Based on:
- Seidl et al.'s "UML@Classroom", Chapter 4
- Miles and Hamilton's "Learning UML 2.0", Chapters 4 & 5

UML Class Diagrams

Software Design (40007) – 2024/2025

Justus Bogner
Ivano Malavolta

VRIJE UNIVERSITEIT AMSTERDAM

Software and Sustainability research group (S2)
Department of Computer Science, Faculty of Sciences

Roadmap

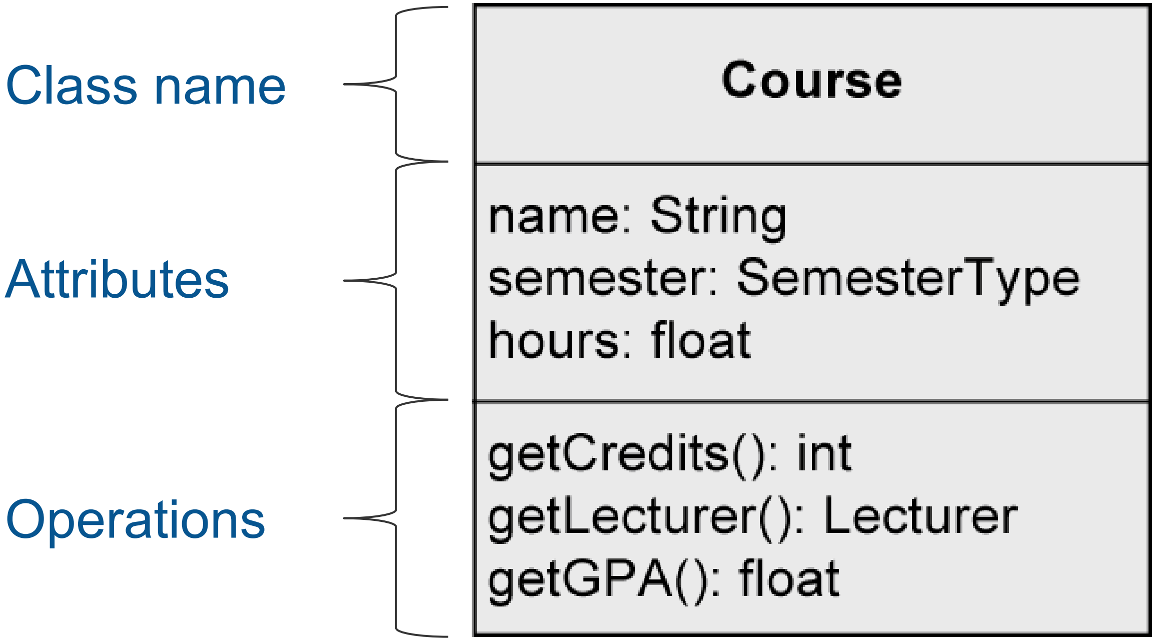
- Basics of class diagrams
- Relationships between classes
- Creating class diagrams (concrete example)

} Focused on UML syntax

Basics of Class Diagrams

Class

A class is a **construction plan** for a set of similar objects.



Class name

Attributes

Operations

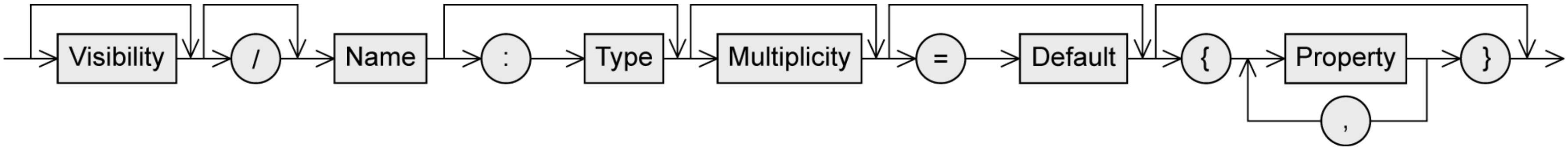
Course

name: String
semester: SemesterType
hours: float

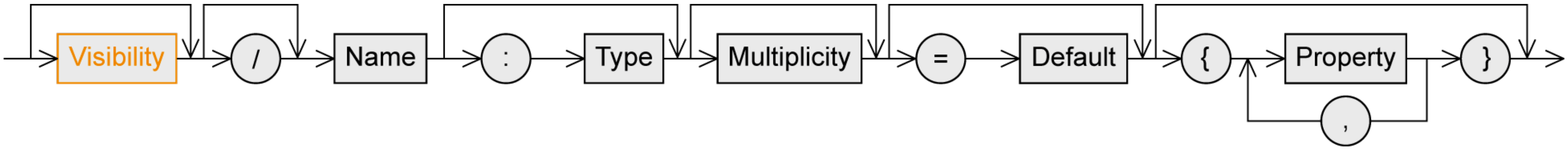
getCredits(): int
getLecturer(): Lecturer
getGPA(): float

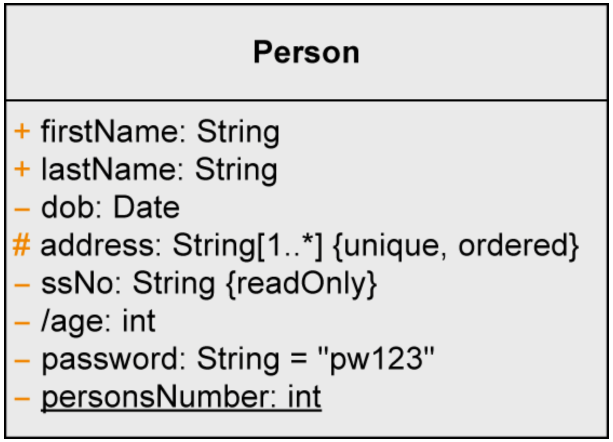
Note: in UML, an **`operation`** is the specification, while a **`method`** is the actual implementation.

Attribute syntax



Attribute syntax - visibility





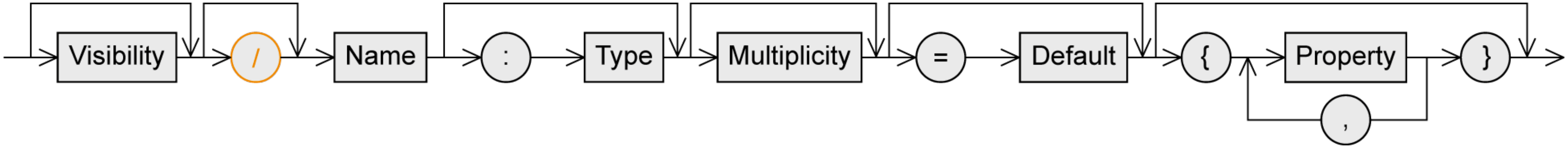
Person

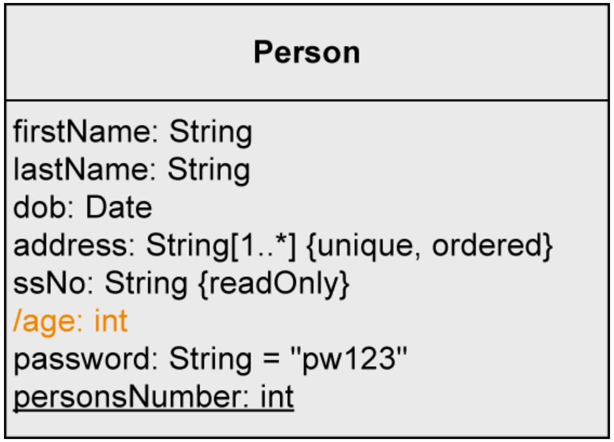
+ firstName: String
+ lastName: String
– dob: Date
# address: String[1..*] {unique, ordered}
– ssNo: String {readOnly}
– /age: int
– password: String = "pw123"
– personsNumber: int

Who is permitted to access the attribute

- + ... **public**: every other class
- - ... **private**: only the class itself
- # ... **protected**: class itself and subclasses
- ~ ... **package**: classes in the same package

Attribute syntax - derived attribute





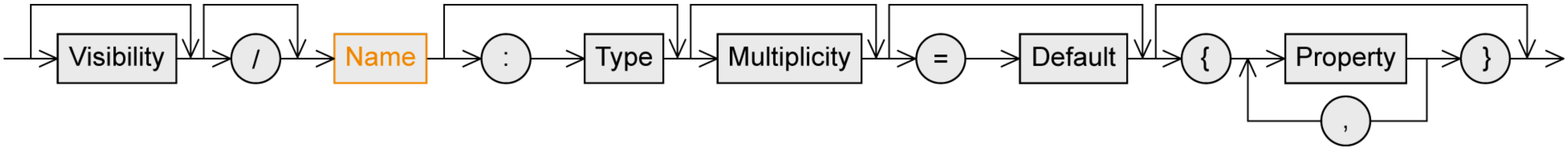
Person

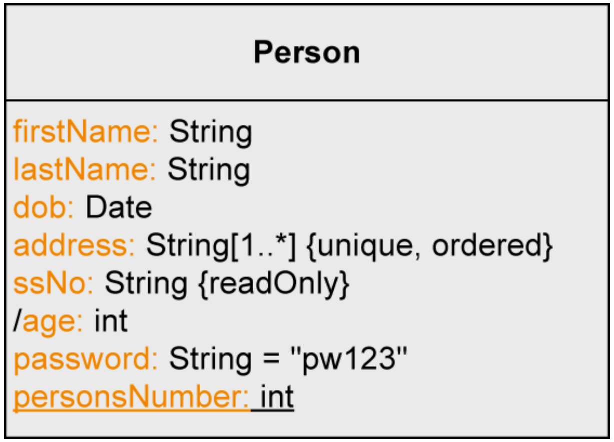
firstName: String
lastName: String
dob: Date
address: String[1..*] {unique, ordered}
ssNo: String {readOnly}
/age: int
password: String = "pw123"
personsNumber: int

Value is derived from other attributes

- **age**: calculated from the date of birth

Attribute syntax - name

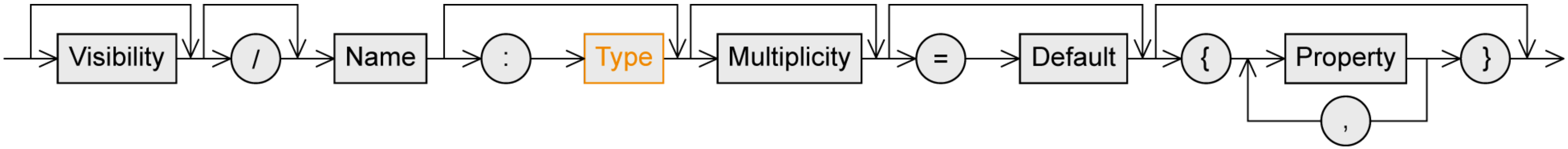


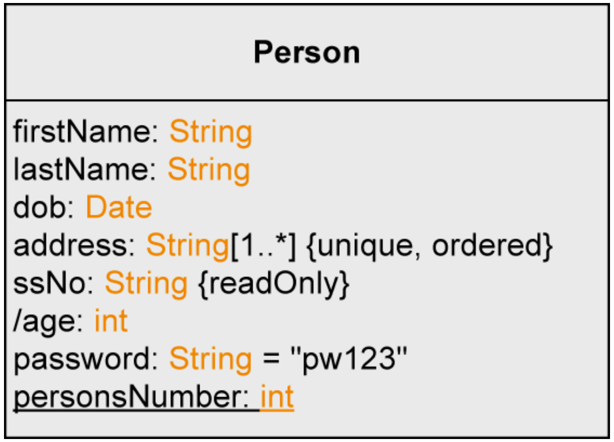

Person

firstName: String
lastName: String
dob: Date
address: String[1..*] {unique, ordered}
ssNo: String {readOnly}
/age: int
password: String = "pw123"
personsNumber: int

