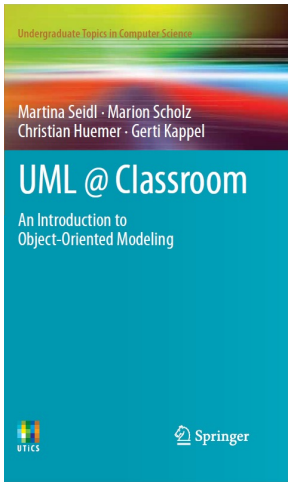
Business Informatics Group

Vienna University of Technology



Object-Oriented Modeling

Class Diagram

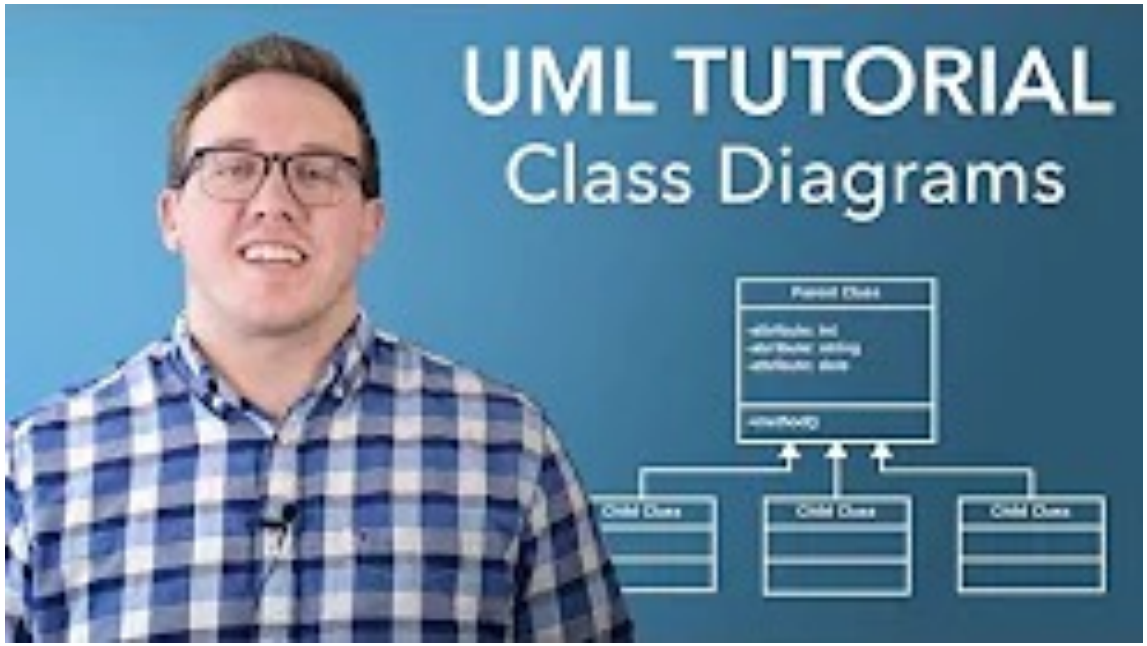
Slides accompanying UML@Classroom
Version 1.0

Business Informatics Group

Institute of Software Technology and Interactive Systems
Vienna University of Technology

Favoritenstraße 9-11/188-3, 1040 Vienna, Austria
phone: +43 (1) 58801-18804 (secretary), fax: +43 (1) 58801-18896
office@big.tuwien.ac.at, www.big.tuwien.ac.at

Introduction



https://youtu.be/UI6lqHOVHic

Class Diagram

- We use the class diagram to model the static structure of a system, thus describing the elements of the system and the relationships between them. These elements and the relationships between them do *not change over time*.
- For example, **students** have a *name* and a *number* and *attend various courses*. This sentence covers a small part of the university structure and does not lose any validity even over years. It is only the specific students and courses that change.
- The class diagram is without doubt the most widely used UML diagram. It is applied in various phases of the software development process.

Class Diagram: Level of Details

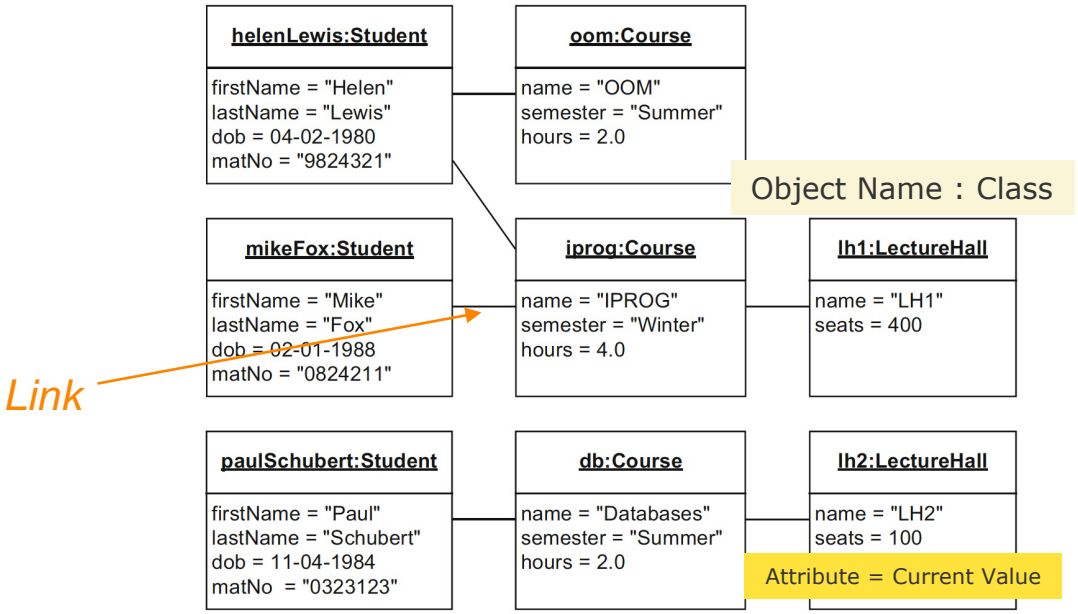
- The level of detail or abstraction of the class diagram is different in each phase.
- In the early project phases, a class diagram allows you to create a *conceptual view of the system* and to *define the vocabulary to be used*. In the context of object-oriented programming, the class diagram visualizes the classes a software system consists of and the relationships between these classes.
- Due to its simplicity and its popularity, the class diagram is ideally suited for quick sketches and documentation purposes. However, you can also use it to *generate program code automatically*.

Object Diagram

- Before we introduce the concepts of the class diagram, let us first take a look at objects, which are modeled in **object diagrams**.
- Object diagrams allow you to depict concrete objects that appear in a system at a specific point in time.
- Classes provide schemas for characterizing objects and **objects** are instances of **classes**.
- The object diagram visualizes instances of classes that are modeled in a class diagram.

Objects

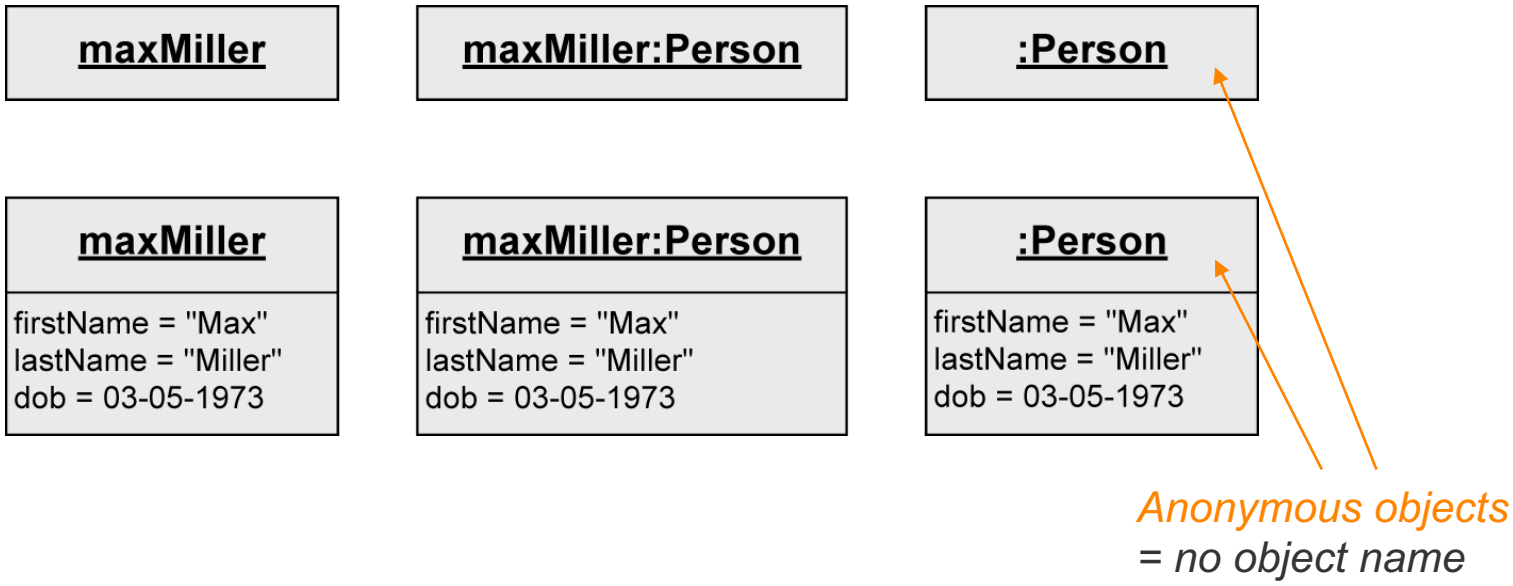
- A system contains numerous different individuals. Individuals might be not only persons but also animals, plants, inanimate objects, artifacts, etc. that can be identified uniquely.
  - For example, as part of her **IT Studies** program, **Helen Lewis** attends the lecture **Object-Oriented Modeling (OOM)** at the university.
  - Helen Lewis, IT Studies, and Object-Oriented Modeling are individuals (*concrete objects*) in a university administration system and are in a relationship with one another.



Object

- The characteristics of an object include its structural characteristics (**attributes**) and its behavior (in the form of **operations**).
- Whilst concrete values are assigned to the attributes in the object diagram, operations are generally not depicted.
- Operations are identical for all objects of a class and are therefore usually described exclusively for the class.

- Alternative notations:



From Object to Class

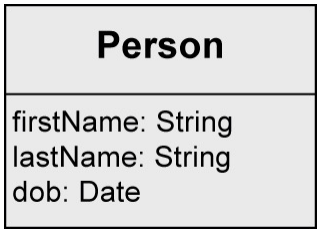
- Individuals of a system often have **identical characteristics** and **behavior**.
- A class is a construction plan for a set of **similar objects of a system**.

- Objects are instances of classes

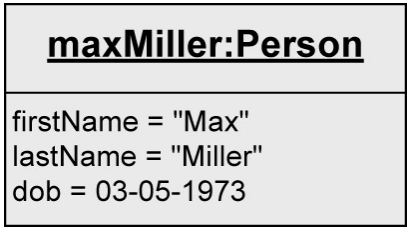
- Attributes: structural characteristics of a class
  - Different value for each instance (= object)

- Operations: behavior of a class
  - Identical for all objects of a class
    → not depicted in object diagram

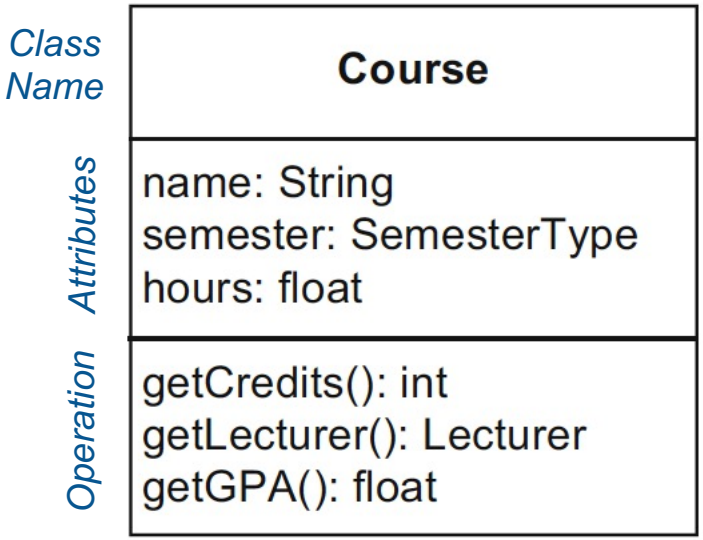
Class



Object of that class



Class

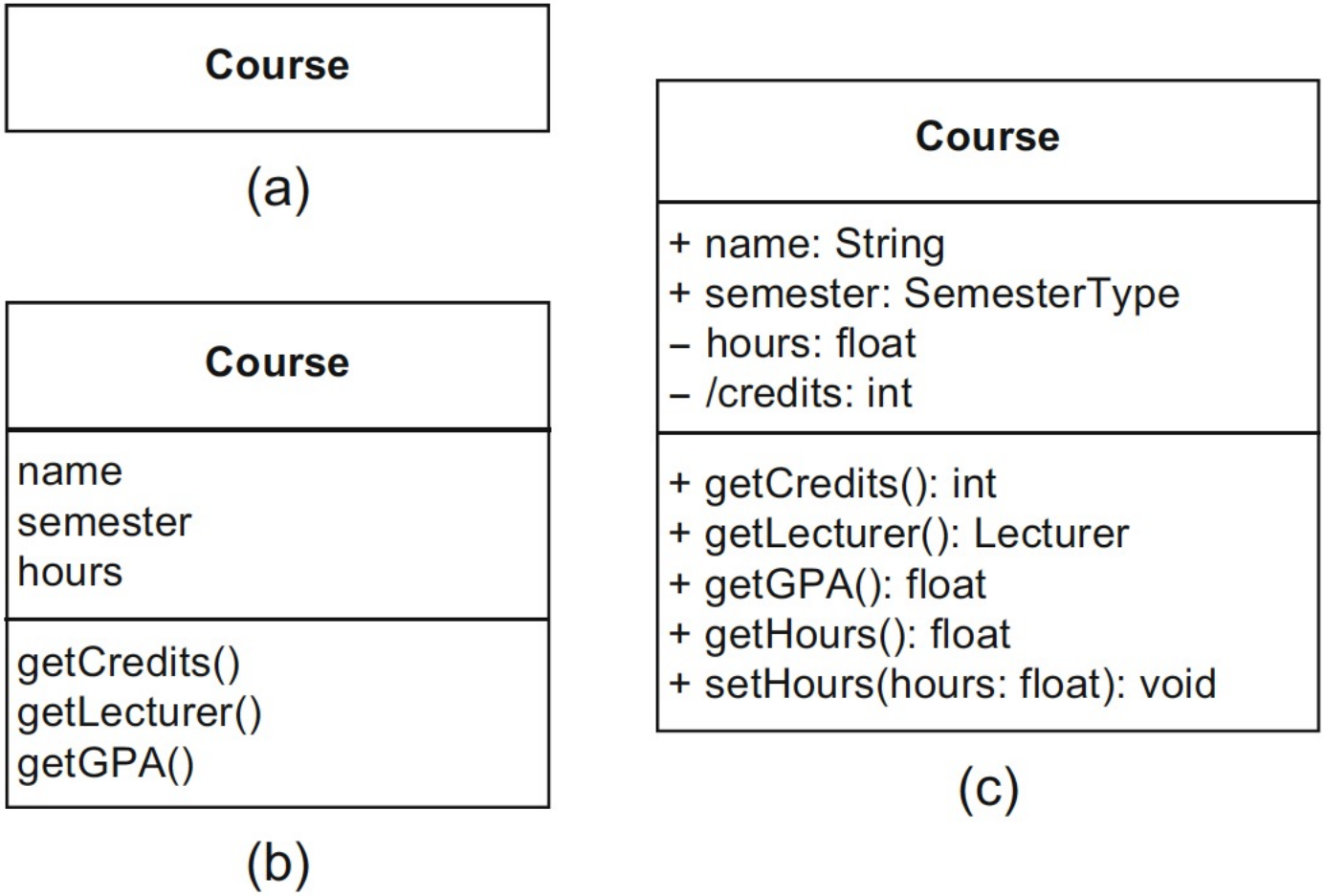


```
class Course {

  String name;
  SemesterType semester;
  float hours;

  int getCredits();
  Lecturer getLecturer();
  float getGPA();
}
```

