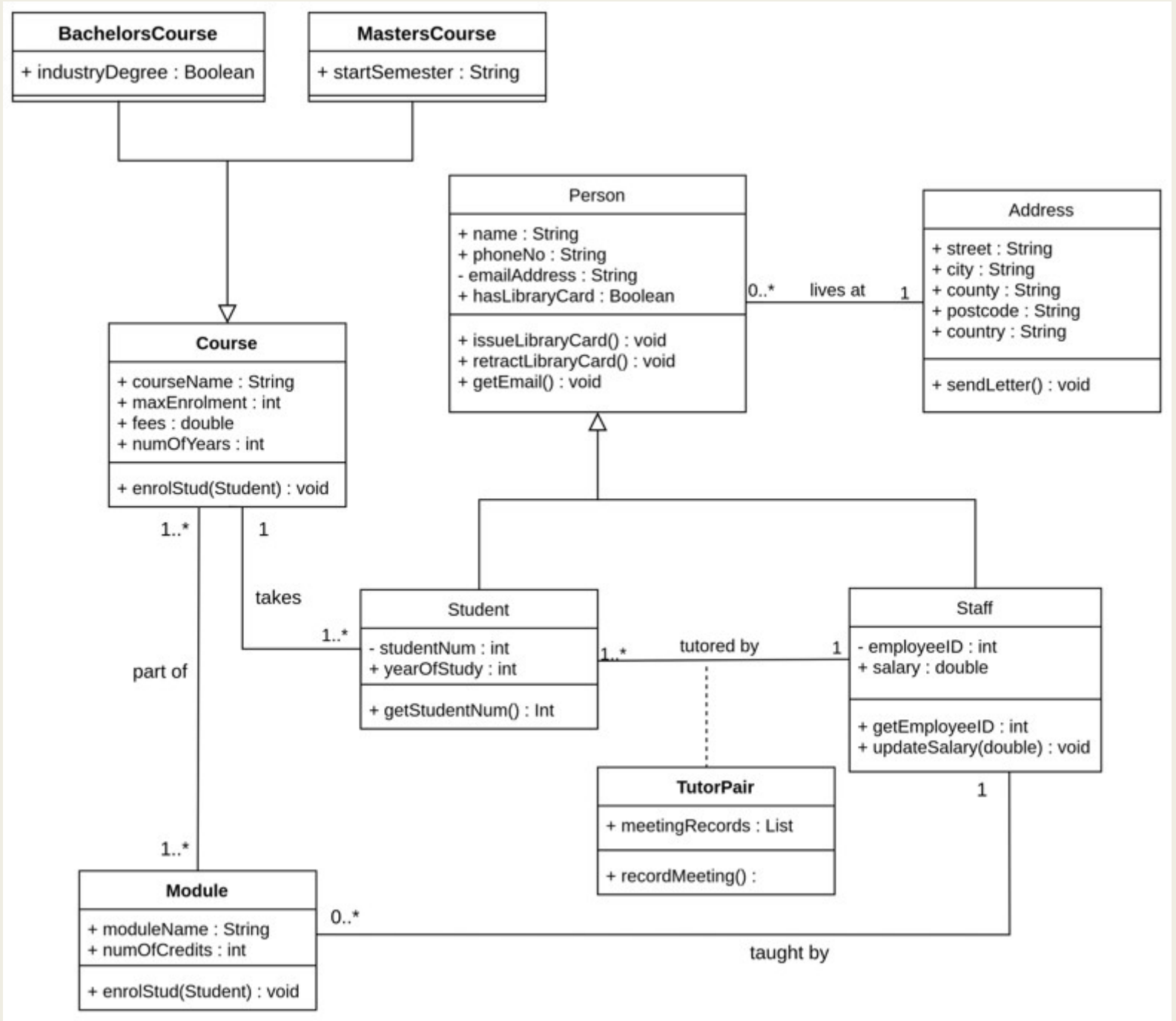
SIMPLE EXAMPLE

Simple example: University management system



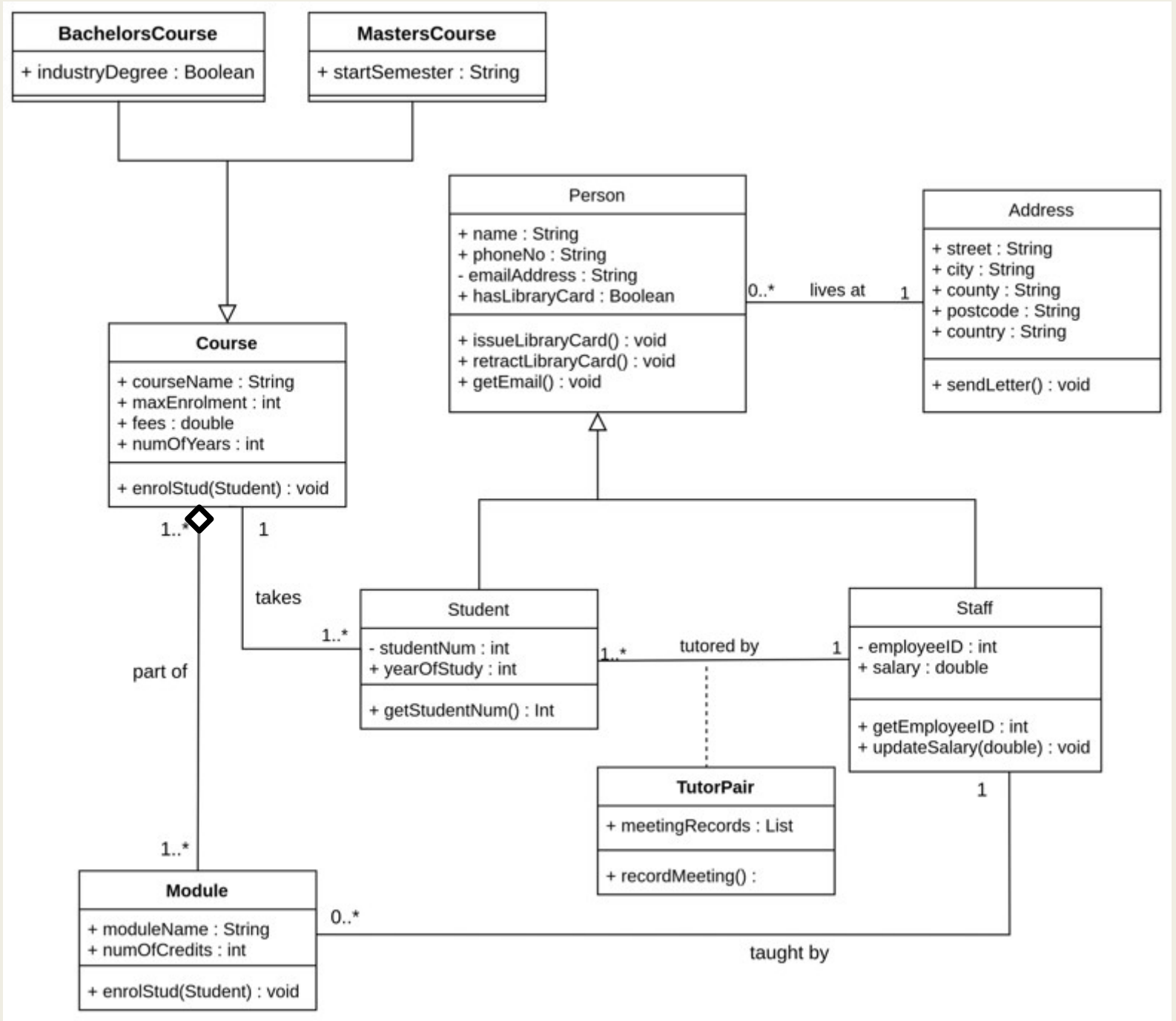
Simple example: University management system

- **Question 1:**
  How could we show show that each course ***must*** consist of a number of modules, but modules can still exist independently of any course?