Name of the attribute

Attribute syntax - type

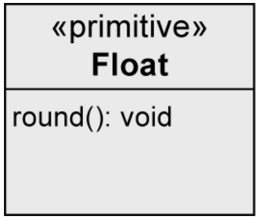



Person

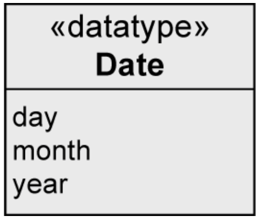
firstName: String
lastName: String
dob: Date
address: String[1..*] {unique, ordered}
ssNo: String {readOnly}
/age: int
password: String = "pw123"
personsNumber: int

Type

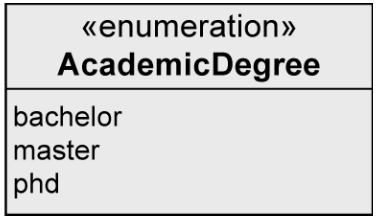
- Primitive data type
  - Pre-defined:
    - Boolean, Integer, String, UnlimitedNatural
  - User-defined: «**`primitive`**»
  - Composite data type: «**`datatype`**»
    - Similar to class but:
      - Instances are value objects
      - Identity defined by its properties
      - Immutable
- Enumerations: «**`enumeration`**»


«primitive»
Float

round(): void

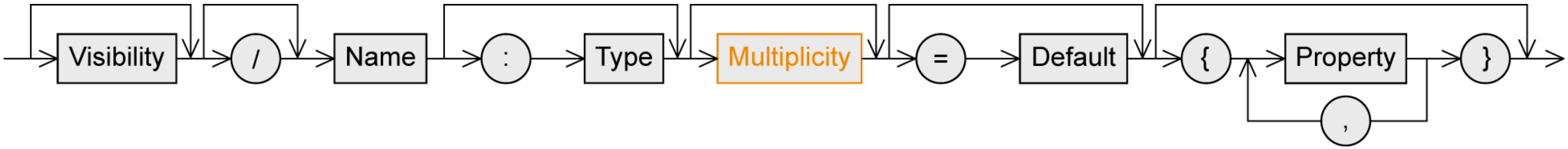

«datatype»
Date

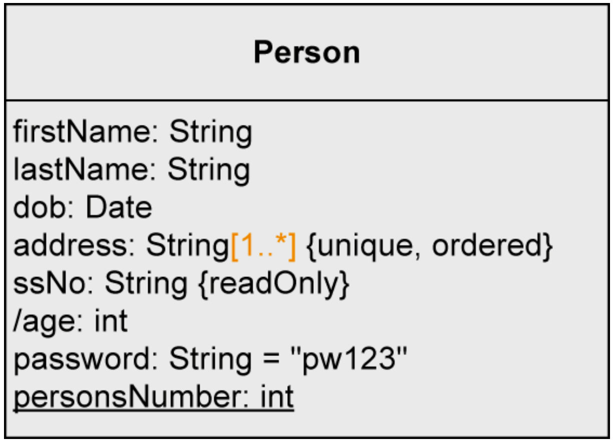
day
month
year


«enumeration»
AcademicDegree

bachelor
master
phd

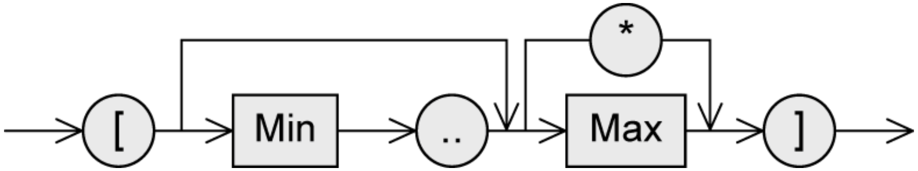
Attribute syntax - multiplicity



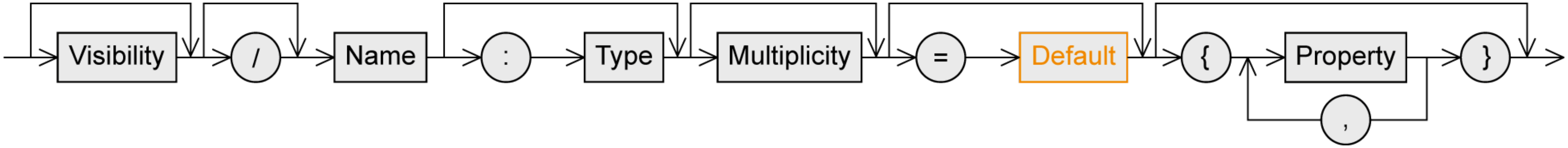

Person

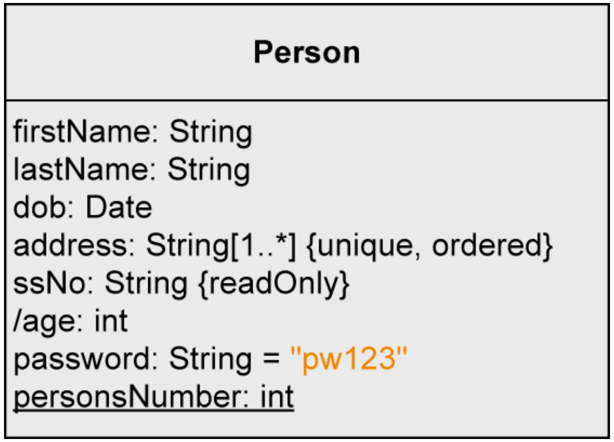
firstName: String
lastName: String
dob: Date
address: String[1..*] {unique, ordered}
ssNo: String {readOnly}
/age: int
password: String = "pw123"
personsNumber: int

- Number of values an attribute may have
- Default value: 1
- Notation: **[min..max]**
  - no upper limit: **[*]** or **[0..*]**



Attribute syntax - default value





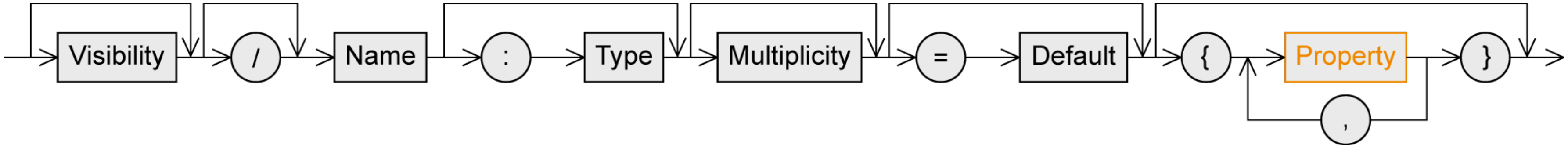
Person

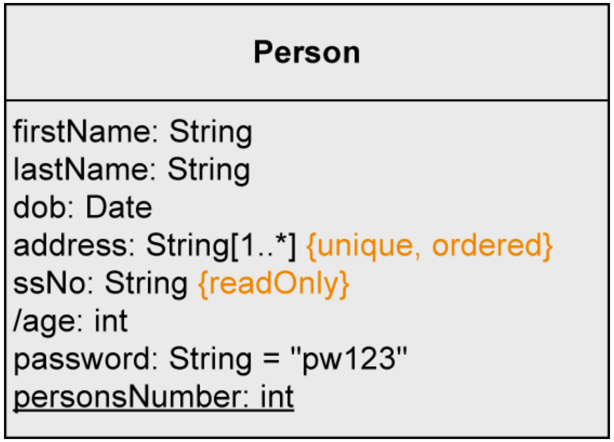
firstName: String
lastName: String
dob: Date
address: String[1..*] {unique, ordered}
ssNo: String {readOnly}
/age: int
password: String = "pw123"
personsNumber: int

Default value

- Used if the attribute value is not set explicitly by the user

Attribute syntax - properties





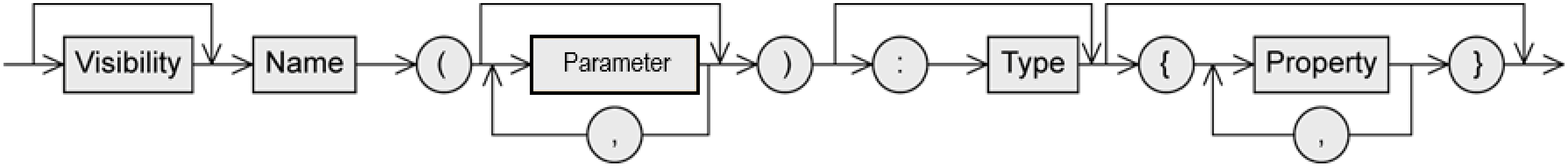
Person

firstName: String
lastName: String
dob: Date
address: String[1..*] {unique, ordered}
ssNo: String {readOnly}
/age: int
password: String = "pw123"
personsNumber: int

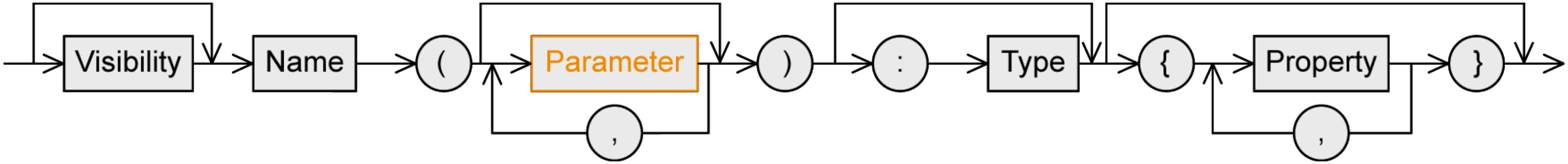
- Pre-defined properties
  - {**readOnly**} … value cannot be changed
  - {**unique**} … no duplicates permitted
  - {**non-unique**} … duplicates permitted
  - {**ordered**} … fixed order of the values
  - {**unordered**} … no fixed order of the values

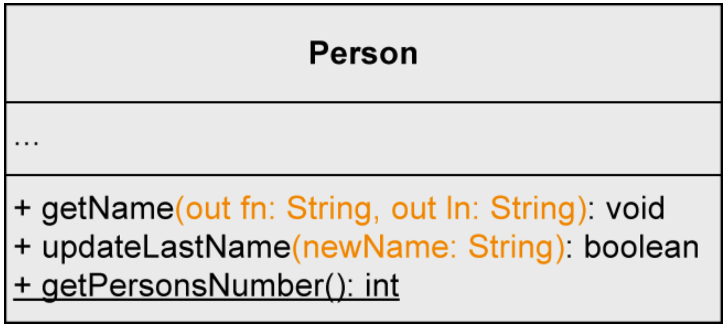
- Examples:
  - Set: {unordered, unique}
  - List in ascending order: {ordered, non-unique}

Operation syntax



Operation syntax - parameters





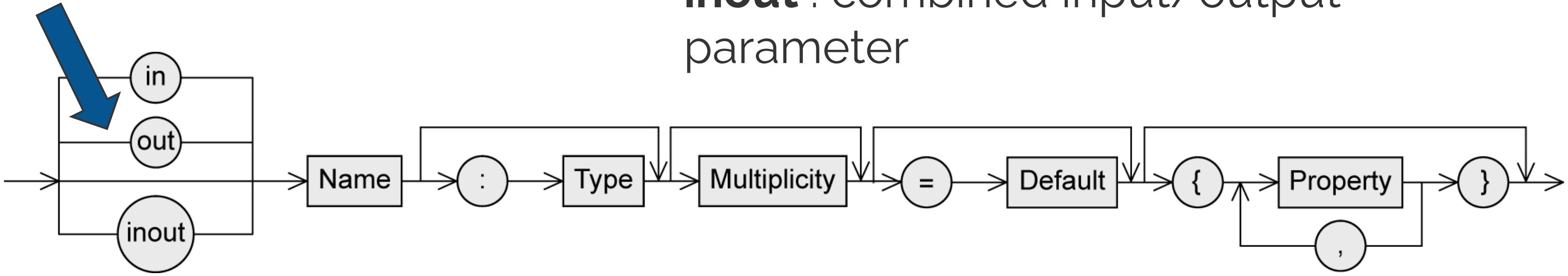
Person

...