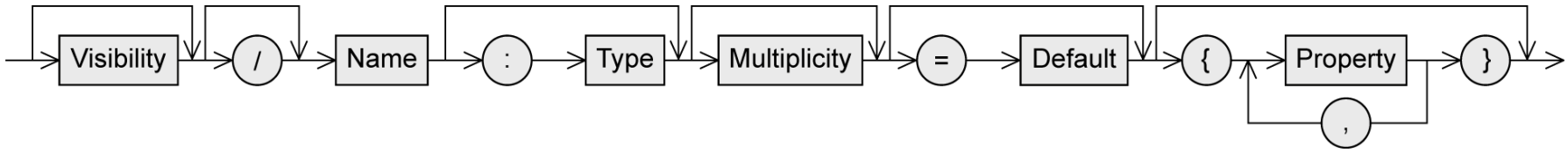
Class Abstraction

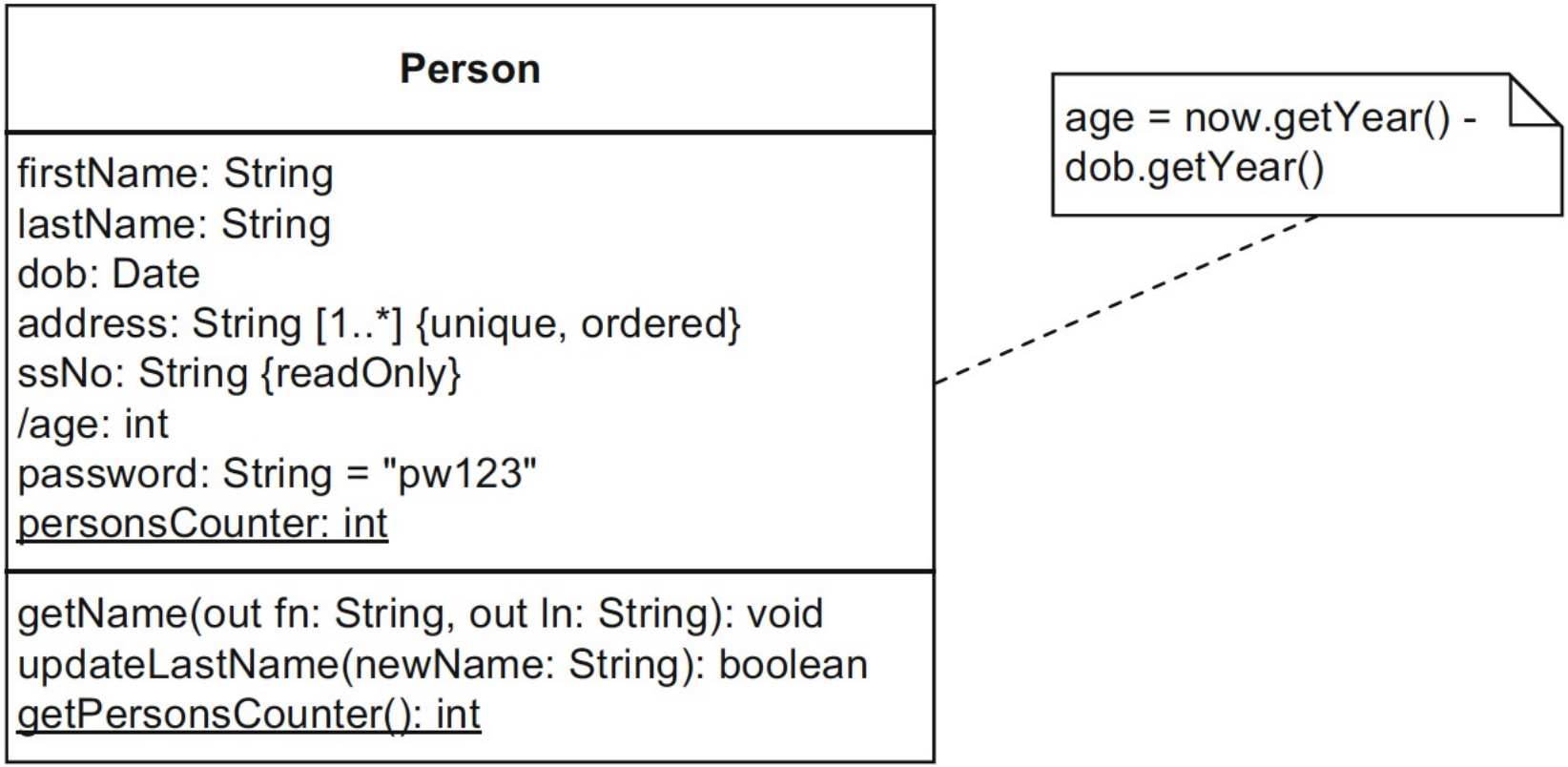
- To ensure that a model remains **clear** and **understandable**, we generally do **not** model all of the details of the content:
  - we only include the information that is relevant for the moment and for the system to be implemented.
  - This means that we abstract from reality to make the model less complex and to avoid an unnecessary flood of information.
- In the model, we restrict ourselves to the essentials.
  - For example, in a university administration system, it is important to be able to manage the names and numbers of the students; in contrast, their shoe size is irrelevant and is therefore not included.

Notation

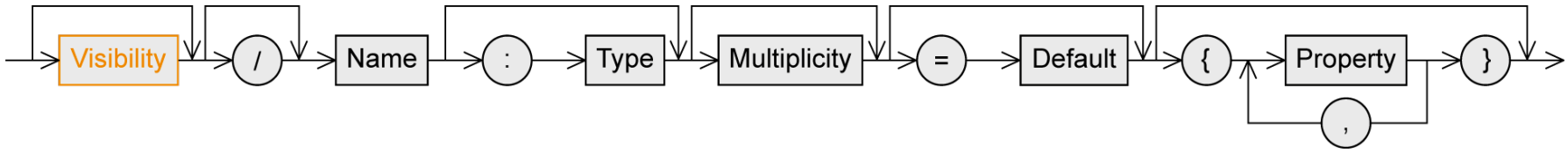

Course
(a)
Course
name
semester
hours
getCredits()
getLecturer()
getGPA()
(b)
Course
+ name: String
+ semester: SemesterType
– hours: float
– /credits: int
+ getCredits(): int
+ getLecturer(): Lecturer
+ getGPA(): float
+ getHours(): float
+ setHours(hours: float): void
(c)

Attribute Syntax





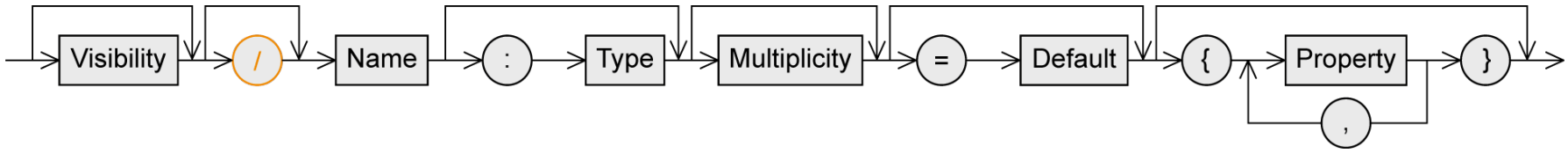
Attribute Syntax - Visibility



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: everybody
  - - ... private: only the object itself
  - # ... protected: class itself and subclasses
  - ~ ... package: classes that are in the same package

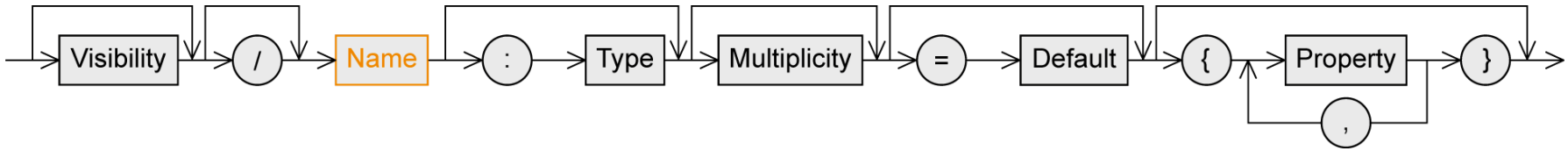
Attribute Syntax - Derived Attribute



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

- Attribute value is derived from other attributes
  - **age**: calculated from the date of birth

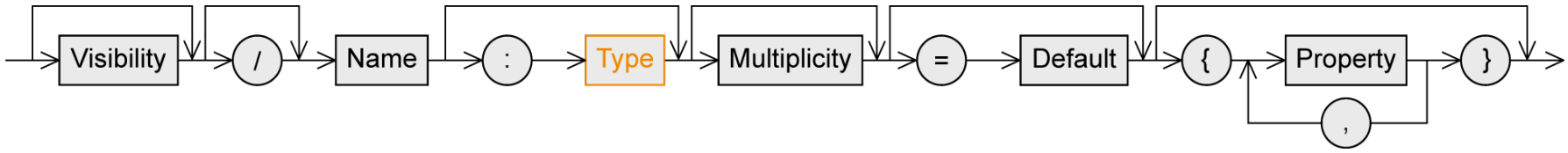
Attribute Syntax - Name



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

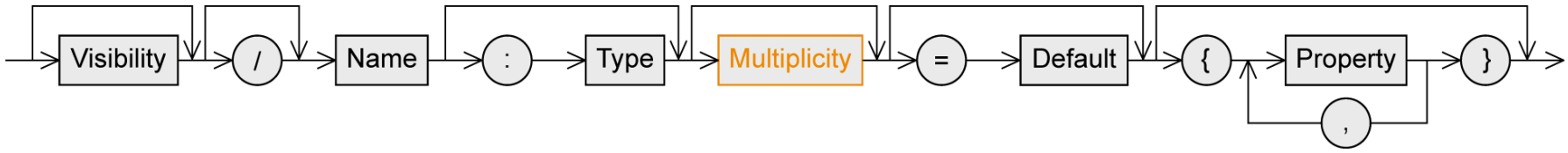
- Type
  - User-defined classes
  - Data type
    - Primitive data type
      - Pre-defined: Boolean, Integer, UnlimitedNatural, String
      - User-defined: «**primitive**»
      - Composite data type: «**datatype**»
    - Enumerations: «**enumeration**»

«primitive» **Float**
round(): void

«datatype» **Date**
day month year

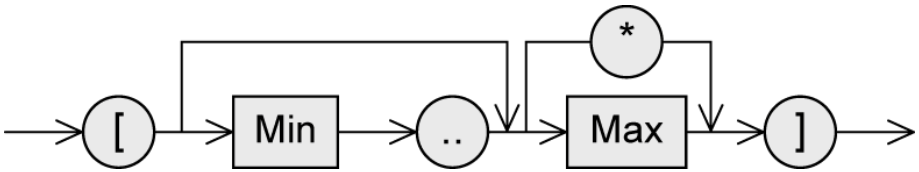
«enumeration» **AcademicDegree**
bachelor master phd

Attribute Syntax - Multiplicity

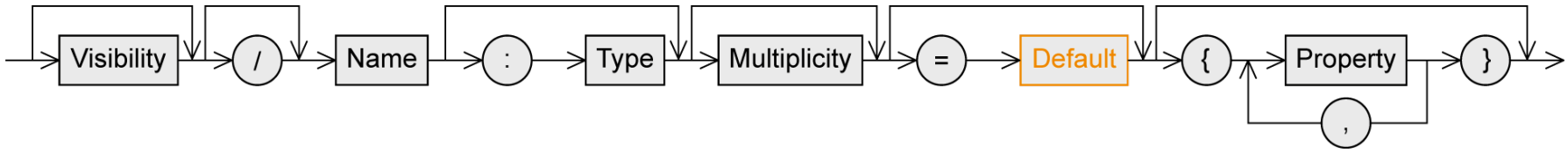


Person
firstName: String lastName: String dob: Date address: String[1..*] {unique, ordered} ssNo: String {readOnly} /age: int password: String = "pw123" personsNumber: int