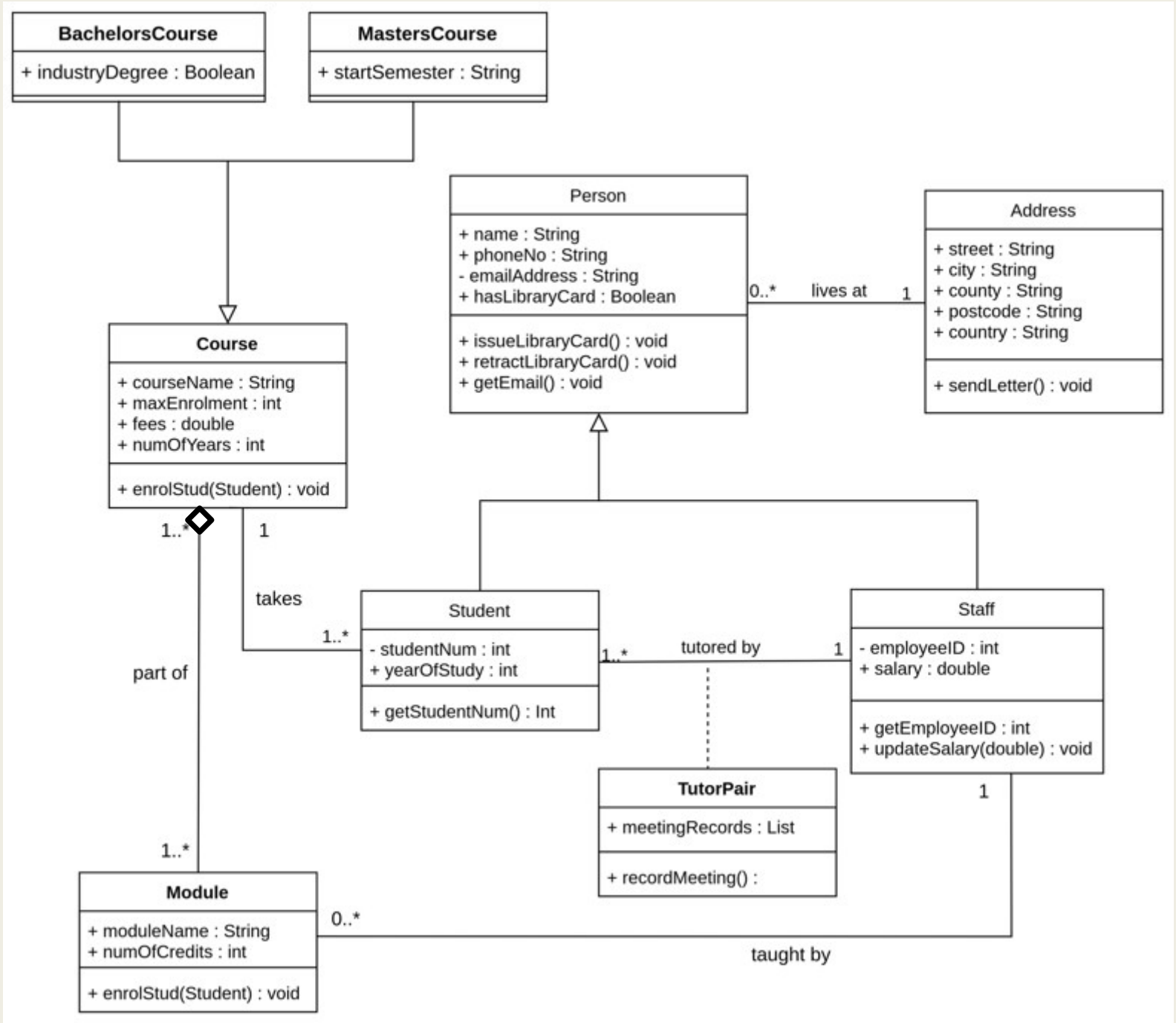
Simple example: University management system