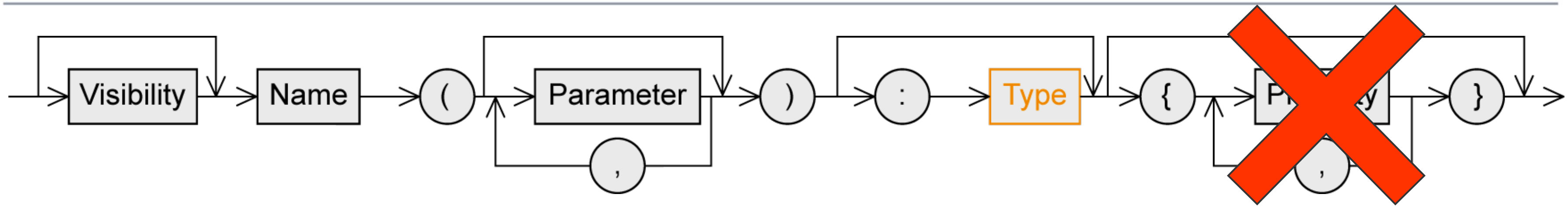
+ getName(out fn: String, out ln: String): void
+ updateLastName(newName: String): boolean
+ getPersonsNumber(): int

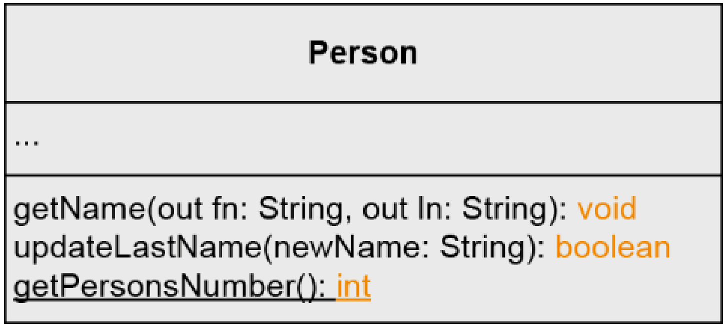
- Notation similar to attributes
- Direction of the parameter
  - **in** … input parameter
    - When the operation is used, a value is expected from this parameter
  - **out** … output parameter
    - After the execution of the operation, the parameter has adopted a new value
  - **inout** : combined input/output parameter

Only situationally useful; suggestion: ignore them



Operation syntax - type




Person

...

getName(out fn: String, out ln: String): void
updateLastName(newName: String): boolean
getPersonsNumber(): int

- Type of the return value

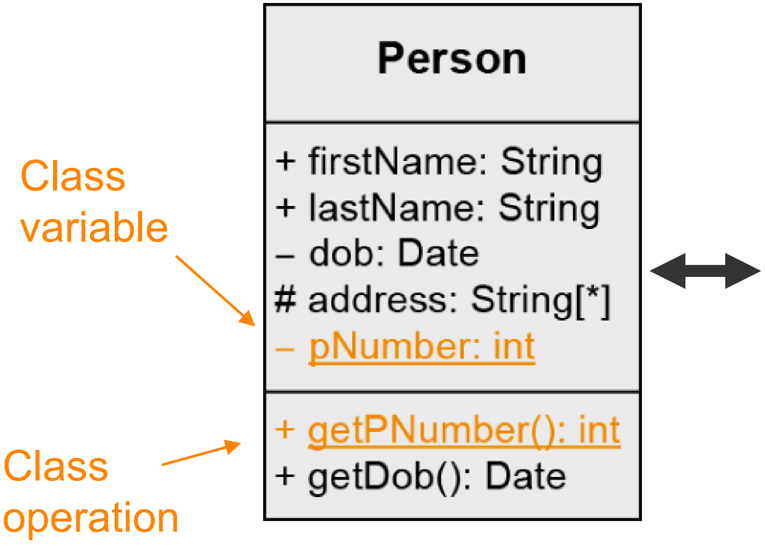
Properties for operations exist, but they are rarely used (we don't cover them). You can check them out here: https://www.uml-diagrams.org/operation.html

Class variables and class operations

- Default: instance variable (= instance attribute): attributes defined on the instance level
- Class variable (= class attribute, static attribute)
  - Defined only once per class,
    - i.e., shared by all instances of the class
    - e.g., counters for the number of instances of a class, constants, etc.
- Class operation (= static operation)
  - Can be used if no instance of the corresponding class was created
    - e.g., constructors, counting operations, math functions, etc.
- Notation: <u>underlining</u> the name of class variable or class operation

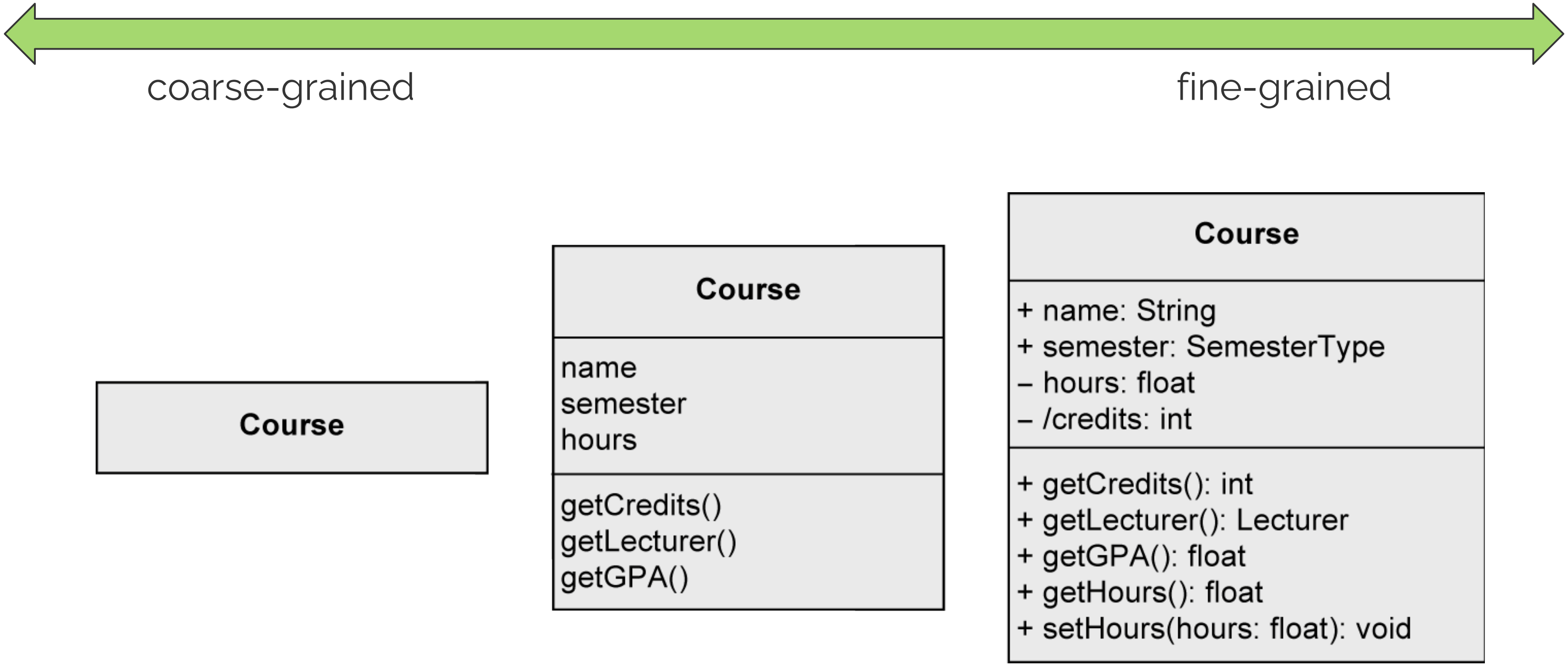
Class variables and class operations - example

How would you implement this in Java?



```
class Person {

    public String firstName;
    public String lastName;
    private Date dob;
    protected String[] address;
    private static int pNumber;
    public static int getPNumber() {…}
    public Date getDob() {…}
}
```

Specification of classes: different levels of detail



coarse-grained

fine-grained

Course

Course

name
semester
hours

getCredits()
getLecturer()
getGPA()

Course

+ name: String
+ semester: SemesterType
– hours: float
– /credits: int

+ getCredits(): int
+ getLecturer(): Lecturer
+ getGPA(): float
+ getHours(): float
+ setHours(hours: float): void

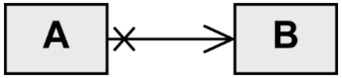
You can use a more coarse-grained class diagram as a descriptive model.

Relationships Between Classes

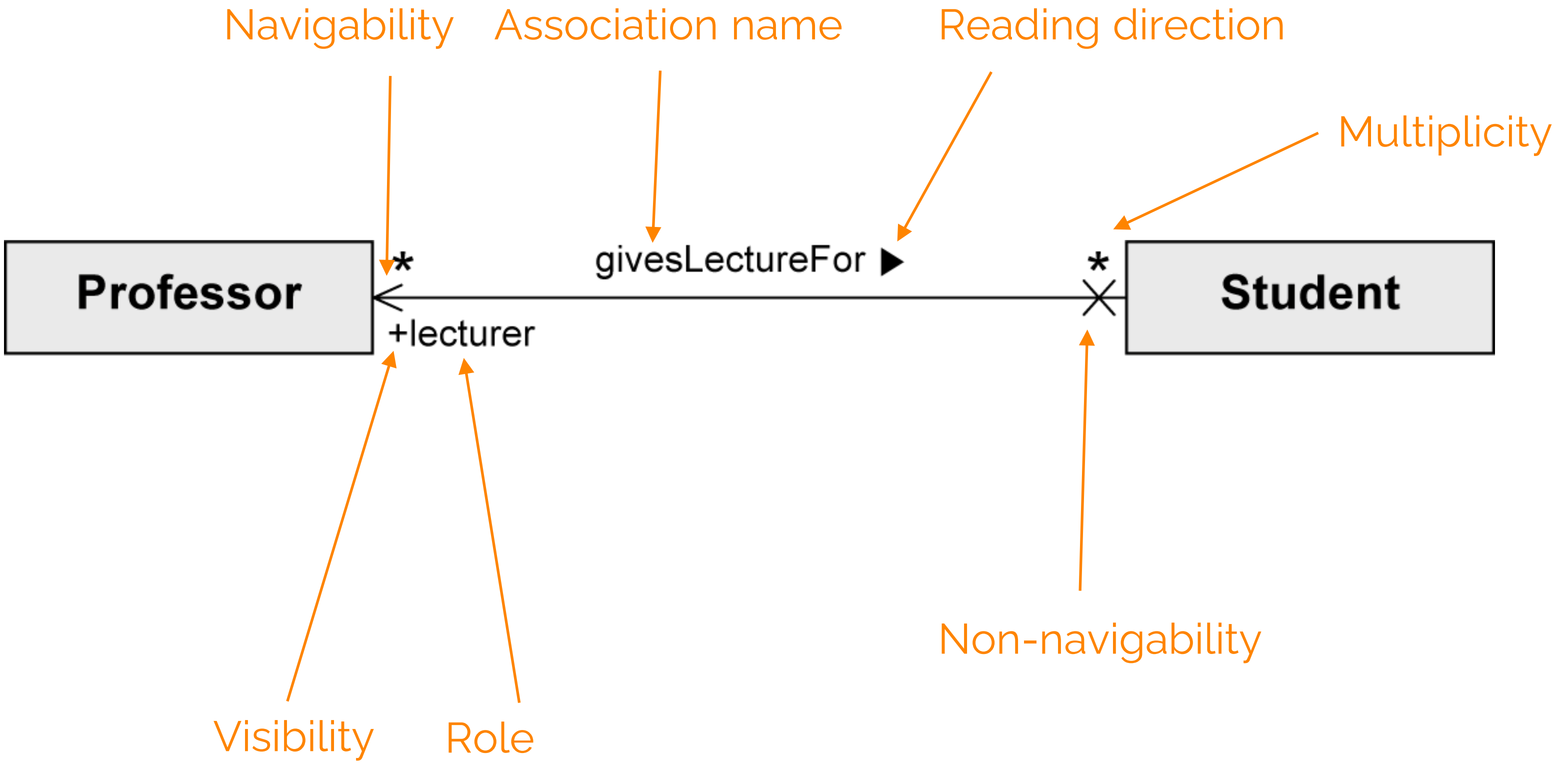
Roadmap

- Associations
  - Binary
  - N-ary
  - Association Class
- Aggregation / Composition
- Generalization
  - Inheritance
  - Abstract classes