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



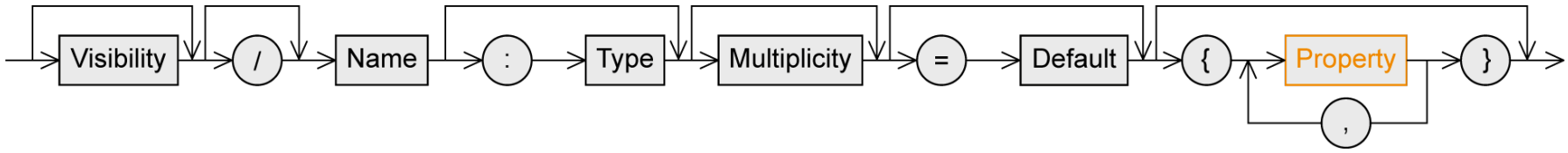
Attribute Syntax – Default Value



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

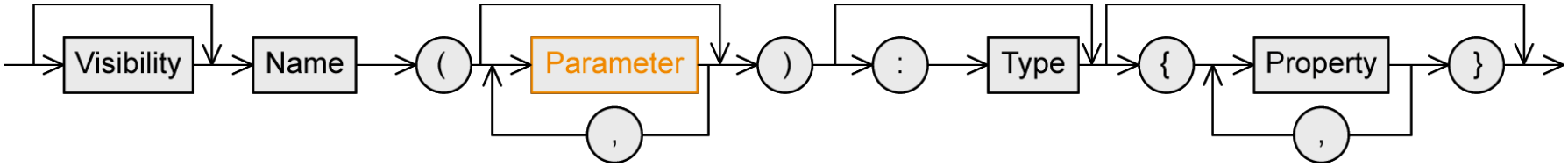
Attribute Syntax – Properties

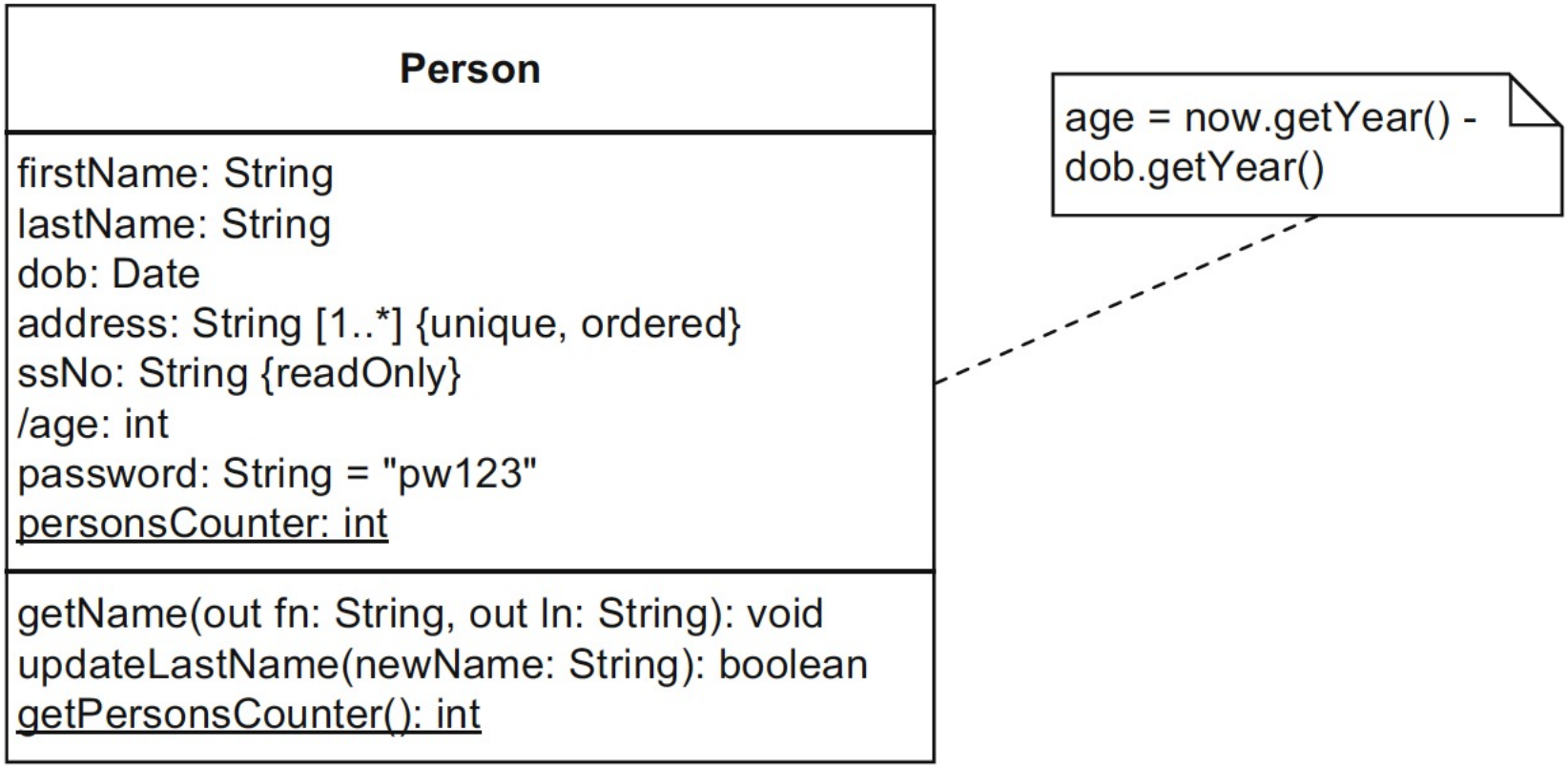


Person
firstName: String lastName: String dob: Date address: String[1..*] {unique, ordered} ssNo: String {readOnly} /age: int password: String = "pw123" personsNumber: int

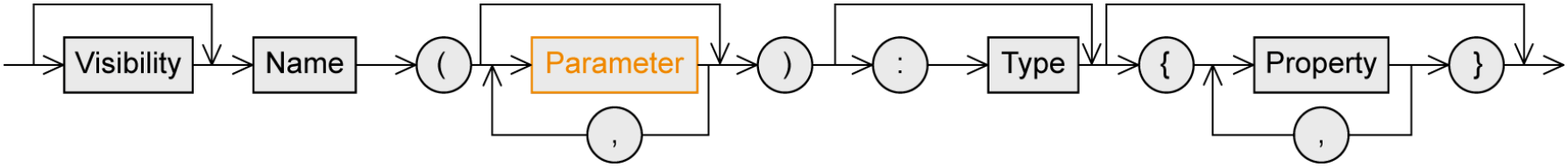
- Pre-defined properties
  - `{readOnly}` … value cannot be changed
  - `{unique}` … no duplicates permitted
  - `{non-unique}` … duplicates permitted
  - `{ordered}` … fixed order of the values
  - `{unordered}` … no fixed order of the values
- Attribute specification
  - Set: `{unordered, unique}`
  - Multi-set: `{unordered, non-unique}`
  - Ordered set: `{ordered, unique}`
  - List: `{ordered, non-unique}`

Operation Syntax





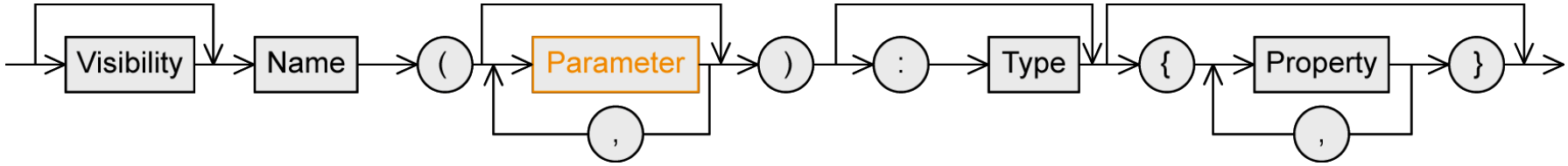
Operation Syntax - Visibility



- Visibilities are used to realize **information hiding**, an important concept in computing. Marking the attributes that represent the state of an object as private protects this state against unauthorized access.
- Access is therefore only possible via a clearly defined interface, such as via operations that are declared public.

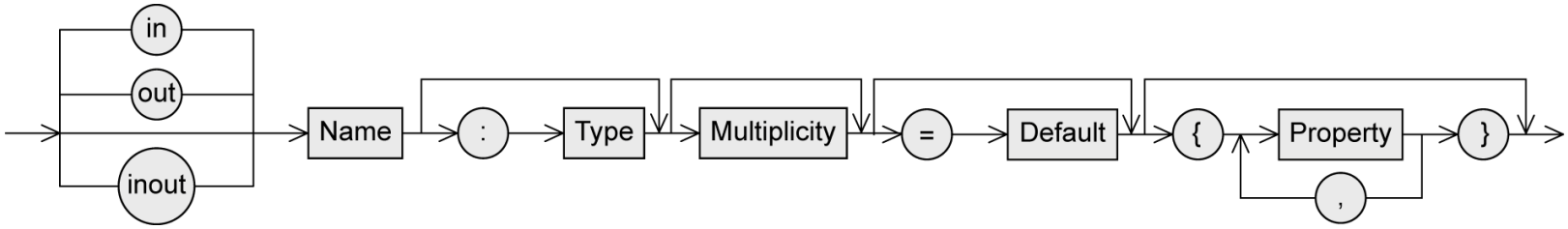
Name	Symbol	Description
public	+	Access by objects of any classes permitted
private	−	Access only within the object itself permitted
protected	#	Access by objects of the same class and its subclasses permitted
package	~	Access by objects whose classes are in the same package permitted

Operation Syntax - Parameters

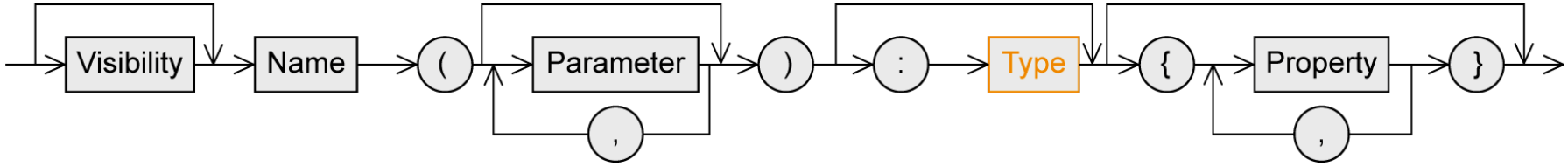


Person
...
+ getName(out fn: String, out ln: String): void + updateLastName(newName: String): boolean + getPersonsNumber(): int

- Notation like 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



Operation Syntax - Type

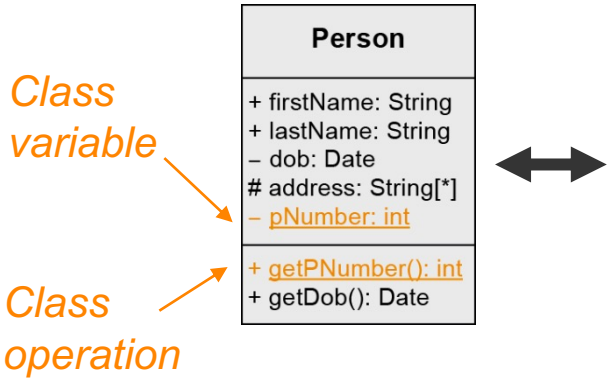


Person
...
getName(out fn: String, out ln: String): void updateLastName(newName: String): boolean getPersonsNumber(): int

- Type of the return value

Class Variable and Class Operation

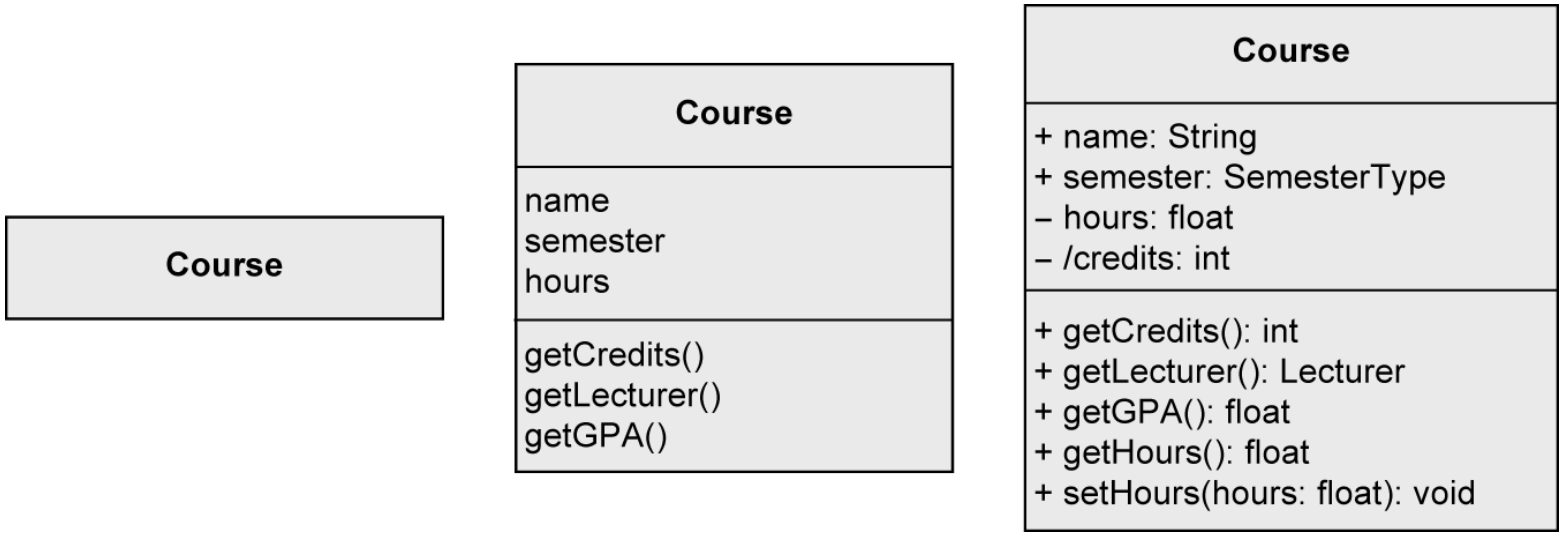
- Instance variable (= instance attribute): attributes defined on instance level
- Class variable (= class attribute, static attribute)
  - Defined only once per class, i.e., shared by all instances of the class
  - E.g., counters for the number of instances of a class, constants, etc.
- Class operation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




```
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() {…}
}
```

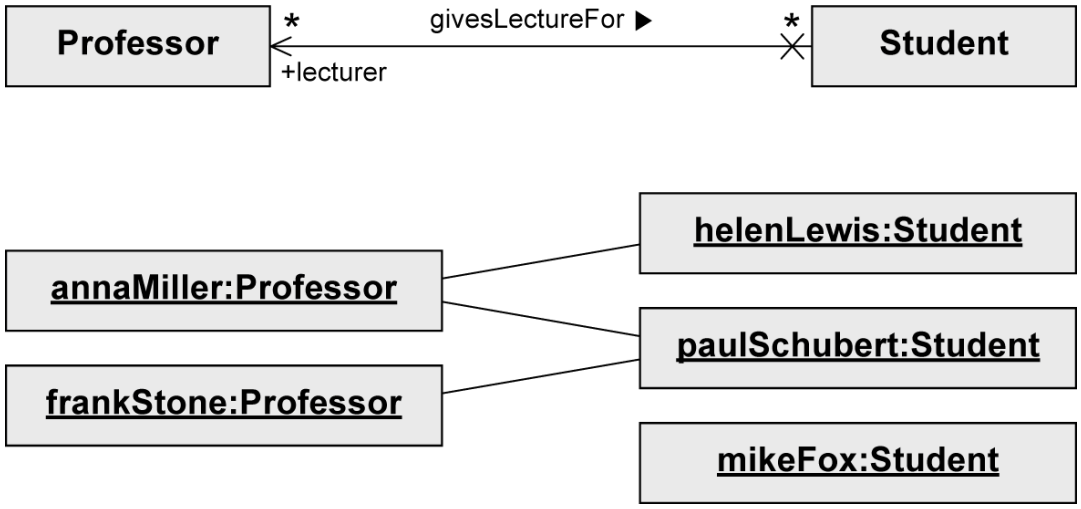
Specification of Classes: Different Levels of Detail