- **Answer:** Aggregation association



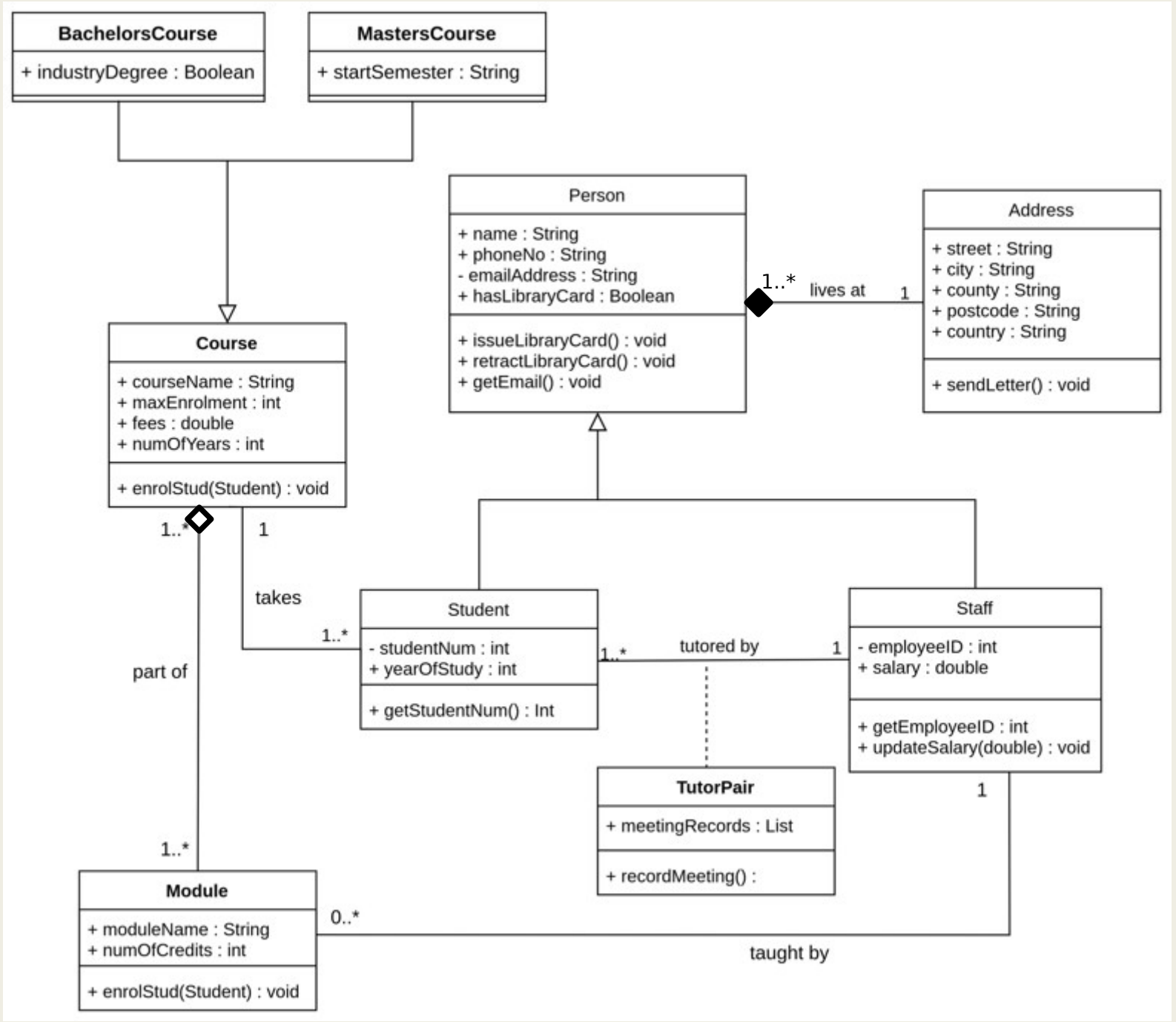
Simple example: University management system

- **Question 2:** How can we show that each Person needs to have an address, but that an address cannot exist in the system without at least one Person living there?



Simple example: University management system

- **Answer:** Composition association



What’s next?

- In this week’s tutorial you will:
  - *Create a UML Class Diagram for your project*
  - *Please don’t forget to update your WeeklyLog!*

- Lecture next week:
  Class Modelling
  - *Learn more about the class modelling process;*
  - *Learn how to analyse textual descriptions and use case diagrams to arrive at an appropriate class model;*
  - *Consider some further examples of Class Diagrams*

Task for the Surgery this week

- Create a UML Class Diagram
- **The diagram should contain:**
  - *A set of appropriately named classes with sensible attributes and methods*
  - *Appropriate associations between classes*
- **It should also:**
  - *Satisfy each of the functional requirements of your system*
  - *Enable each of the use cases in your Use Case diagram to be carried out by your system's end users*

Questions?

- Dr. Matthias Heintz or Prof. Shigang Yue
  - *mmh21@leicester.ac.uk or sy237@Leicester.ac.uk*
  - *Microsoft Teams*
  - *Office 613 or 608 in Ken Edwards Building*

UNIVERSITY OF LEICESTER