Binary association



Connects instances of two classes with one another



Binary association - navigability

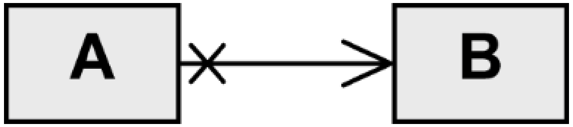
Navigability: an object knows its partner objects and can therefore access their visible attributes and operations

- Indicated by open arrow head

Non-navigability

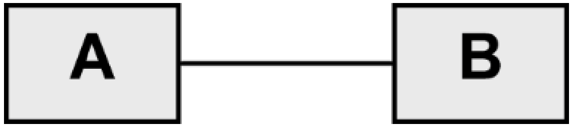
- Indicated by a cross, but this is also often left out

Example:



- **A** can access the visible attributes and operations of **B**
- **B** cannot access any attributes and operations of **A**

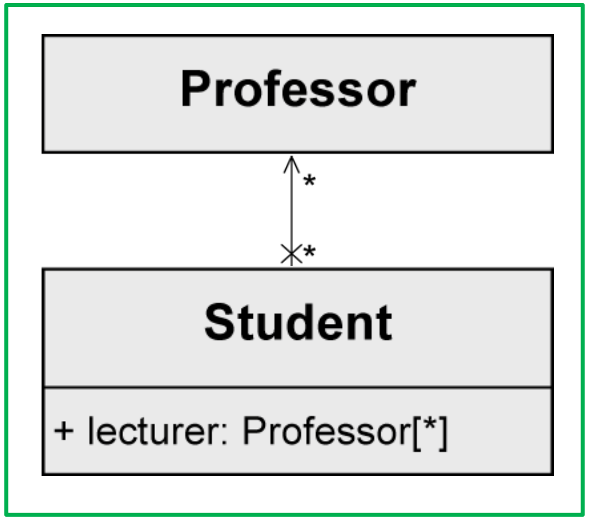
Navigability undefined

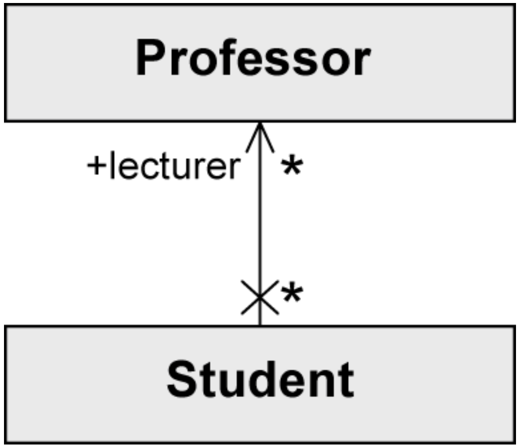


- Bidirectional navigability is assumed

Binary association representation

I prefer this one





In Java:

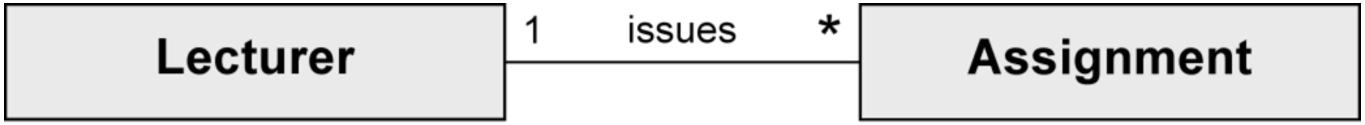
```
class Professor {…}

class Student {
   public Professor[] lecturer;
   …
}
```

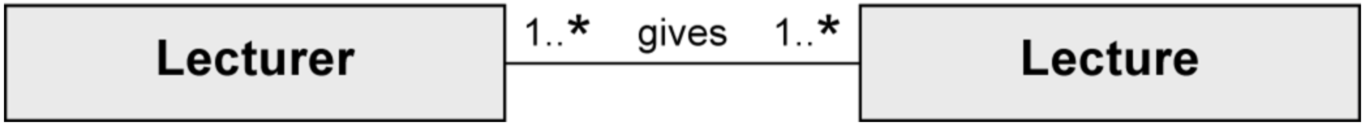
See anything here that is not ideal?

A collection should be reflected in the variable name, e.g., `lecturers` or `lecturerList`.

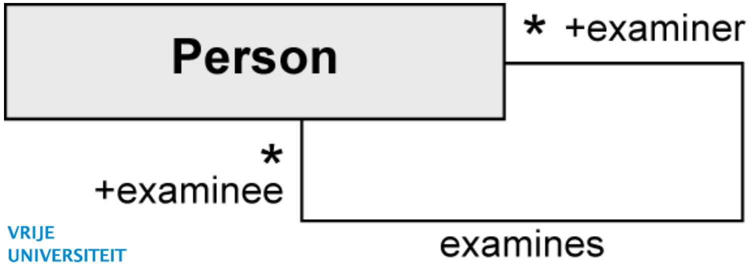
Binary association – multiplicity and role

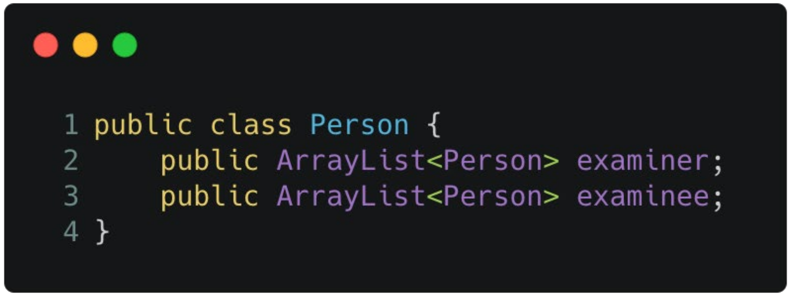
Multiplicity: Number of objects that may be associated with exactly one object of the opposite side





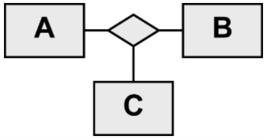
Role: describes the way in which an object is involved in an association relationship



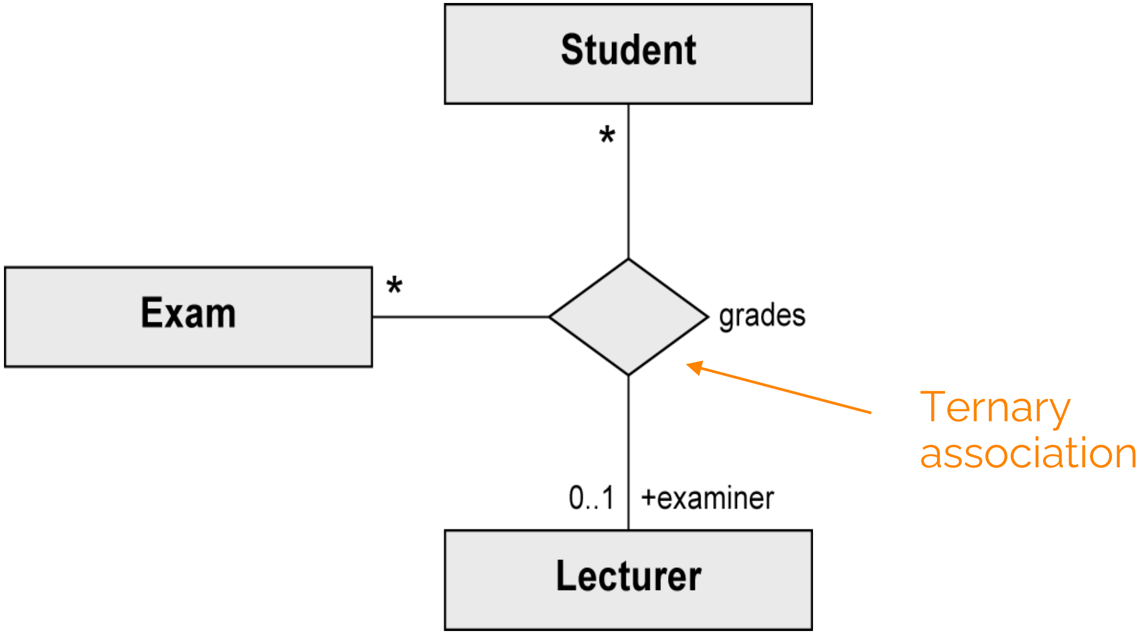


```
public class Person {
    public ArrayList<Person> examiner;
    public ArrayList<Person> examinee;
}
```

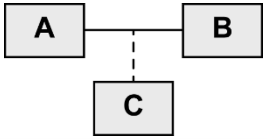
n-ary association



- More than two classes are involved in the relationship
- No navigation directions
- Interpretation can be complex
- Situationally useful, but mostly avoid