coarse-grained *fine-grained*

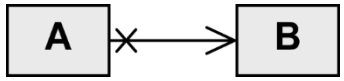


Association

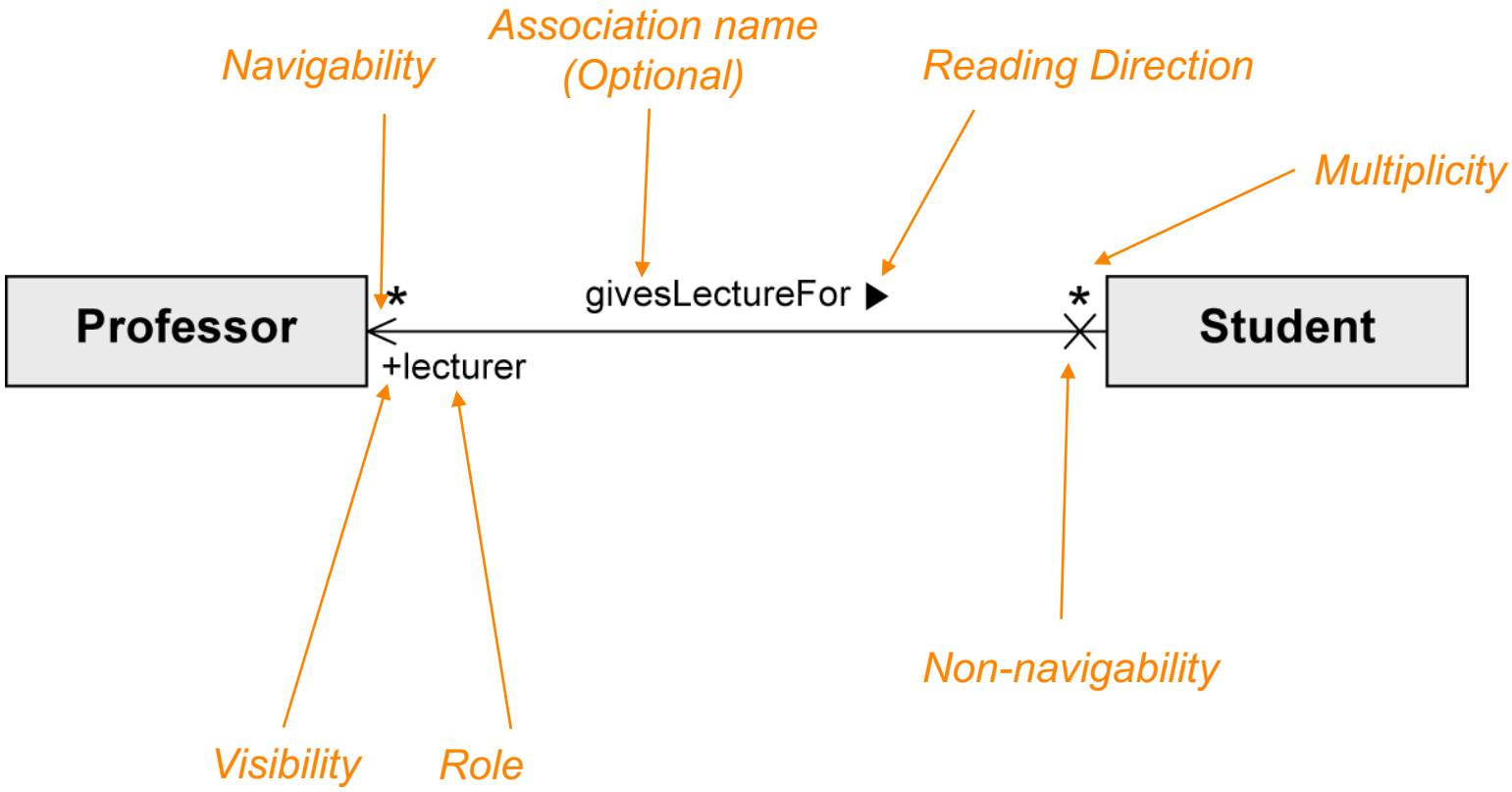
- **Associations** between classes *model possible relationships, known as*
- *links, between instances of the classes*.
- They describe which classes are potential communication partners. If their attributes and operations have the corresponding visibilities, the communication partners can access each other's attributes and operations.
- A class diagram can be viewed as a **graph** in which the classes represent the nodes, and the associations represent the edges.



Binary Association

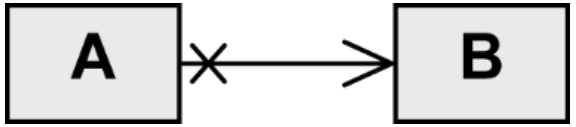


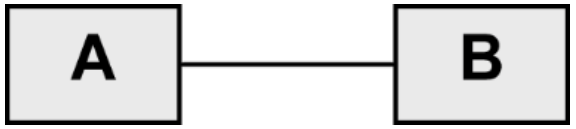
- Connects instances of two classes with one another



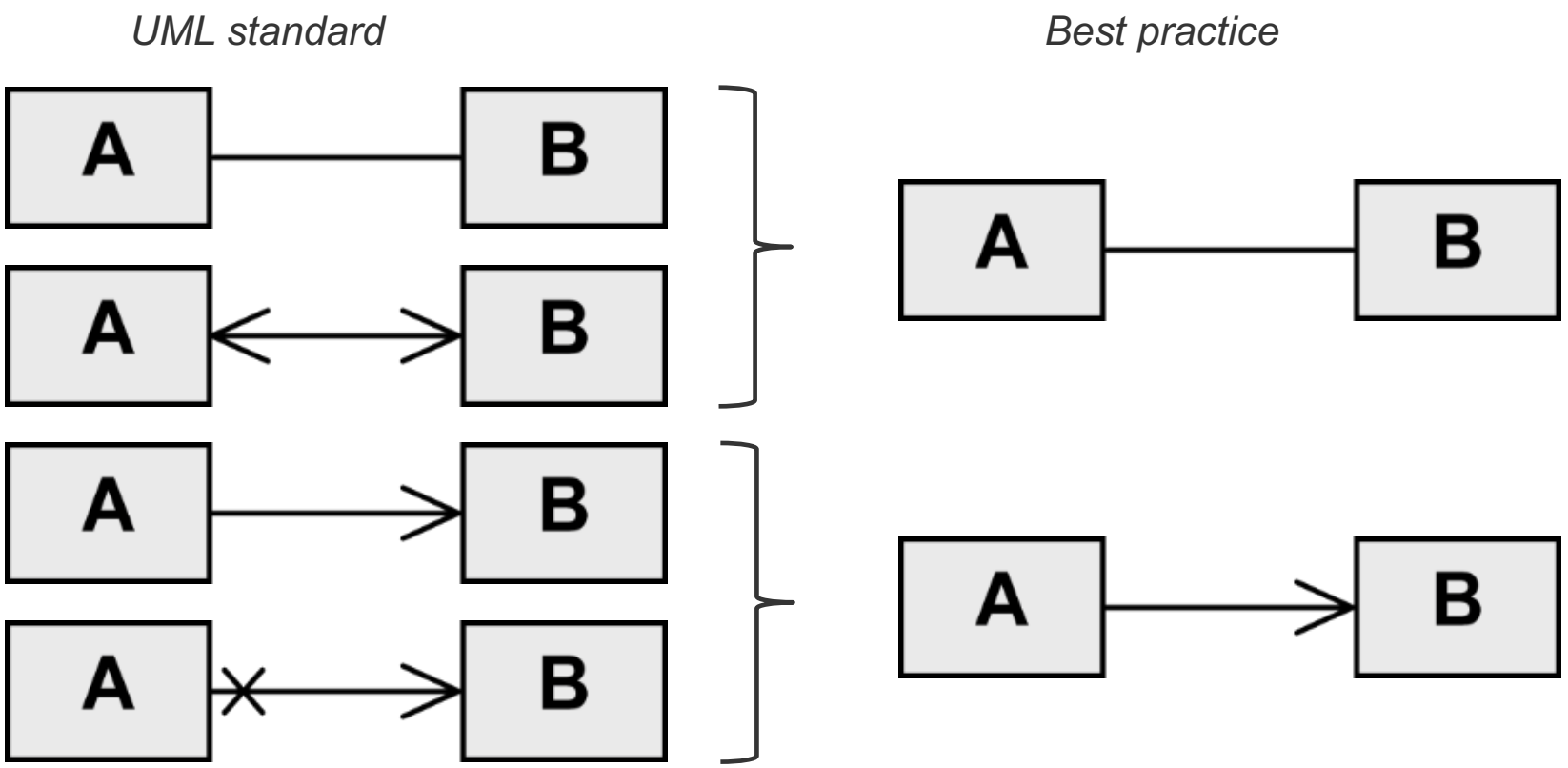
Binary Association - Navigability

- 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 cross
- 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



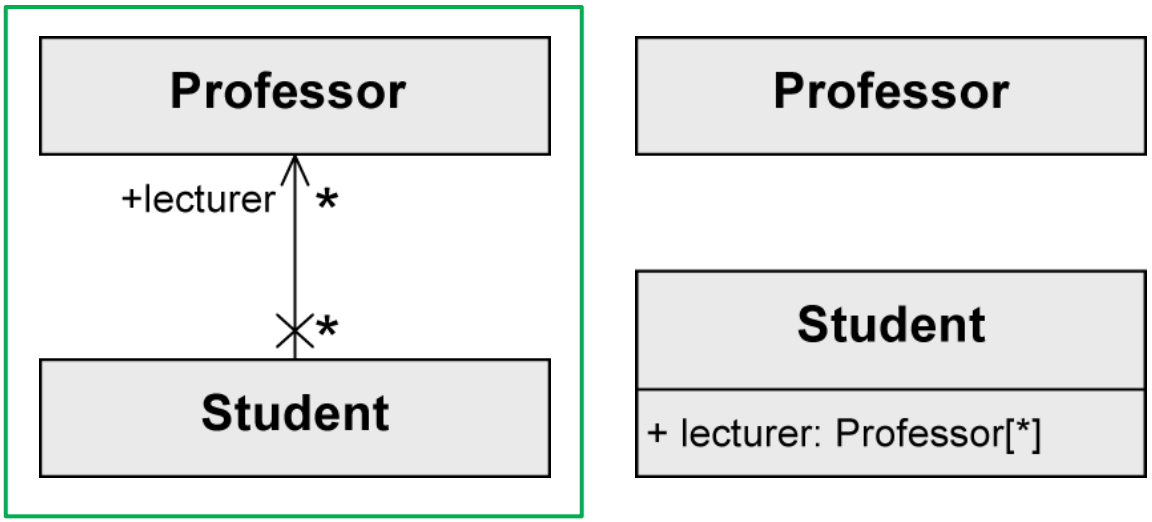


Navigability – UML Standard vs. Best Practice



Binary Association as Attribute

Preferable

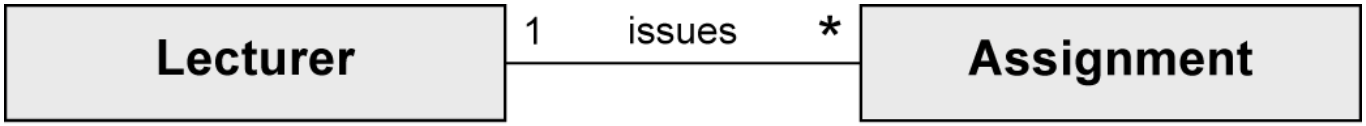


- Java-like notation:

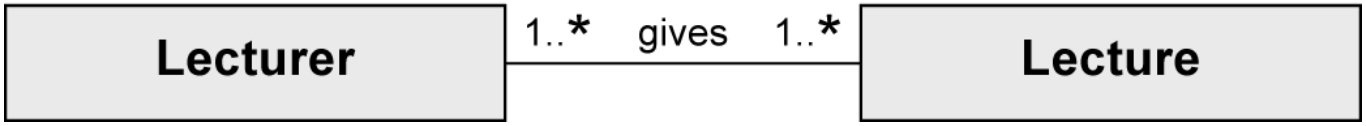
```
class Professor {…}

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

Binary Association – Multiplicity and Role

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

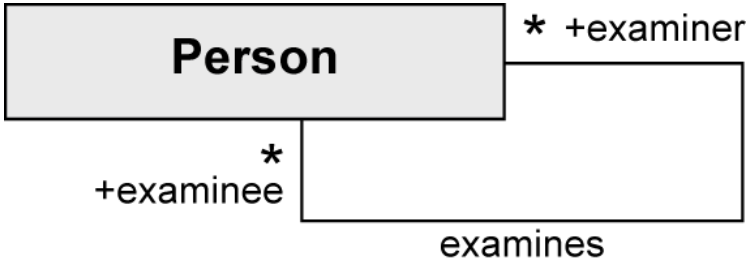


A lecturer may issue no, one, or multiple assignments and that an assignment is issued by exactly one lecturer. No assignment may exist without an association to a lecturer.



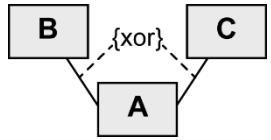
A lecturer gives at least one lecture, and a lecture is given by at least one lecturer.

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

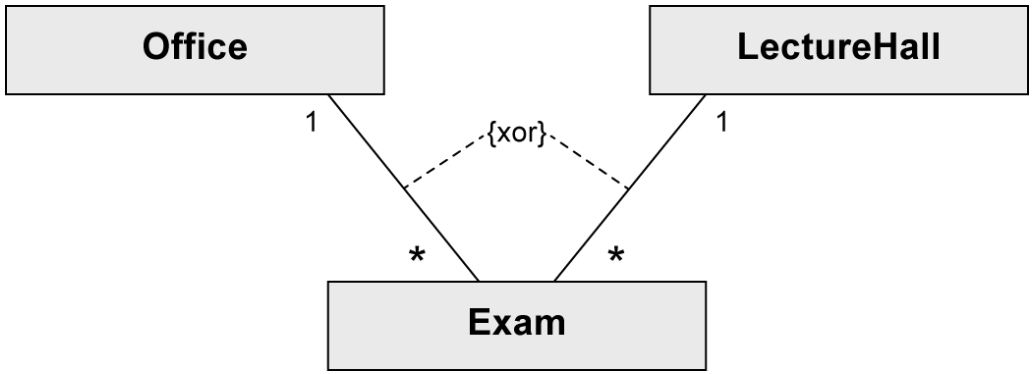


A person in the role of examiner can examine any number (≥ 0) of persons and a person in the role of examinee can be examined by any number of examiners.

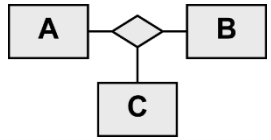
Binary Association – `xor` constraint



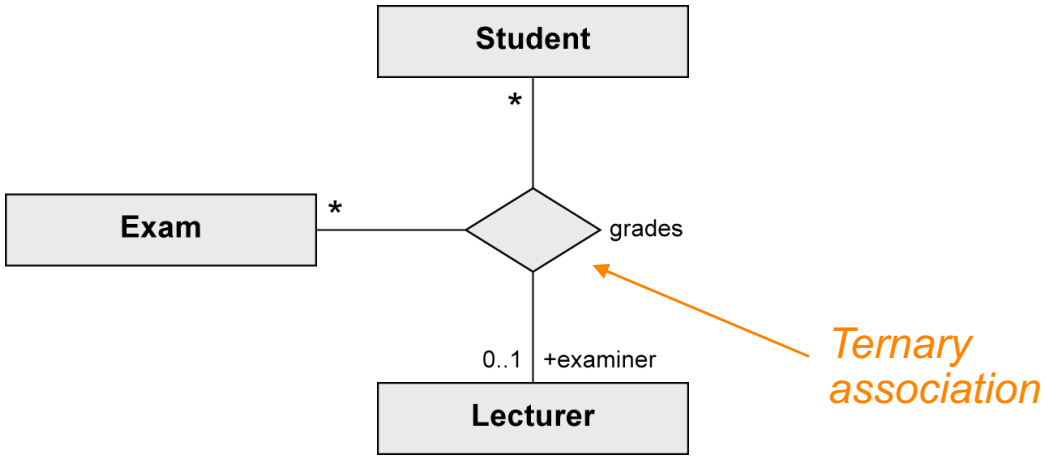
- "exclusive or" constraint
- An object of class **`A`** is to be associated with an object of class **`B`** or an object of class **`C`** but not with both.
- To indicate that two associations from the same class are mutually exclusive, they can be connected by a dashed line labeled {xor}.
- For example, an exam can take place either in an office or in a lecture hall but not in both:



n-ary Association (1/2)

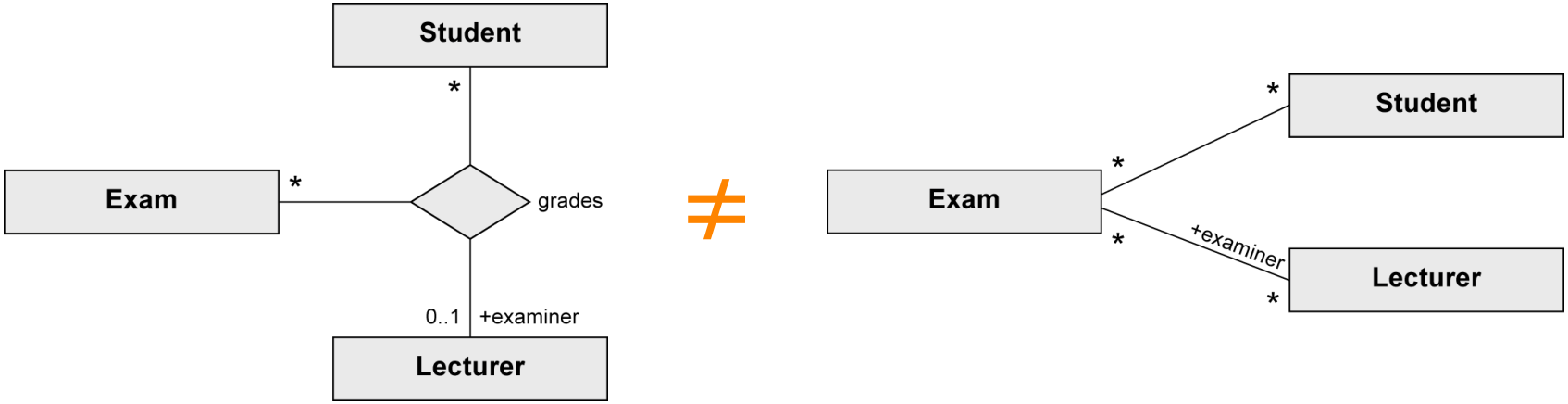


- More than two partner objects are involved in the relationship.
- No navigation directions
- For example:
  - One specific student takes one specific exam with no lecturer (i.e., does not take this exam at all) or with precisely one lecturer.
  - One specific exam with one specific lecturer can of course be taken by any number of students and one specific student can be graded by one specific lecturer for any number of exams.

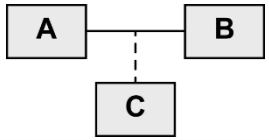


n-ary Association (2/2)

- Example
  - (**`Student`**, **`Exam`**) → (**`Lecturer`**)
    - One student takes one exam with one or no lecturer
  - (**`Exam`**, **`Lecturer`**) → (**`Student`**)
    - One exam with one lecturer can be taken by any number of students
  - (**`Student`**, **`Lecturer`**) → (**`Exam`**)
    - One student can be graded by one **`Lecturer`** for any number of exams

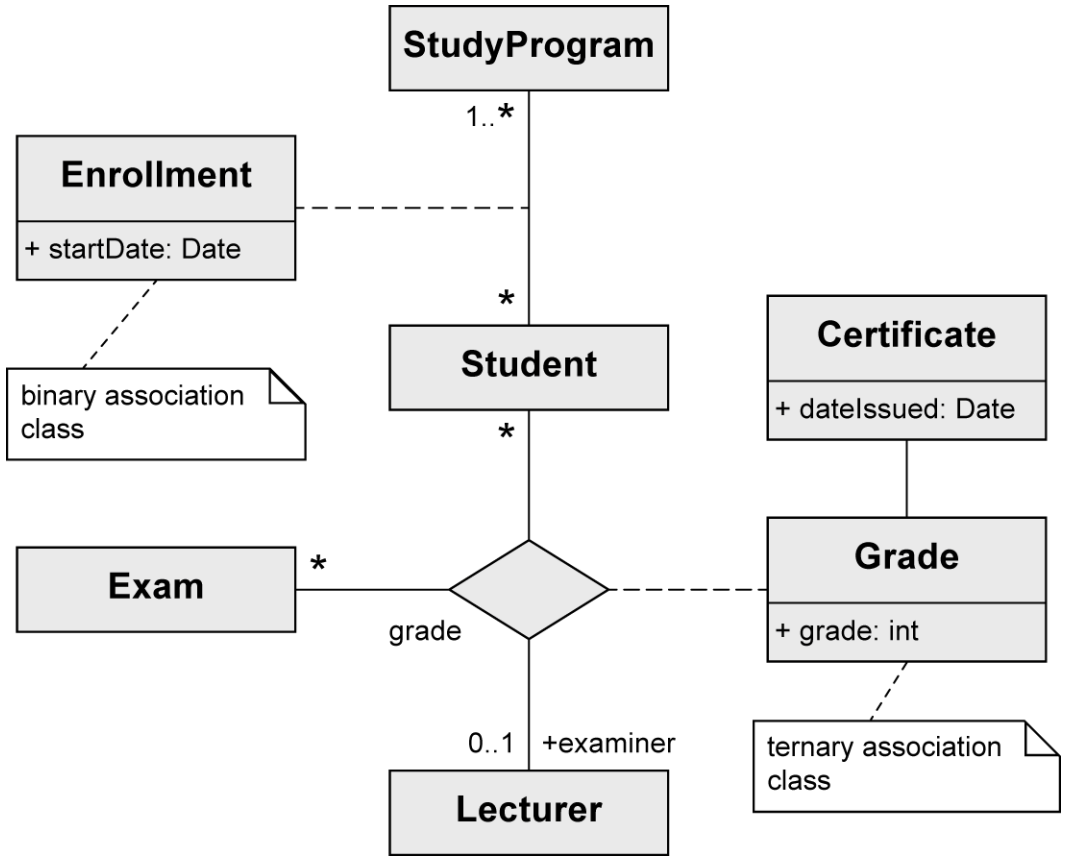


The ternary association clearly shows which lecturer a student passed a specific exam with. For example, it is possible to express that student s1 took the exam e1 with lecturer l1 and that student s2 took the same exam e1 with lecturer l2. With the binary associations, it is only possible to express that the students s1 and s2 took the exam e1 and that exam e1 has two examiners l1 and l2. With this model, you cannot express which lecturer grades which student.



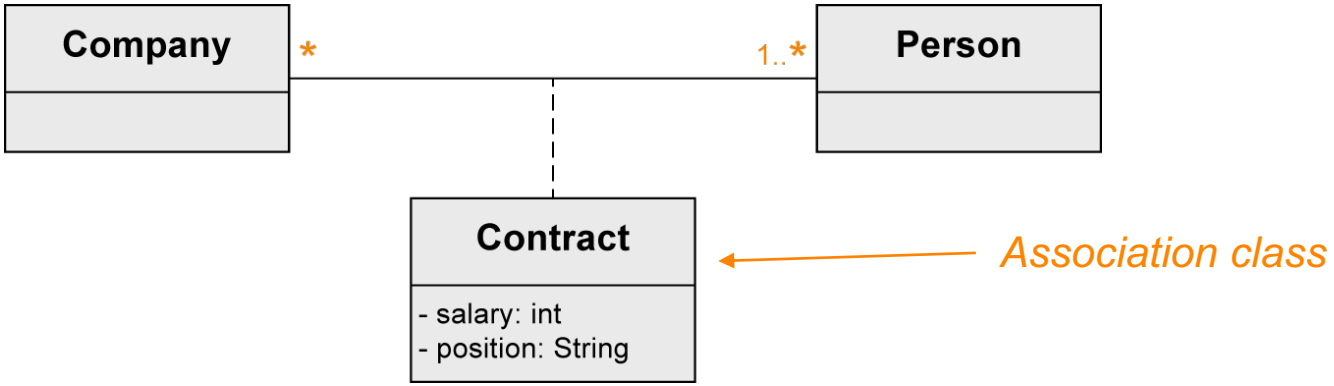
Association Class

- Assign attributes to the relationship between classes rather than to a class itself
- The class and association of an association class must have the same name.

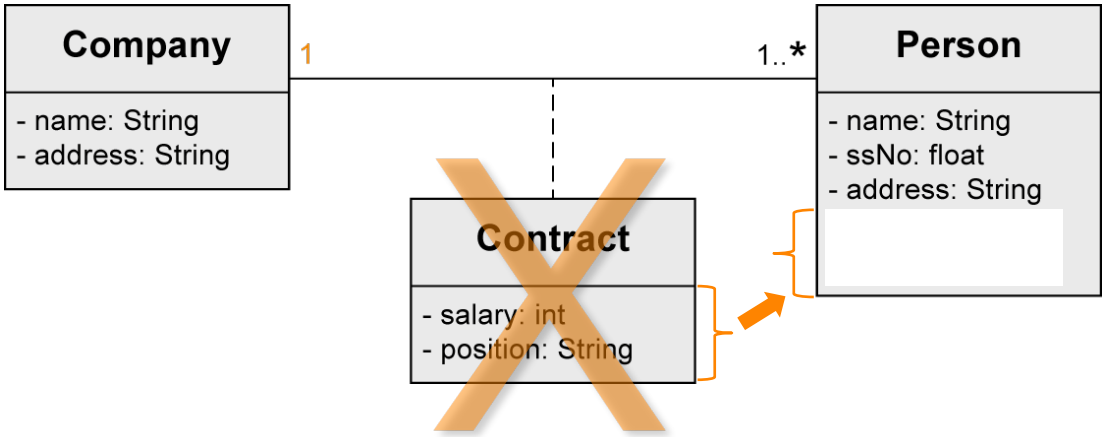