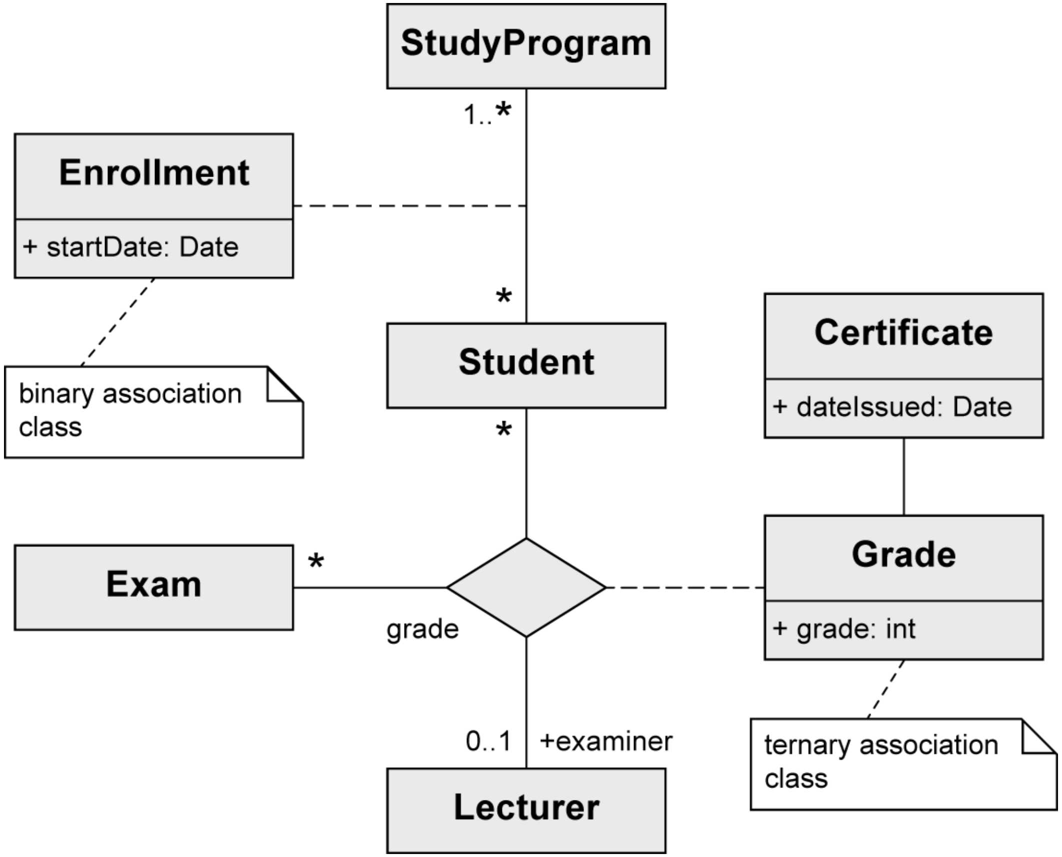


Ternary association



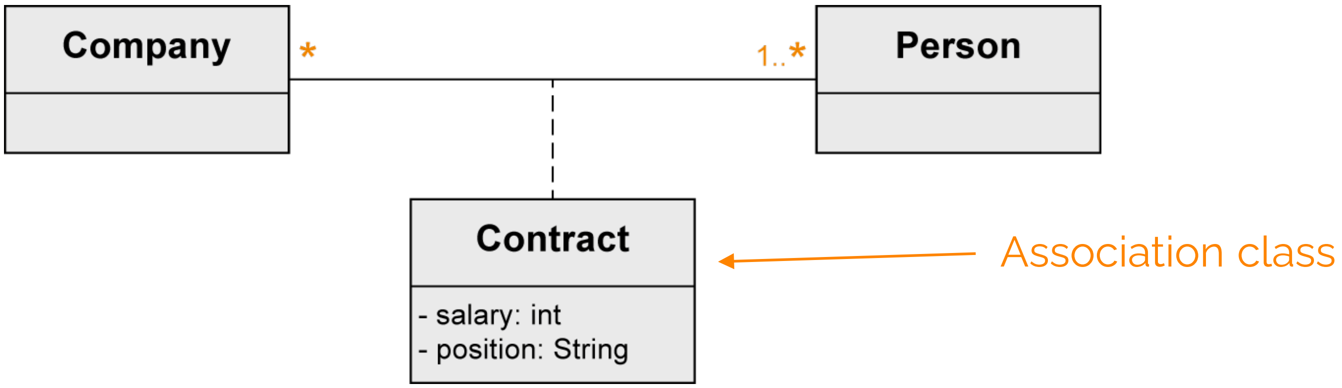
Association class

Assign attributes to the relationship between classes rather than to a class itself

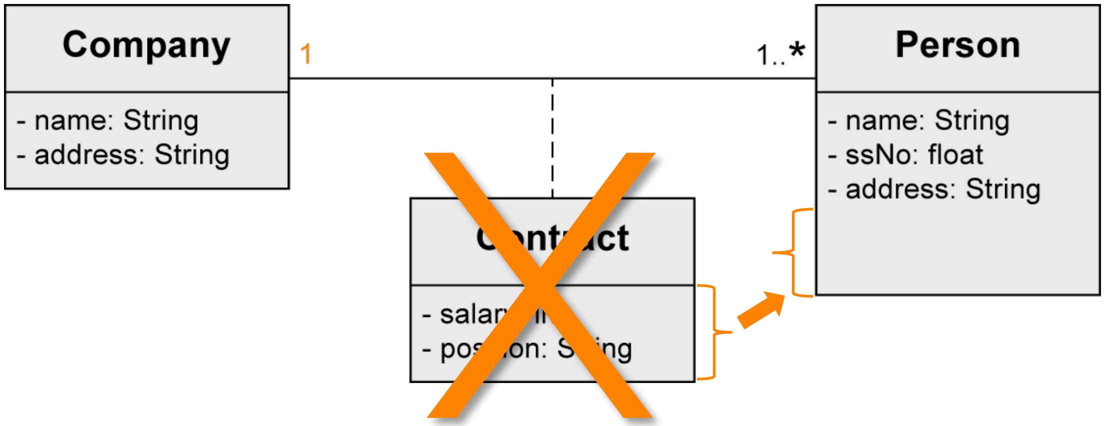


When to use an association class

- Mandatory when modeling n:m associations



- With 1:1 or 1:n → possible but not mandatory



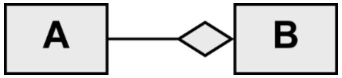
Aggregation

- Special form of association
- Used to express that a class **is part of** another class

- Properties of the aggregation association:
  - **Transitive**: if **B** is part of **A** and **C** is part of **B**, **C** is also part of **A**
  - **Asymmetric**: it is not possible for **A** to be part of **B** and **B** to be part of **A** simultaneously

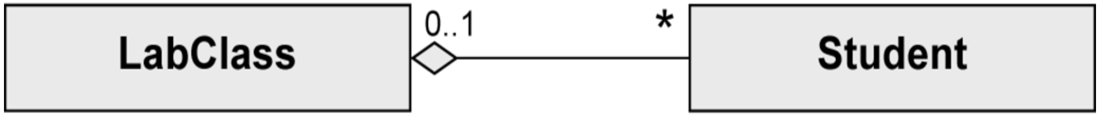
- Two types:
  - (Shared) aggregation
  - Composition

(Shared) Aggregation

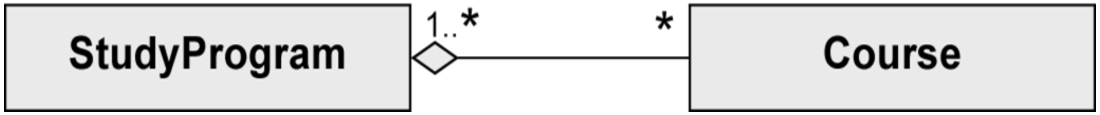


- Expresses a weak belonging of the parts to a whole
  - Parts also exist independently of the whole
- Multiplicity at the aggregating end may be >1
  - One element can be part of multiple other elements simultaneously

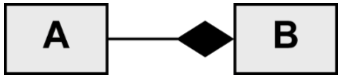
- Example:
  - **Student** is part of **LabClass**



  - **Course** is part of **StudyProgram**



Composition

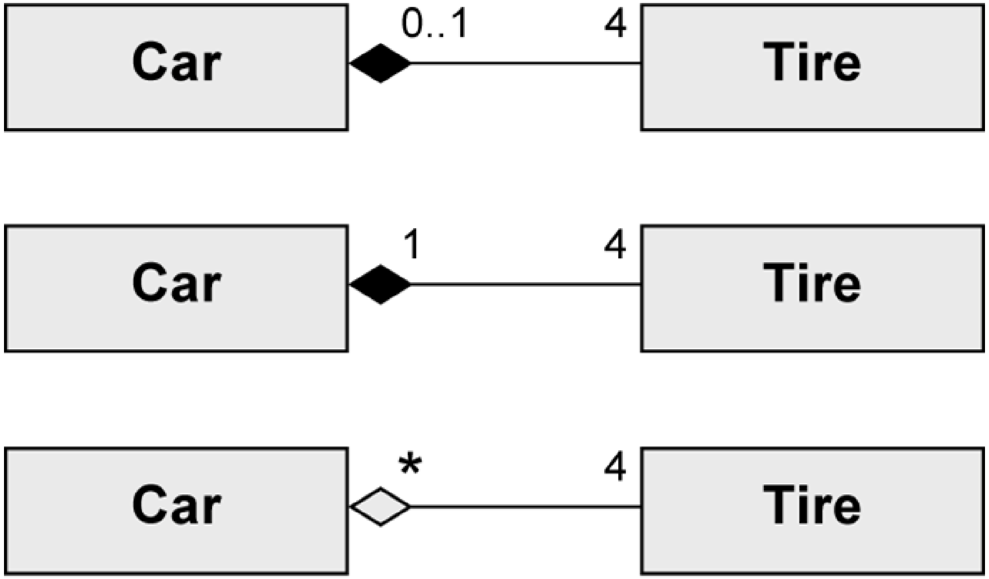


- **Existence dependency** between the composite object and its parts
- An existing part is contained in at most one composite object at one specific point in time
  - Multiplicity at the aggregating end max 1

- If the composite object is deleted, its parts are also deleted → the part **usually** cannot exist without the whole (exceptions are possible with multiplicity `0..1` instead of `1`)

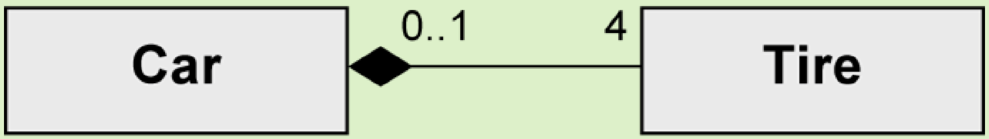
Aggregation vs. composition

Which one is the most reasonable representation of reality?



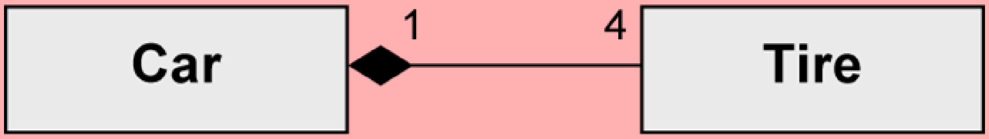
Aggregation vs. composition

Which one is the most reasonable representation of reality?

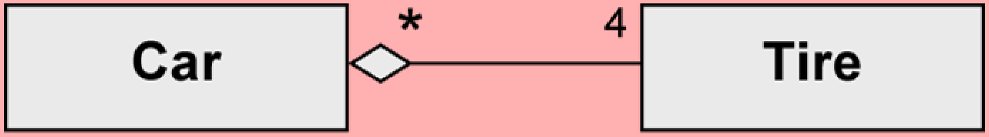


A Tire can exist without a Car. A Tire belongs to one Car at most

Yes



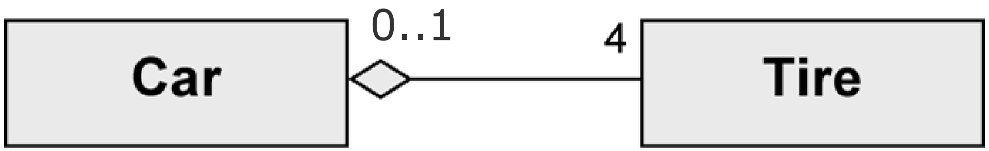
A Tire cannot exist without a Car



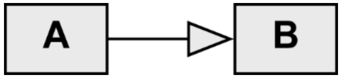
A Tire can belong to multiple Cars

No

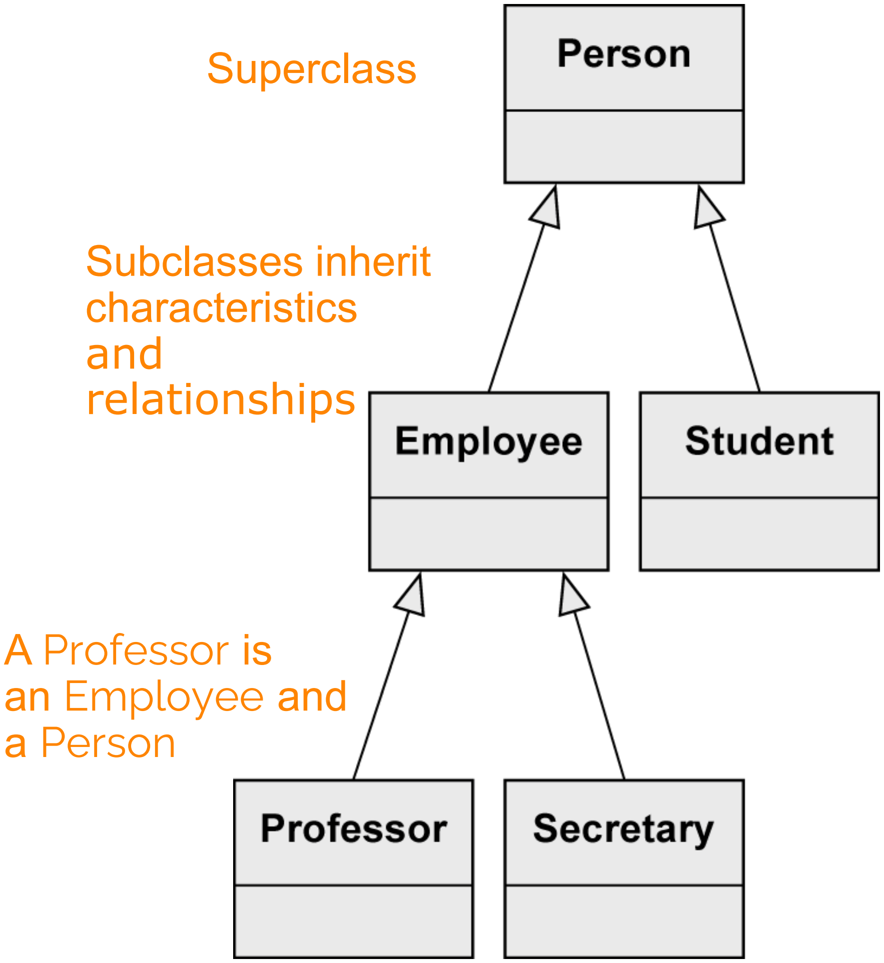
Another valid option:



Generalization - inheritance



- Attributes, operations, and relationships of the general class **are passed on to its subclasses** (except private ones)
- Every instance of a subclass is also an indirect instance of the superclass
- Subclasses may have further characteristics and relationships
- Generalizations are transitive

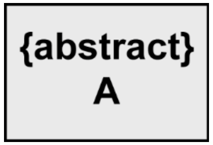


Superclass

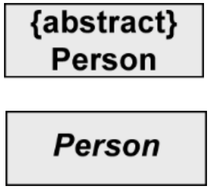
Subclasses inherit characteristics and relationships

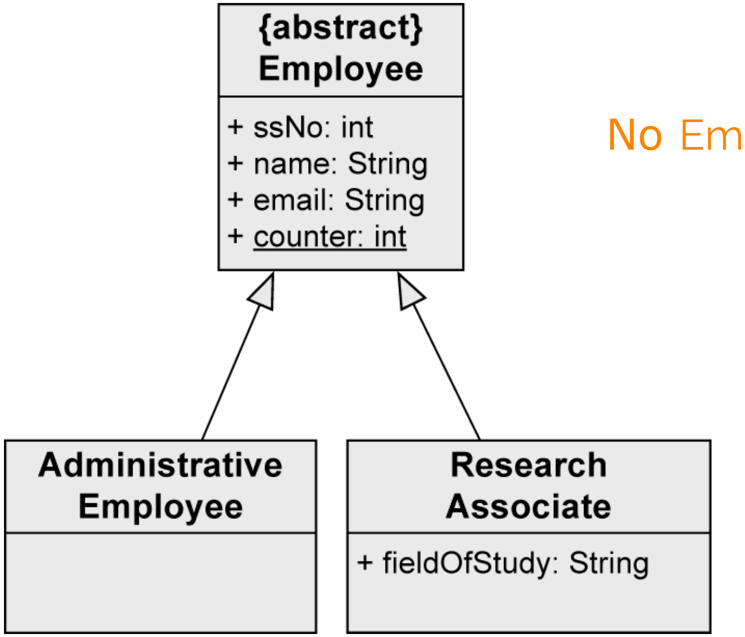
A Professor is an Employee and a Person

Generalization – abstract classes



- Used to highlight common characteristics of their subclasses while ensuring **no direct instances of the superclass**
- Only its non-abstract subclasses can be instantiated
- Useful in the context of generalization relationships

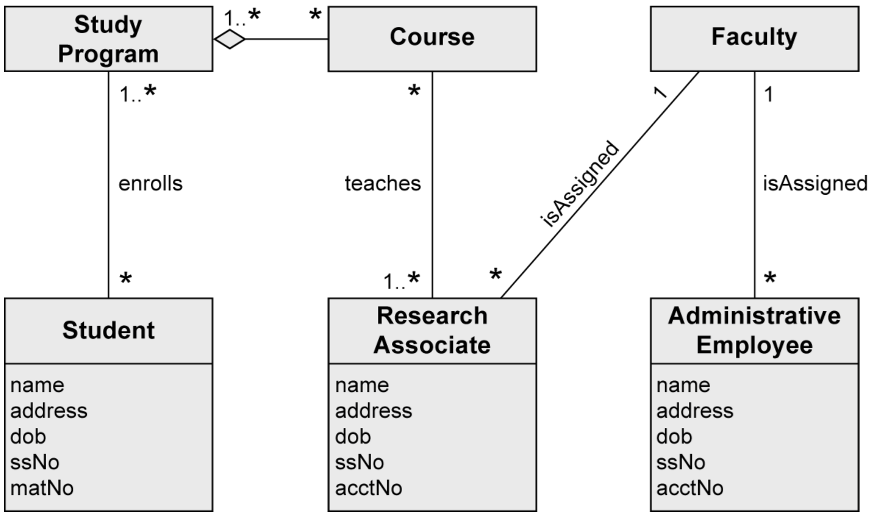


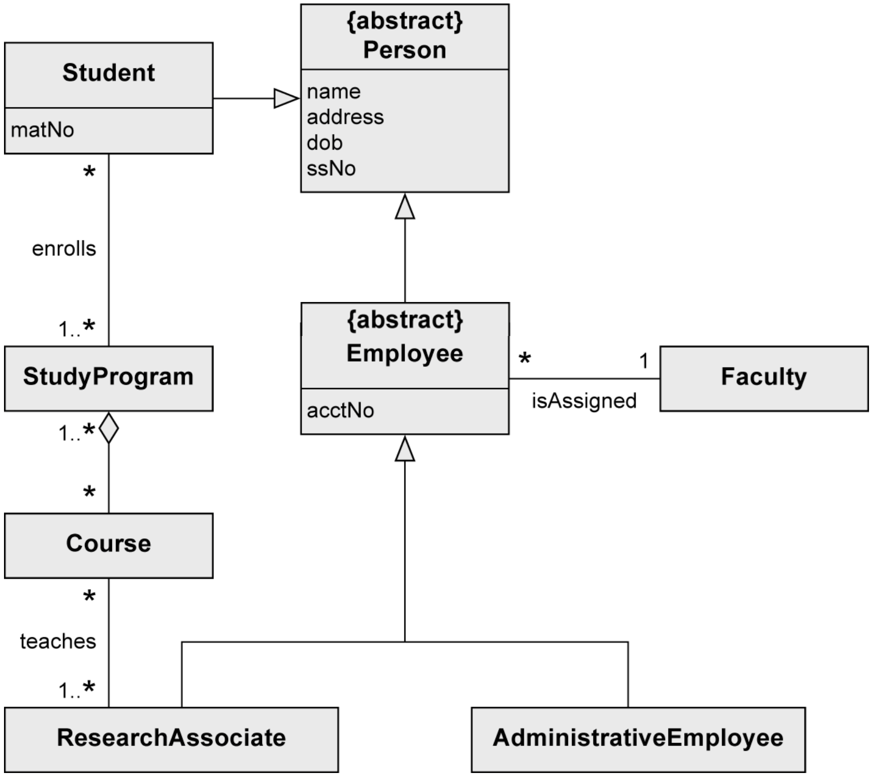


No Employee **object possible**

Two types of Employees: Administrative Employee **and** Research Associate

Example: generalization avoids duplication





Do you see something “incorrect” here?

Relationship decision guidance

1. Is one class a **specialized version** of the other, e.g., person and employee?

   Yes: → **Generalization**

   No: continue

2. Is one class a **part** of the other (whole-part relationship), e.g., institute and faculty or CPU and computer?

   No: → **Association**

   Yes: continue

3. Is the whole-part relationship **very strong** (part cannot really exist without whole, destroying the whole also destroys parts, part can only belong to at most one whole)?

   Yes: → **Composition**

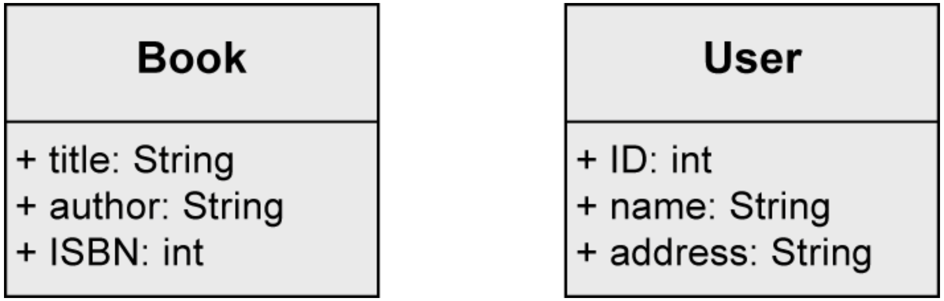
   No: → **Aggregation**

Creating Class Diagrams (Concrete Example)

Creating a class diagram

- Difficult to extract classes, attributes, and associations from natural language automatically
- Guidelines
  - **Nouns** often indicate classes, but can also be attributes (“have”)
  - **Adjectives** often indicate attribute values
  - **Verbs** often indicate operations or relationships

- Example: the library management system stores users with their unique ID, name, and address as well as books with their title, author, and ISBN number. Ann Foster wants to use the library.



Book

+ title: String
+ author: String
+ ISBN: int

User

+ ID: int
+ name: String
+ address: String

Question: What about “Ann Foster”?

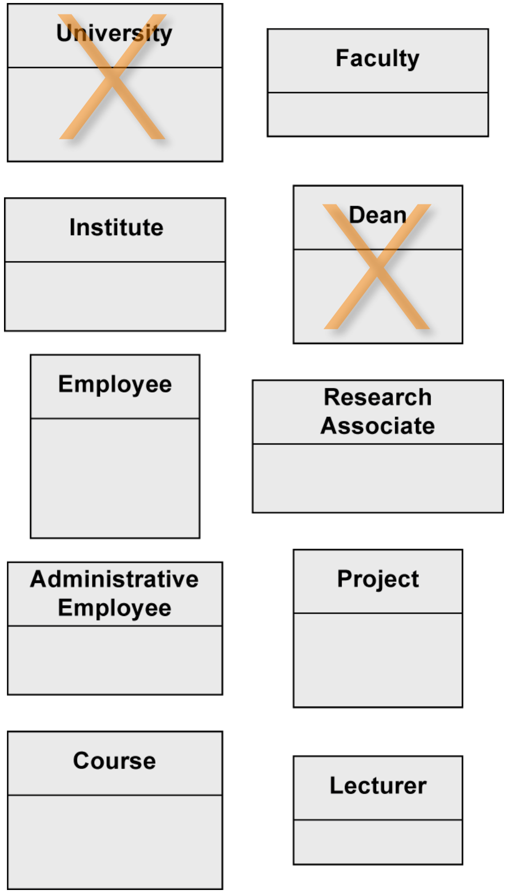
Example – University Information System

- A university consists of multiple faculties, which are composed of various institutes.
- Each faculty and each institute has a name.
- An address is known for each institute.
- Each faculty is led by a dean, who is an employee of the university.
- The total number of employees is known.
- Employees have a social security number, a name, and an email address. There is a distinction between research and administrative personnel.
- Research associates are assigned to at least one institute. The field of study of each research associate is known. Furthermore, research associates can be involved in projects for a certain number of hours, and the name, starting date, and end date of the projects are known.
- Some research associates hold courses. Then they are called lecturers.
- Courses have a unique number (ID), a name, and a weekly duration in hours.

Step 1: identifying classes

- A university consists of multiple faculties, which are composed of various institutes.
- Each faculty and each institute has a name.
- An address is known for each institute.
- Each faculty is led by a dean, who is an employee of the university.
- The total number of employees is known.
- Employees have a social security number, a name, and an email address. There is a distinction between research and administrative personnel.
- Research associates are assigned to at least one institute. The field of study of each research associate is known. Furthermore, research associates can be involved in projects for a certain number of hours, and the name, starting date, and end date of the projects are known.
- Some research associates hold courses. Then they are called lecturers.
- Courses have a unique number (ID), a name, and a weekly duration in hours.

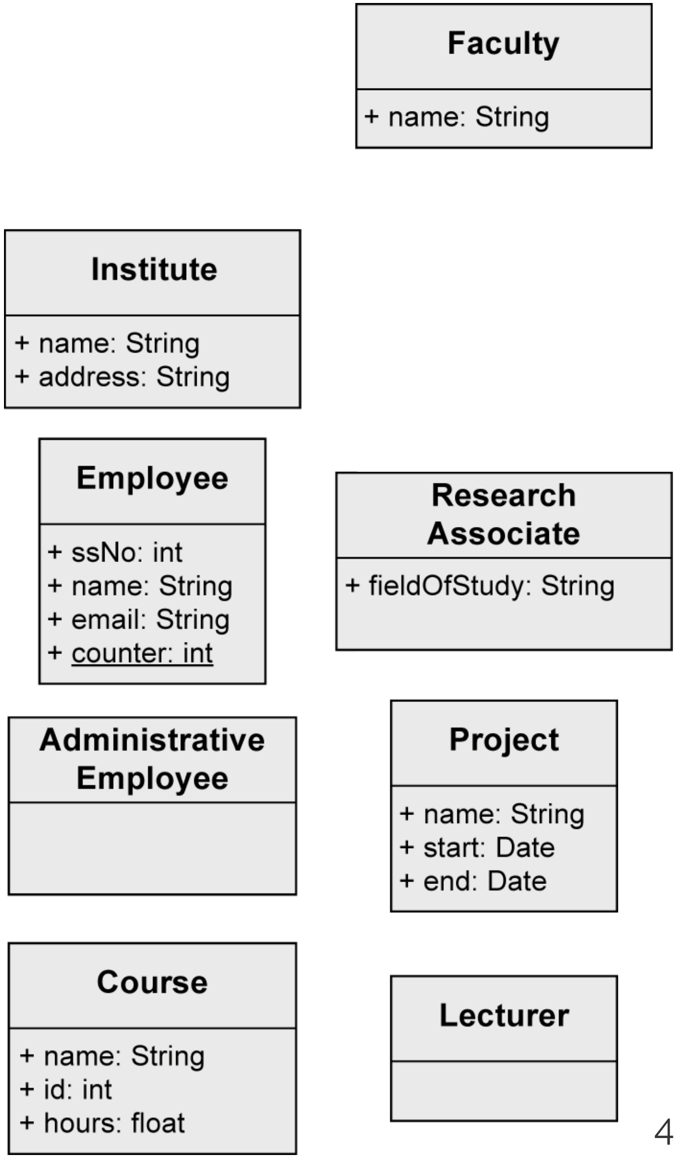
We model the system „University“



Dean has no further attributes than any other employee

Step 2: identifying attributes

- A university consists of multiple faculties which are composed of various institutes.
- Each faculty and each institute has a name.
- An address is known for each institute.
- Each faculty is led by a dean, who is an employee of the university.
- The total number of employees is known.
- Employees have a social security number, a name, and an email address. There is a distinction between research and administrative personnel.
- Research associates are assigned to at least one institute. The field of study of each research associate is known. Furthermore, research associates can be involved in projects for a certain number of hours, and the name, starting date, and end date of the projects are known.
- Some research associates hold courses. Then they are called lecturers.
- Courses have a unique number (ID), a name, and a weekly duration in hours.



Faculty
+ name: String

Institute
+ name: String
+ address: String

Employee
+ ssNo: int
+ name: String
+ email: String
+ counter: int

Research Associate
+ fieldOfStudy: String

Administrative Employee

Project
+ name: String
+ start: Date
+ end: Date

Course
+ name: String
+ id: int
+ hours: float

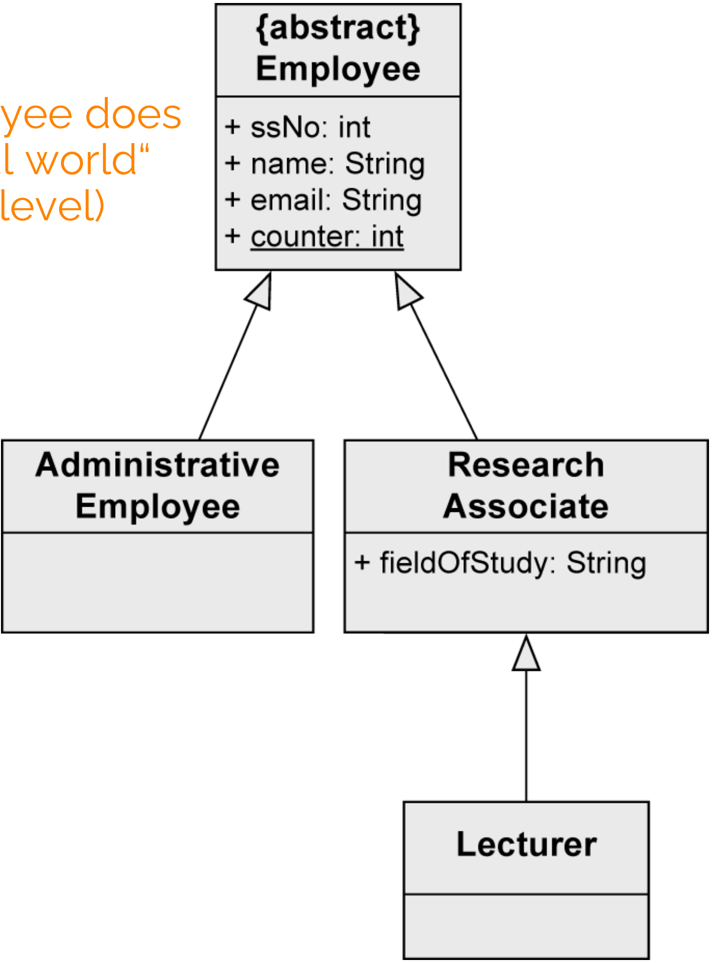
Lecturer

Step 3: identifying relationships (1/6)

Three kinds of relationships:

- Association
- Generalization
- Aggregation

Abstract, i.e., employee does not exist in the „real world“ (i.e., at the instance level)

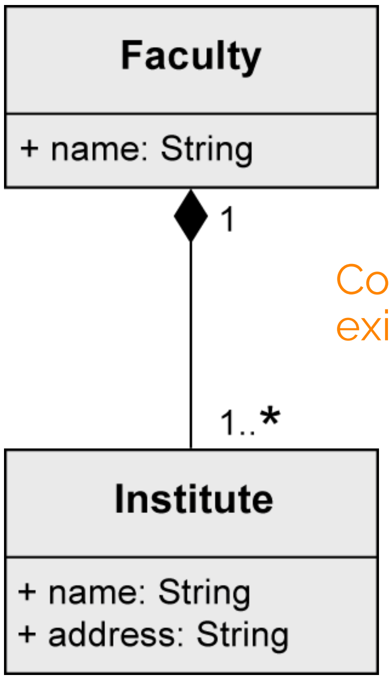


Indication of a generalization

- *“There is a distinction between research and administrative personnel.”*
- *“Some research associates hold courses. Then they are called lecturers.”*

Step 3: identifying relationships (2/6)

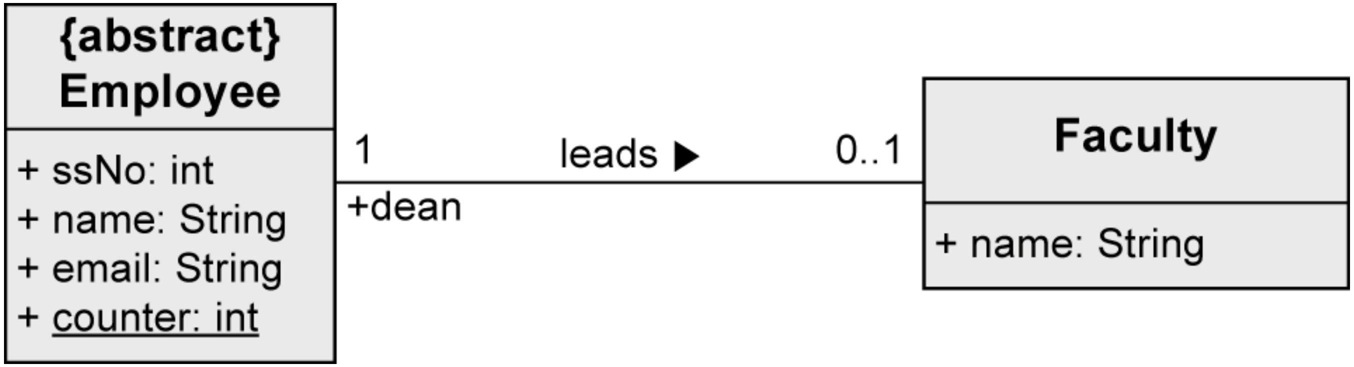
“A university consists of multiple faculties which are composed of various institutes.”



Composition to show existence dependency

Step 3: identifying relationships (3/6)

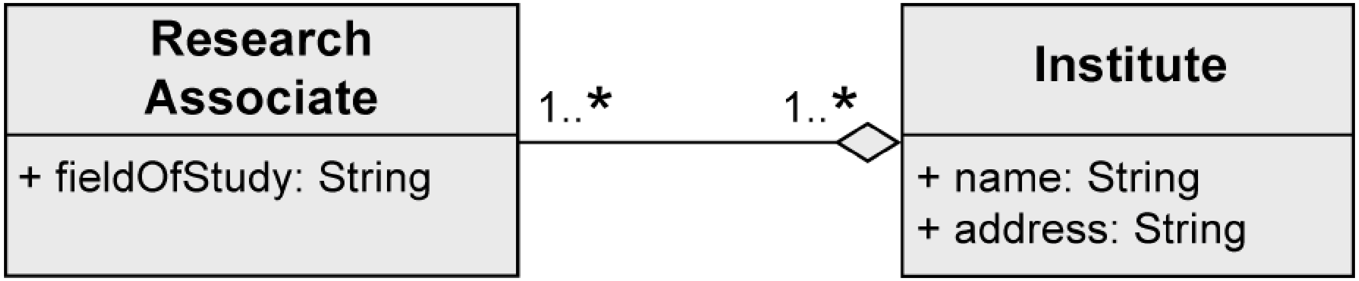
"Each faculty is led by a dean, who is an employee of the university"



In the `leads` relationship, the `Employee` takes the role of a dean.