Association Class

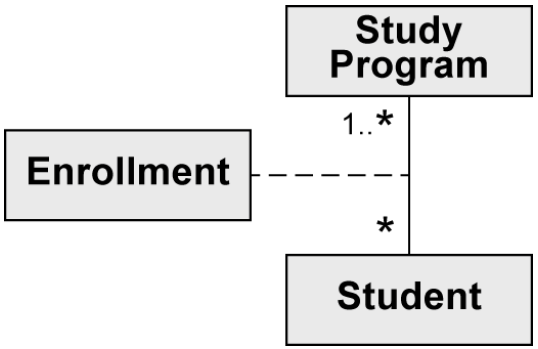
- Necessary when modeling n:m Associations



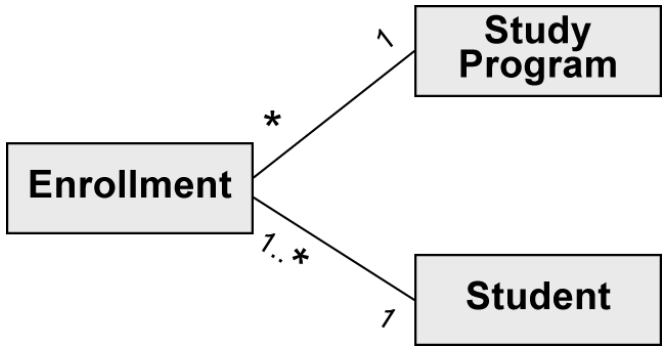
- With 1:1 or 1:n possible but not necessary



Association Class vs. Regular Class



≠

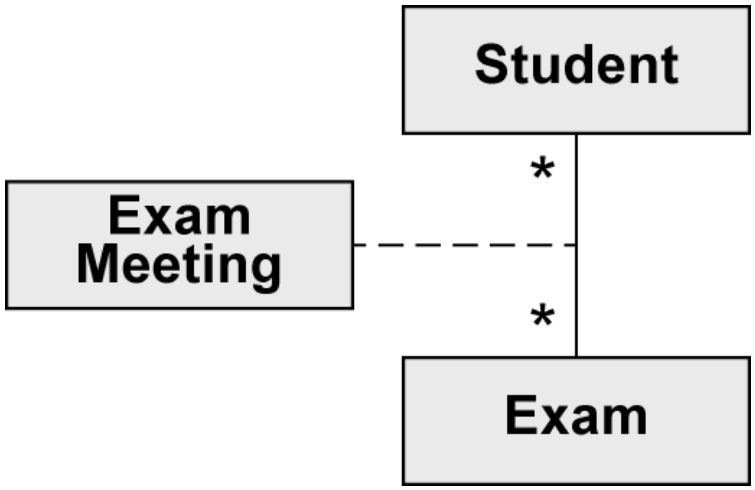


A `Student` *can enroll for one particular* `StudyProgram` *only once*

A `Student` *can have mutiple* `Enrollments` *for one and the same* `StudyProgram`

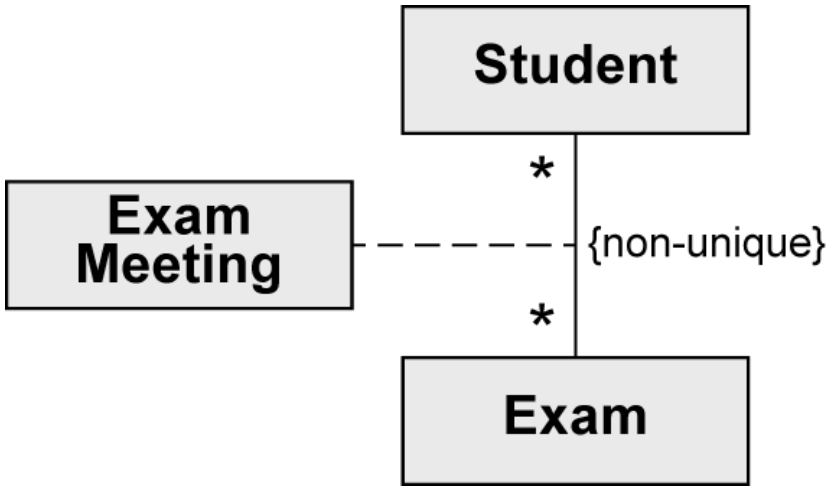
Association Class – `unique/non-unique` (1/2)

- Default: no duplicates



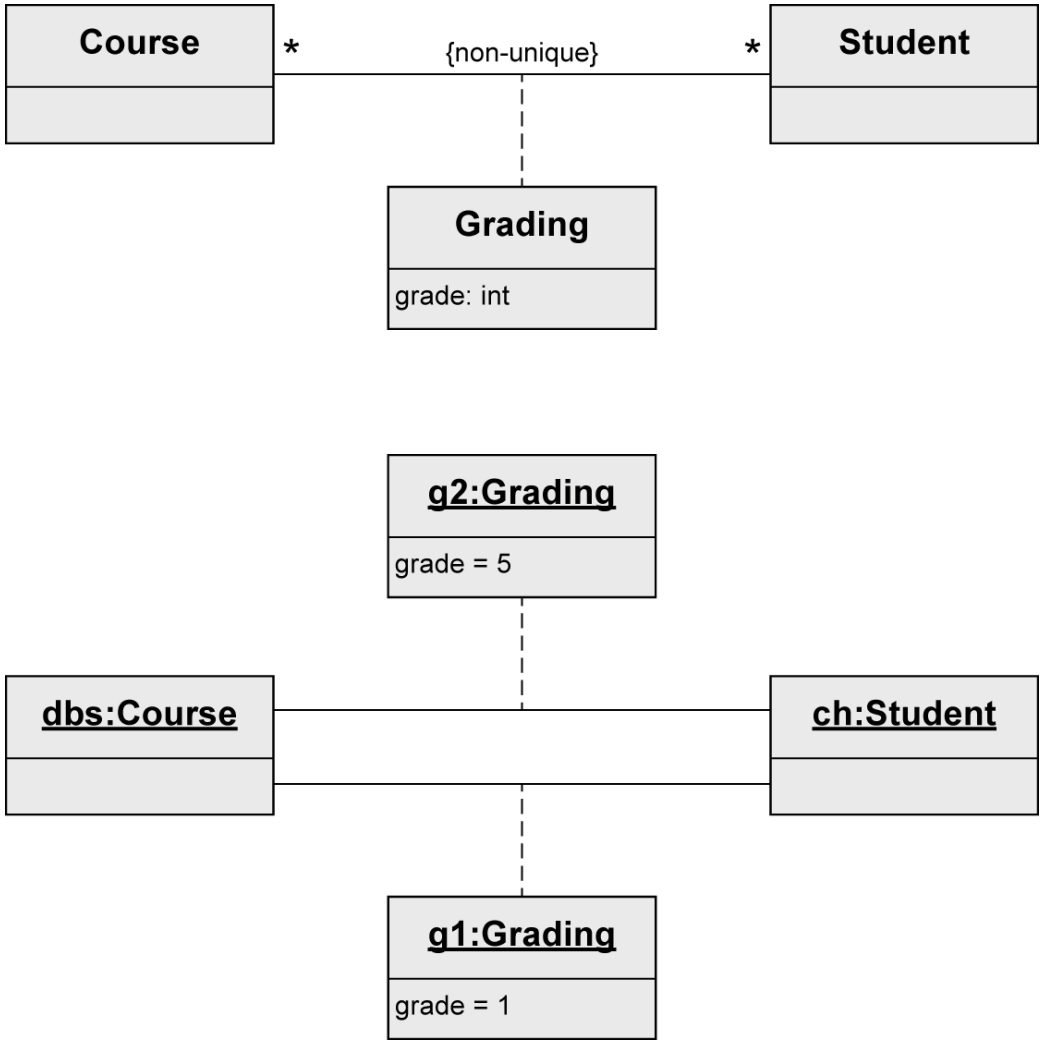
A student can only be granted an exam meeting for a specific exam once.

- non-unique: duplicates allowed



A student can have more than one exam meetings for a specific exam.

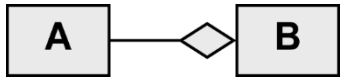
Association Class – `unique/non-unique` (2/2)



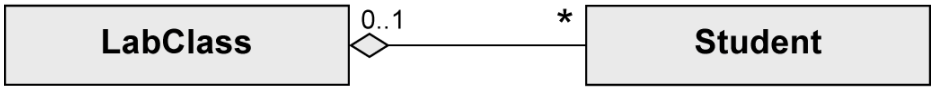
Aggregation

- 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

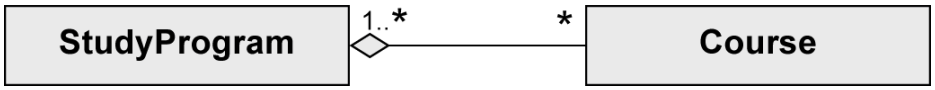
Shared Aggregation



- 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
- Spans a directed acyclic graph
- Syntax: Hollow diamond at the aggregating end
- Example:
  - **`Student`** is part of **`LabClass`**
  - **`Course`** is part of **`StudyProgram`**



A lab class consists of any number of students. However, a student can participate in a maximum of one lab class.



A study program is made up of any (≥ 0) number of courses. A course is assigned to at least one (≥ 1) study program.

Composition

- Existence dependency between the composite object and its parts
- One part can only be contained in at most one composite object at one specific point in time

  Multiplicity at the aggregating end max. 1

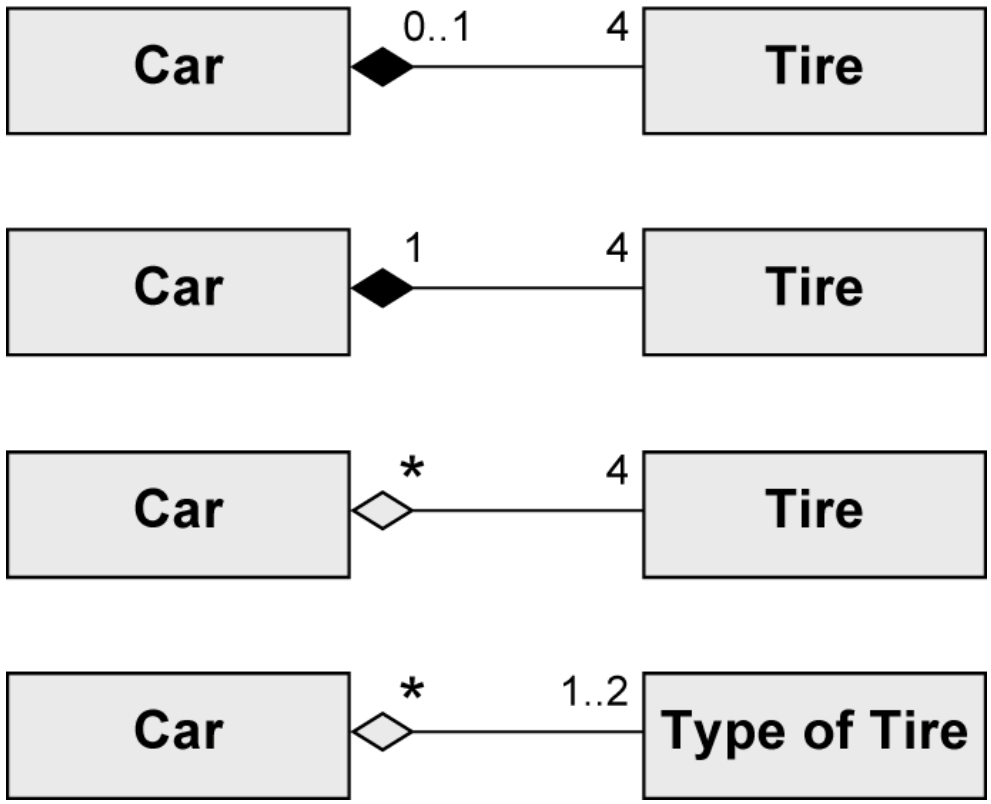
  -> The composite objects form a tree
- If the composite object is deleted, its parts are also deleted.
- Syntax: Solid diamond at the aggregating end
- Example: **`Beamer`** is part of **`LectureHall`** is part of **`Building`**

If the `Building` is deleted, the `LectureHall` is also deleted

The `Beamer` can exist without the `LectureHall`, but if it is contained in the `LectureHall` while it is deleted, the `Beamer` is also deleted

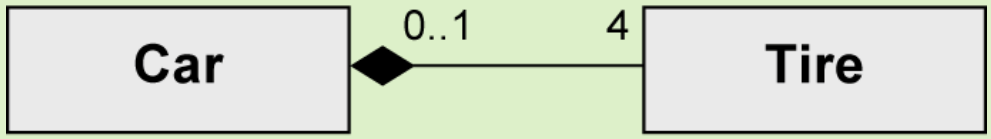
Shared Aggregation and Composition

- Which model applies?



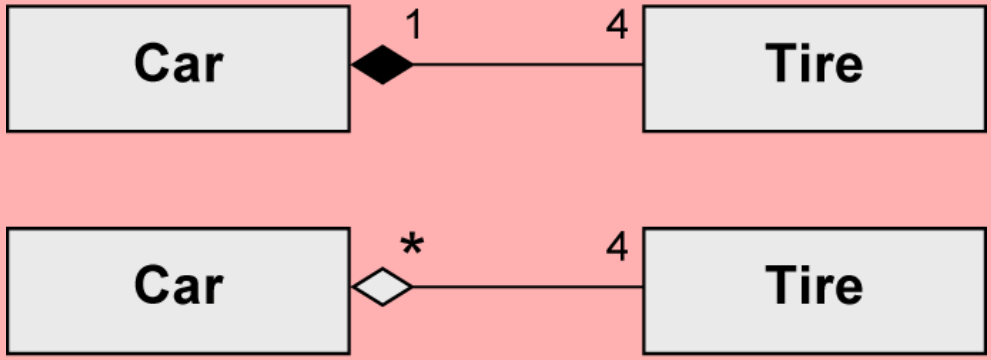
Shared Aggregation and Composition

- Which model applys?



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

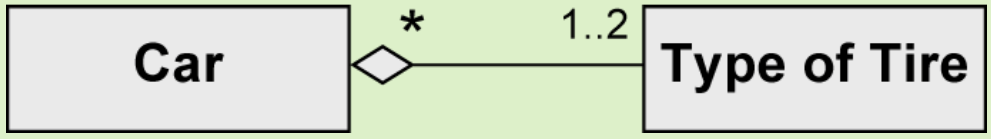
Yes



A `Tire` cannot exist without a `Car`.

A `Tire` can belong to multiple `Cars`

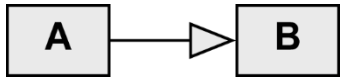
No



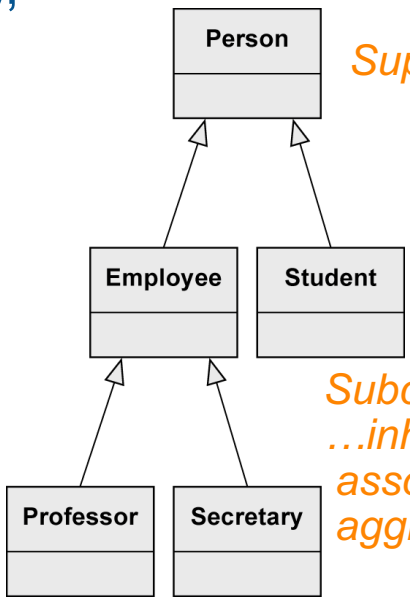
A `Car` has one or two types of `Tires`. Several `Cars` may have the same Type of `Tires`.

Yes

Generalization



- Characteristics (attributes and operations), associations, and aggregations that are specified for a **general** class (*superclass*) are passed on to its *subclasses*.