Step 3: identifying relationships (4/6)

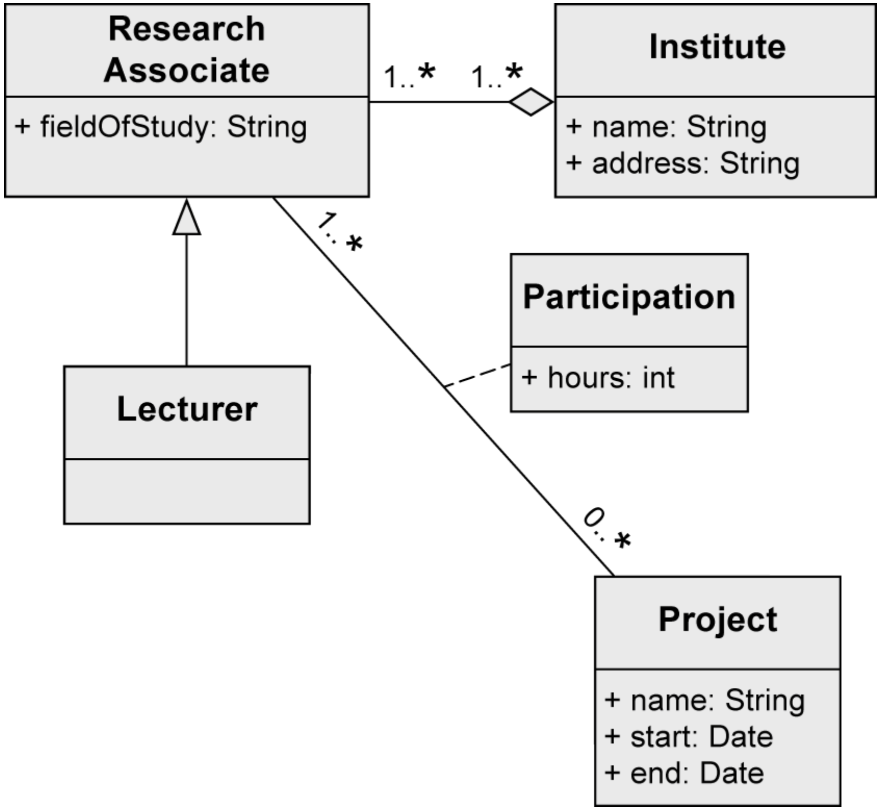
“Research associates are assigned to at least one institute.”



Aggregation to show that `ResearchAssociates` are part of an `Institute`, but there is no existence dependency

Step 3: identifying relationships (5/6)

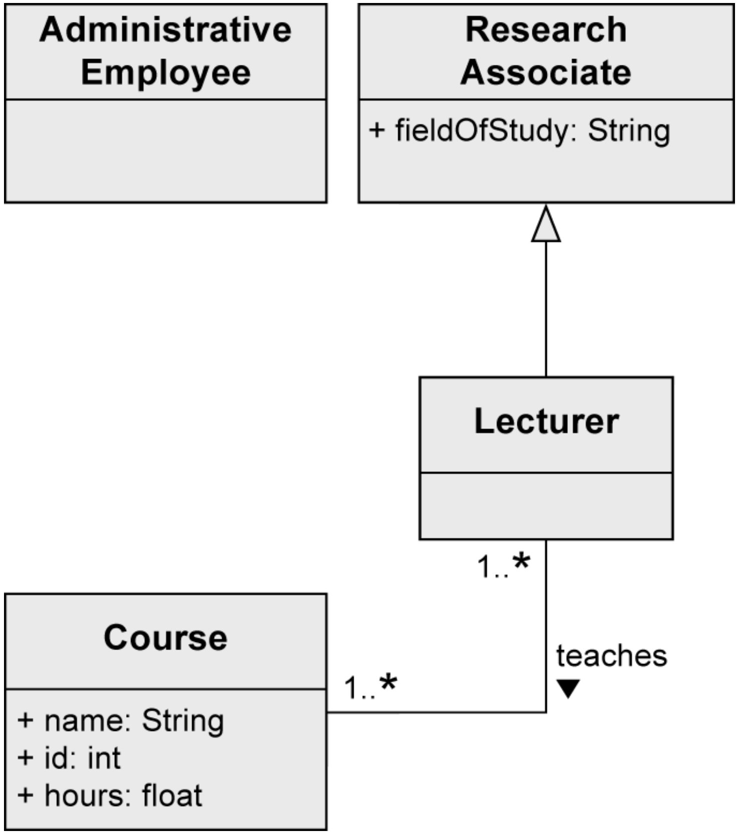
"Furthermore, research associates can be involved in projects for a certain number of hours."



Association class enables to store the number of hours for every `Project` of every `ResearchAssociate`

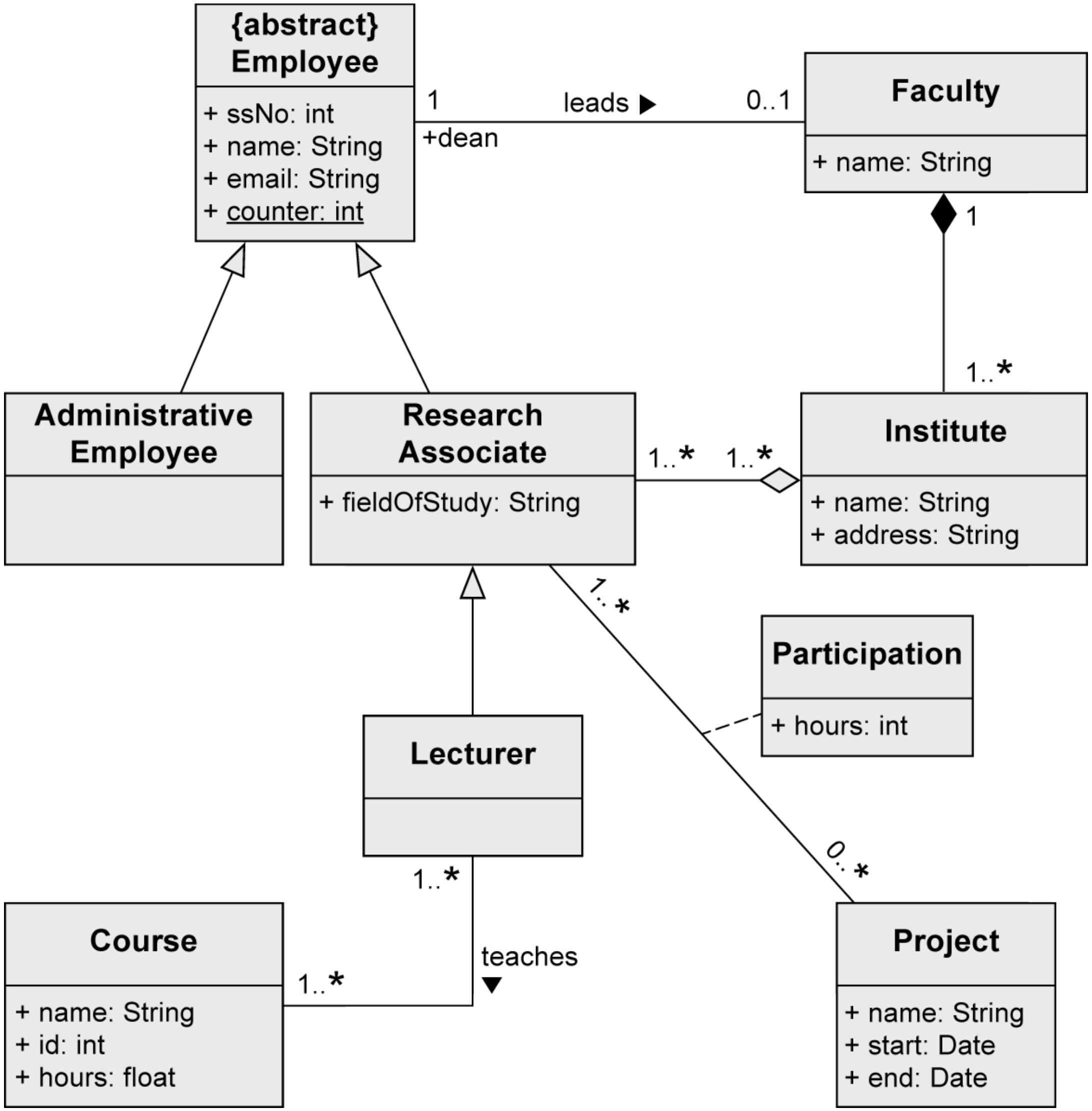
Step 3: identifying relationships (6/6)

“Some research associates hold courses. Then they are called lecturers.”

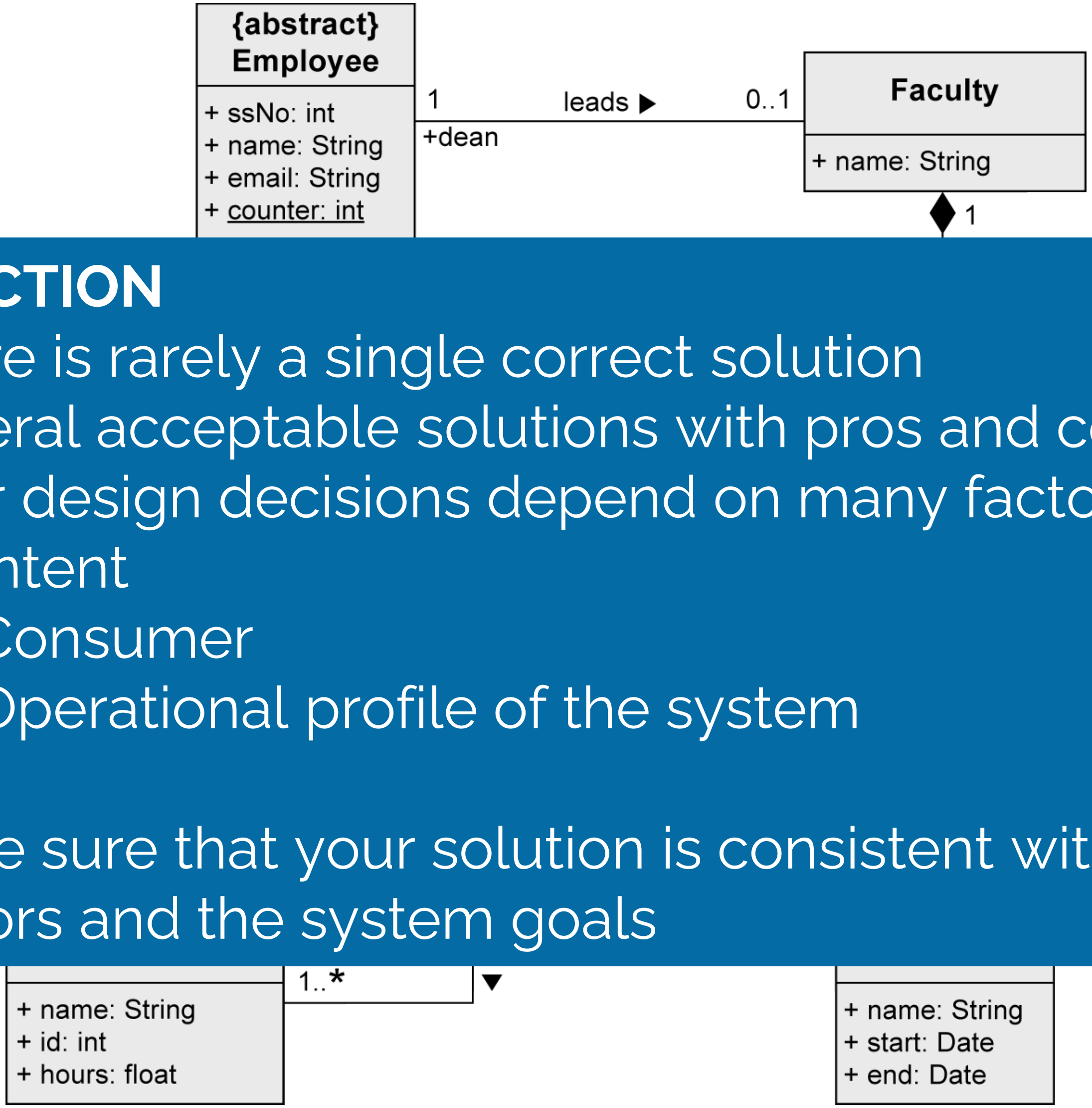


- `Lecturer` inherits all characteristics, associations, and aggregations from `ResearchAssociate`
- `Lecturer` has a `teaches` association with `Course`

Complete class diagram



Complete class diagram



REFLECTION

- There is rarely a single correct solution
- Several acceptable solutions with pros and cons
- Your design decisions depend on many factors, like:
  - Intent
  - Consumer
  - Operational profile of the system
  - …
- Make sure that your solution is consistent with these factors and the system goals