- Every instance of a subclass is simultaneously an indirect instance of the superclass.
- Subclass **inherits** all characteristics, associations, and aggregations of the superclass except **private ones**.
- Subclass may have further characteristics, associations, and aggregations.
- The generalization relationship is also referred to as an “is a” relationship.
- Generalizations are **transitive**.

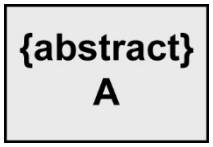


Superclass

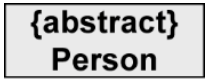
Subclasses …inherit characteristics, associations, and aggregations

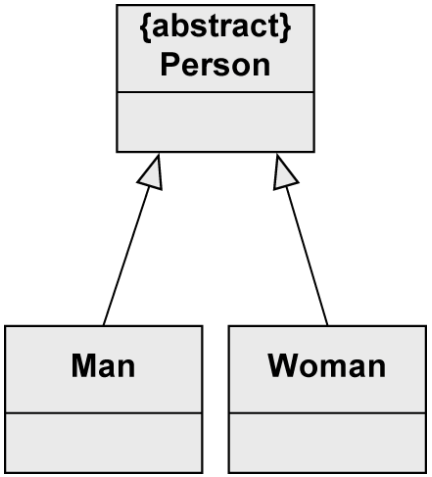
A `Secretary` is an `Employee` and a `Person`

Generalization – Abstract Class



- Used to highlight common characteristics of their subclasses.
- Used to ensure that there are no direct instances of the superclass.
- Only its non-abstract subclasses can be instantiated.
- Useful in the context of generalization relationships.
- Notation: keyword **`{abstract}`** or class name in italic font.





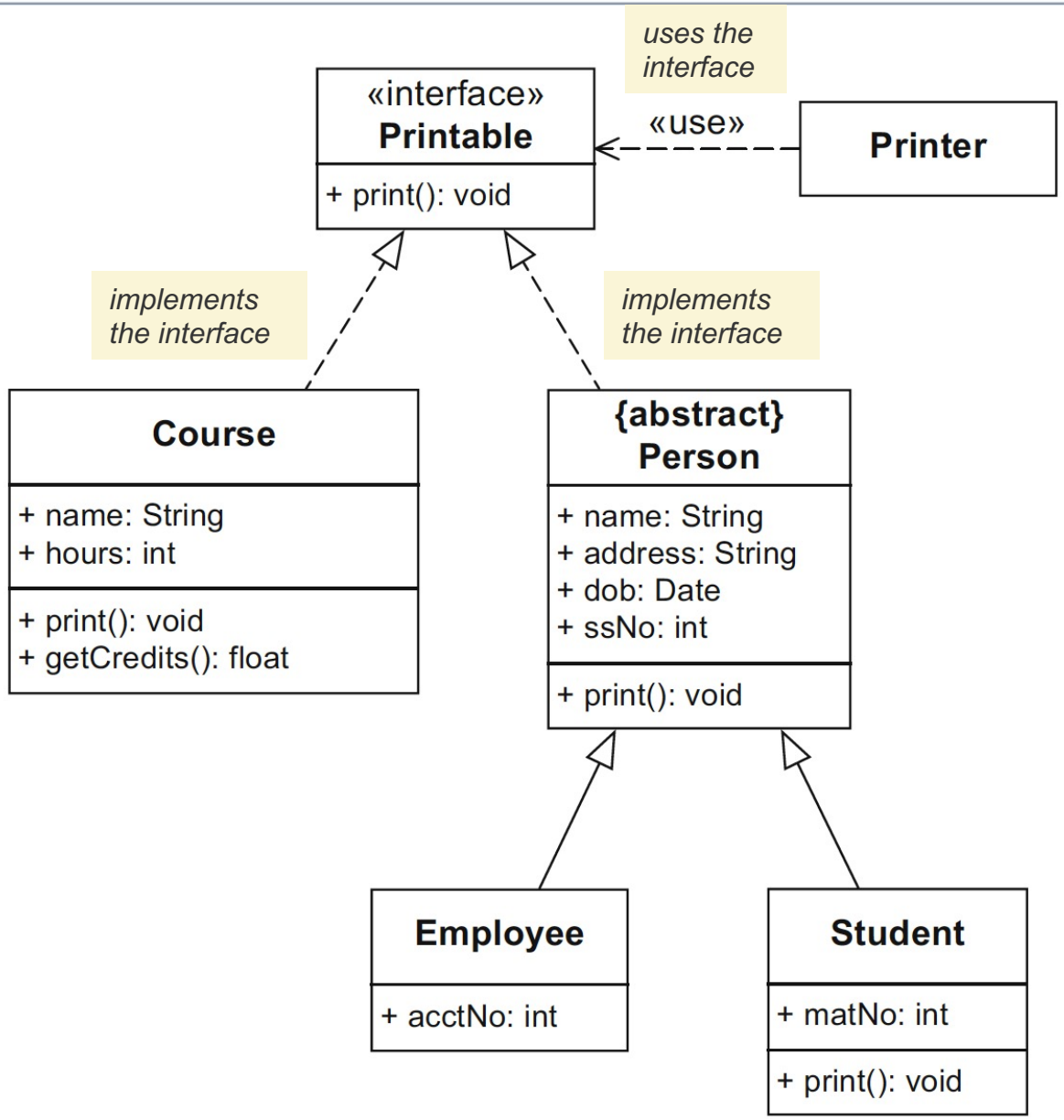
No `Person`-object possible

Two types of `Person`: `Man` and `Woman`

Generalization – Interface Class

- Similarly, to the abstract class, an interface also does not have an implementation or any direct instances. An interface represents a contract.
- The classes that enter into this contract, that is, the classes that implement the interface, obligate themselves to provide the behavior specified by the interface.
- In contrast to the relationship between an abstract class and its subclasses, an “is a” relationship between an interface and the classes that implement it is not necessary.
- Operations of interfaces never have an implementation.
- An interface is denoted like a class but with the additional keyword «interface» before the name.

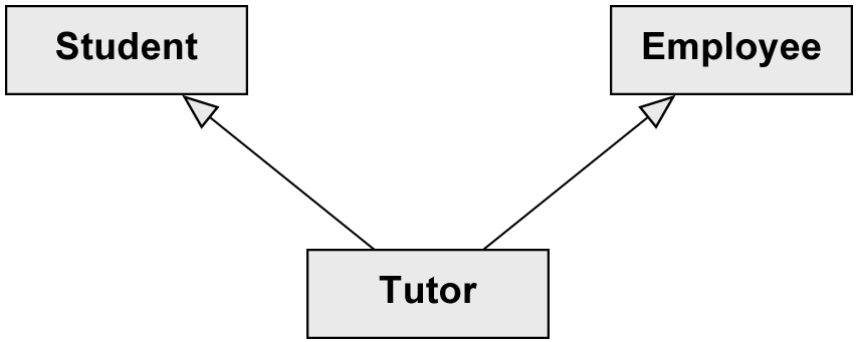
Generalization – Interface Class



Generalization – Multiple Inheritance

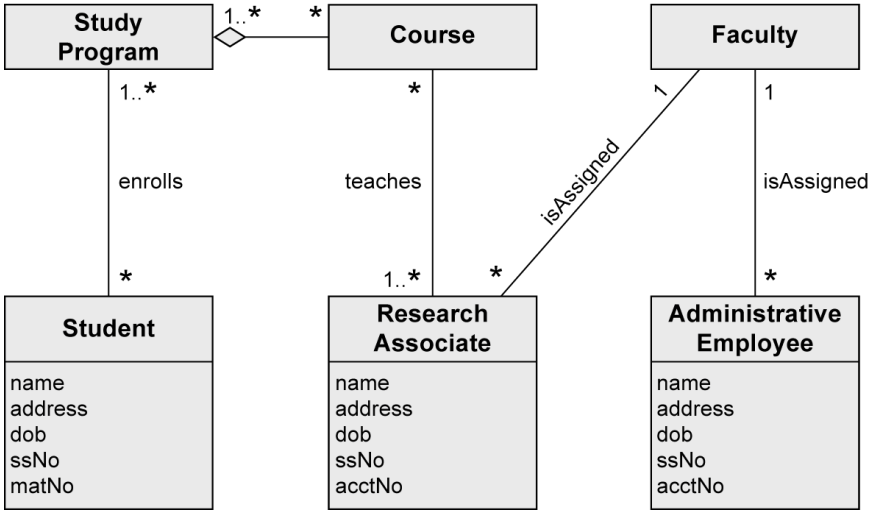
- UML allows **multiple** inheritance.
- A class may have multiple superclasses.

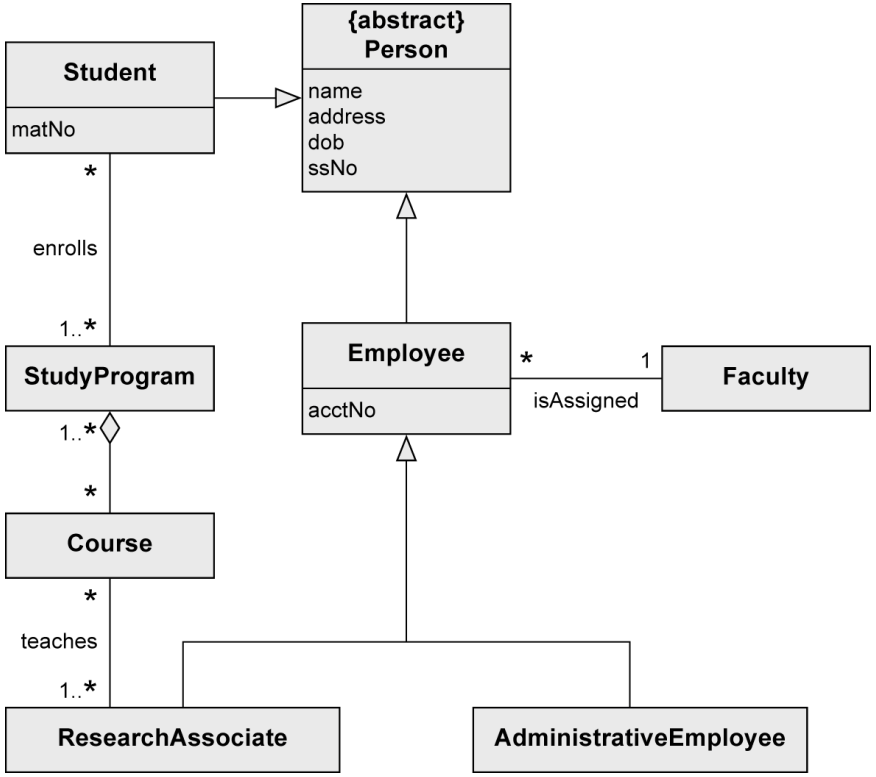
- Example:



A `Tutor` is both an `Employee` and a `Student`

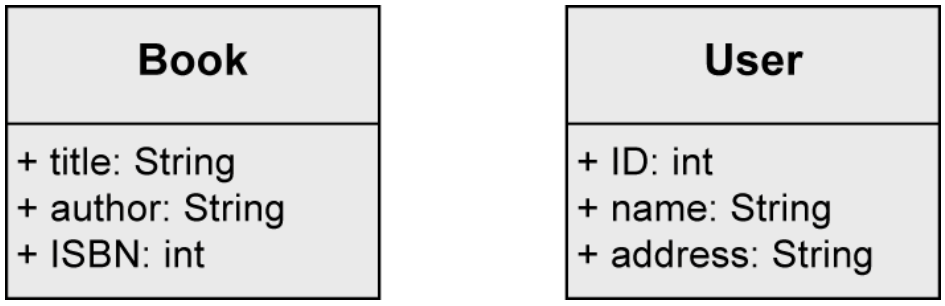
With and Without Generalization





Creating a Class Diagram

- Not possible to completely extract classes, attributes and associations from a natural language text automatically.
- Guidelines
  - **Nouns** often indicate classes
  - **Adjectives** indicate attribute values
  - **Verbs** indicate operations
- 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.



Creating a Class Diagram

- The following three aspects are important:
  - Which operations can an object of a class execute?
  - Which events, to which the object must be able to react, can theoretically occur?
  - Which other events occur as a result? If the values of an attribute can be derived from another attribute, for example, if the age of a person can be calculated from their date of birth, it should be identified as a derived attribute.
- Further, it is essential to consider not only the current requirements but also the extensibility of the system.

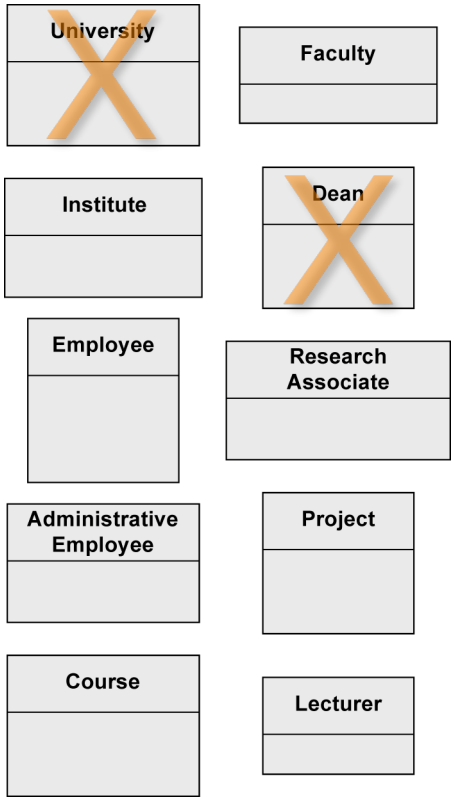
Example – University Information System

- 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.

Example – 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.

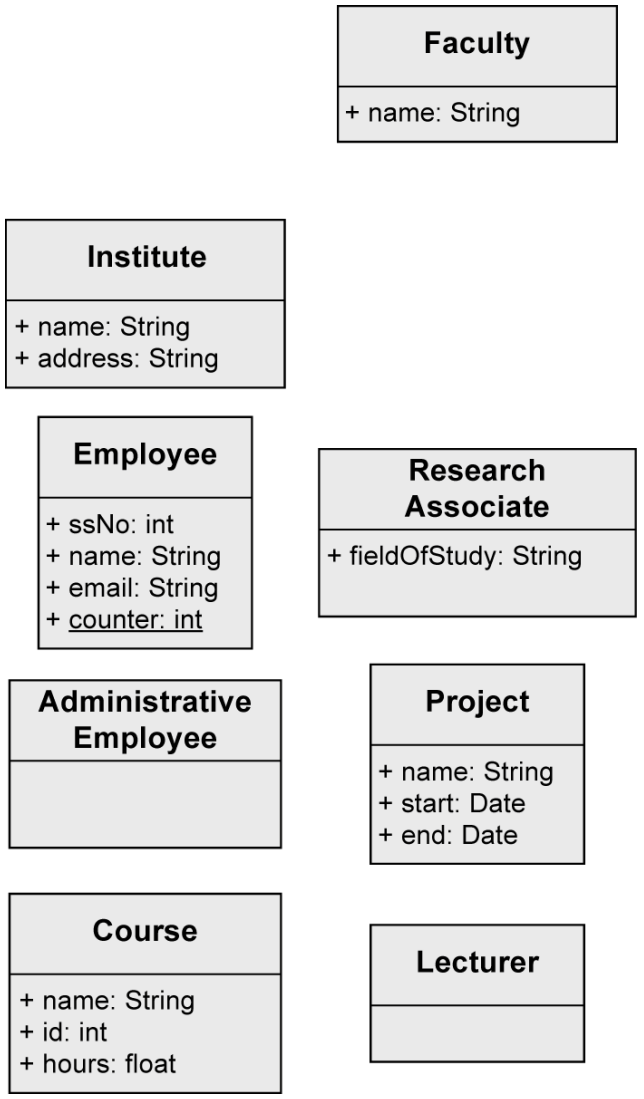
We model the system „University“



Dean has no further attributes than any other employee

Example – Step 2: Identifying the Attributes

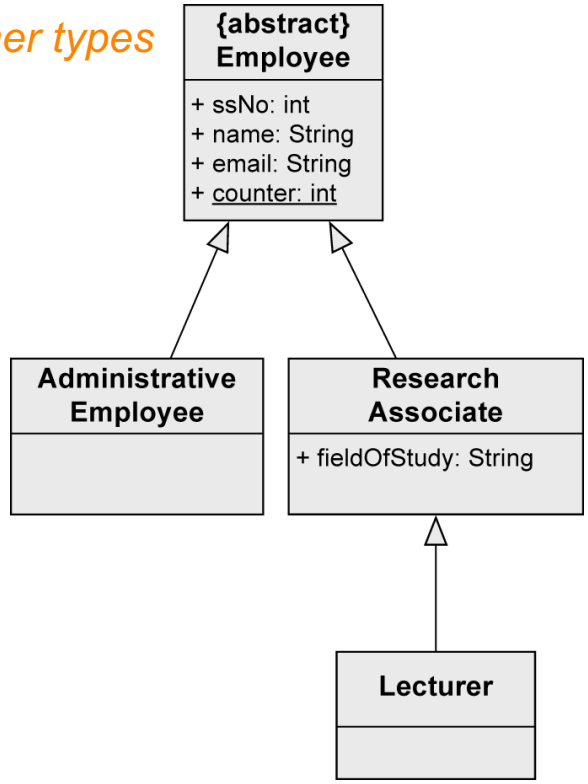
- A university consists of multiple faculties which are composed of various institutes. Each faculty and each institute has a name. An address is known for each institute.
- Each faculty is led by a dean, who is an employee of the university.
- The total number of employees is known. Employees have a 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.



Example – Step 2: Identifying Relationships (1/6)

- Three kinds of relationships:
  - Association
  - Generalization
  - Aggregation

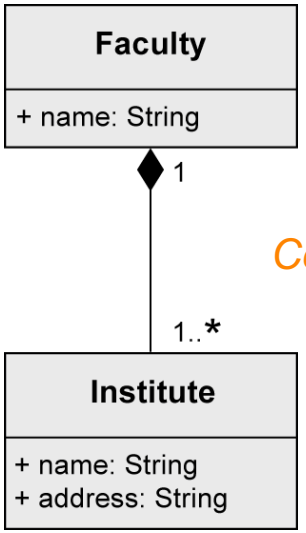
Abstract, i.e., no other types of employees



- Indication of a **generalization**
- *"There is a distinction between research and administrative personnel."*
- *"Some research associates hold courses. Then they are called lecturers."*

Example – Step 2: Identifying Relationships (2/6)

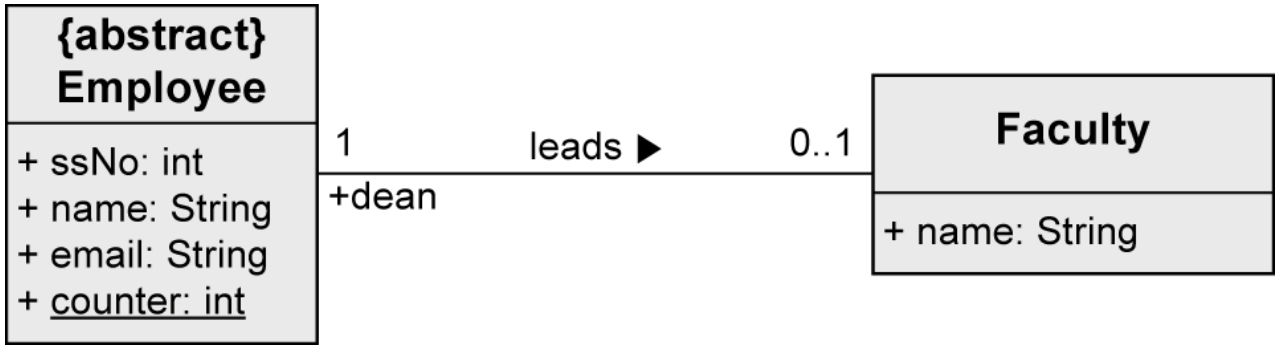
- *"A university consists of **multiple faculties** which are composed of **various institutes**."*



Composition to show existence dependency

Example – Step 2: Identifying Relationships (3/6)

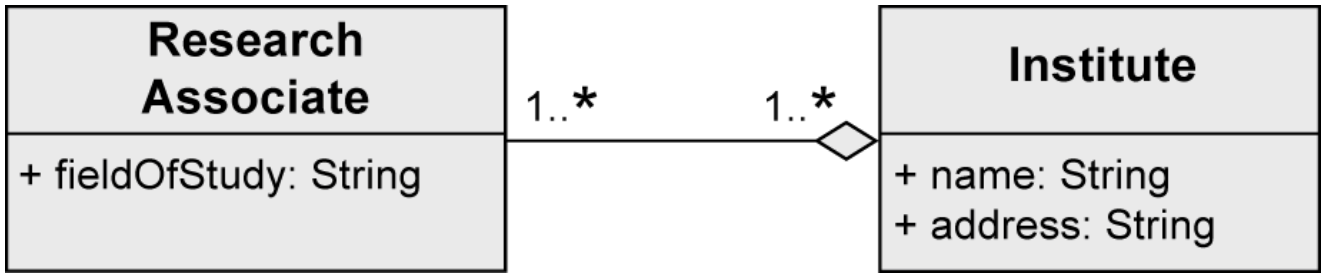
- *"Each **faculty** is led by a **dean**, who is an **employee** of the university"*



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

Example – Step 2: Identifying Relationships (4/6)

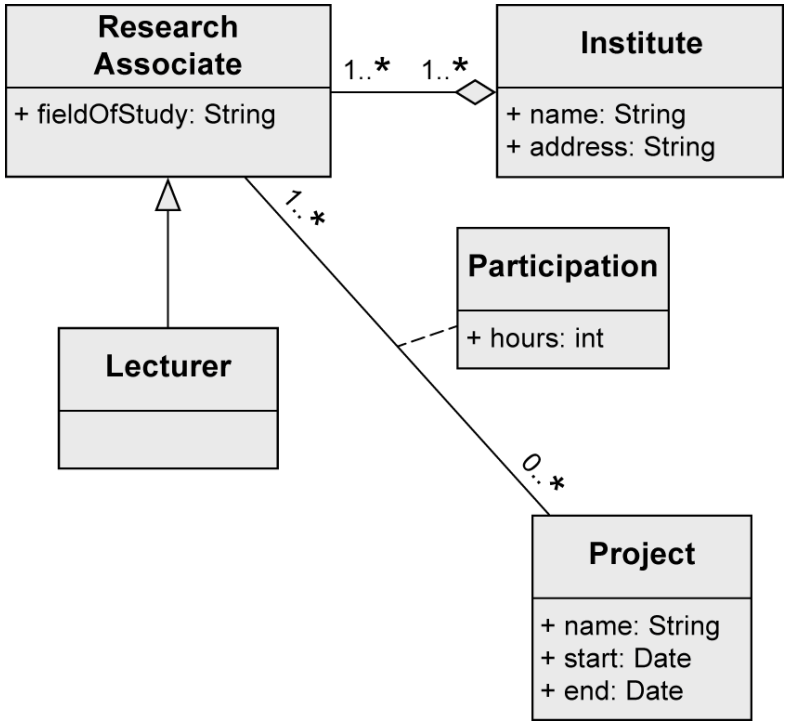
- "**Research associates** are assigned to at least **one institute**."



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

Example – Step 2: Identifying Relationships (5/6)

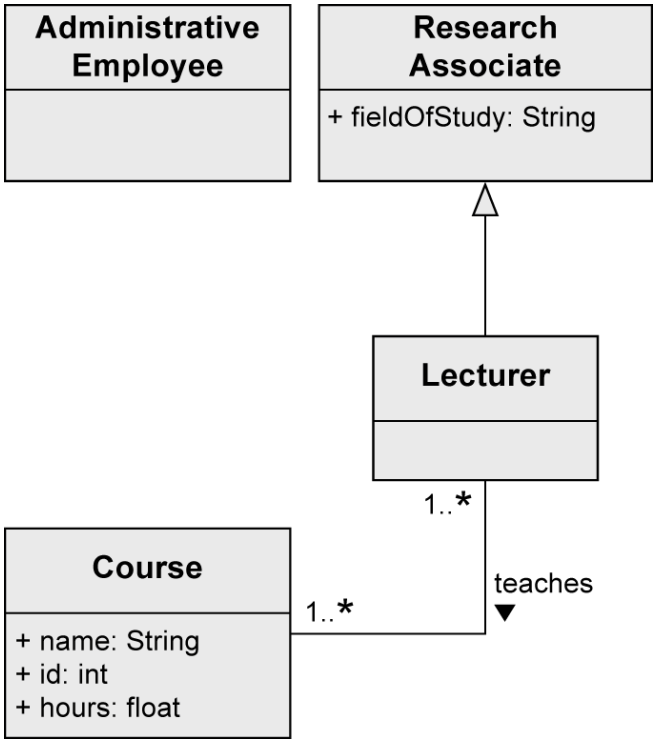
- *"Furthermore, **research associates** can be involved in projects for a certain number of hours."*



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

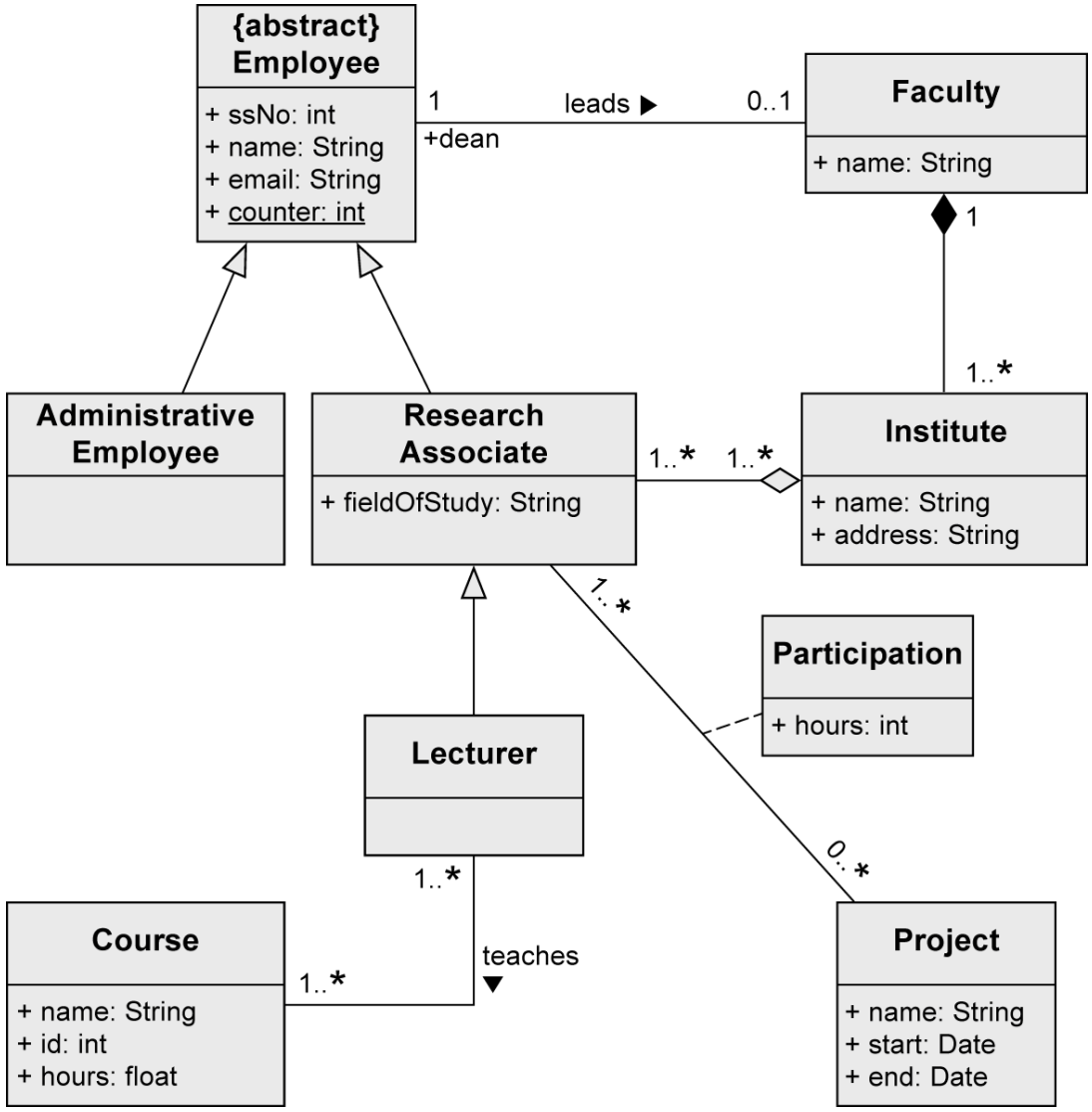
Example – Step 2: Identifying Relationships (6/6)

- *"Some research associates hold courses. Then they are called lecturers."*



`Lecturer` inherits all characteristics, associations, and aggregations from `ResearchAssociate`.
In addtion, a `Lecturer` has an association `teaches` to `Course`.

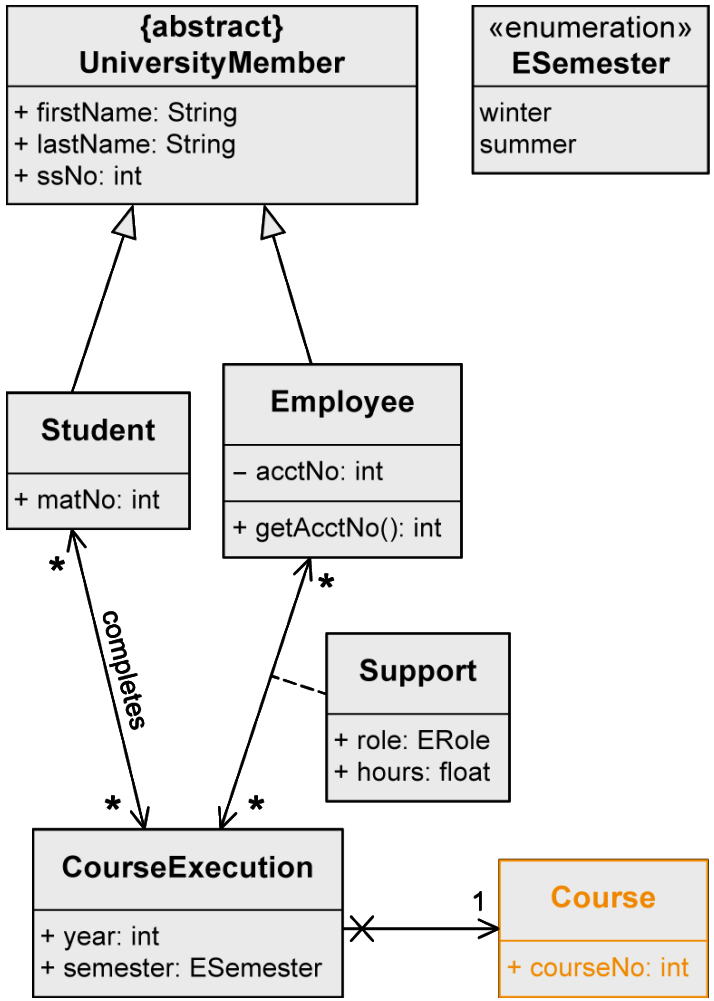
Example – Complete Class Diagram



Code Generation

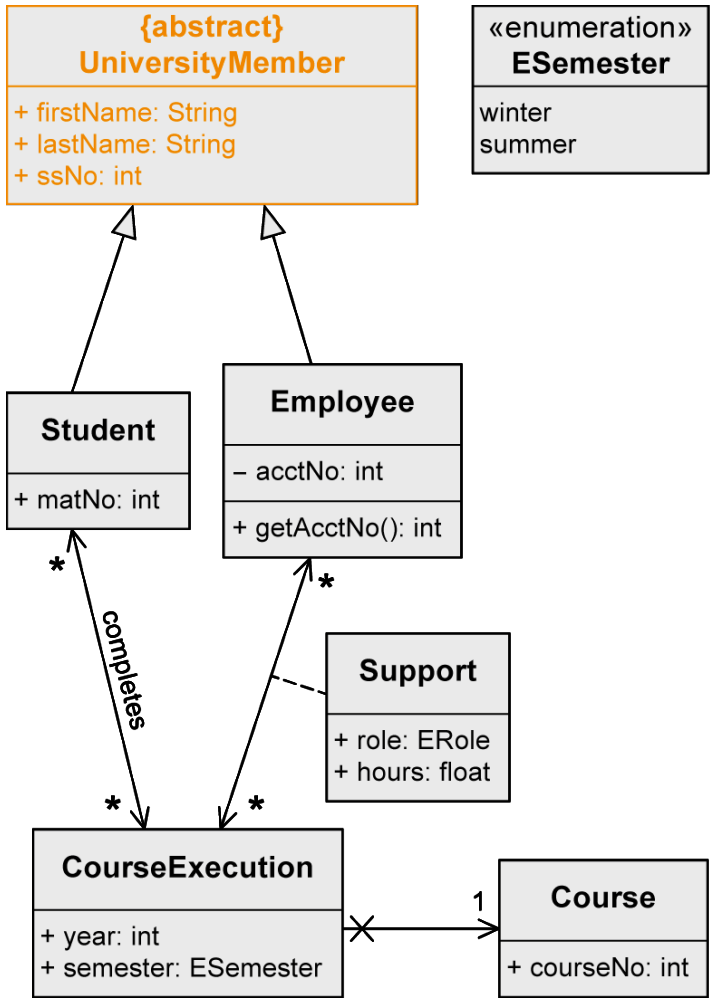
- Class diagrams are often created with the intention of implementing the modeled elements in an object-oriented programming language.
- Often, translation is semi-automatic and requires only minimal manual intervention.
- The class diagram is also suitable for documenting existing program code, with the advantage that the relationships between classes are represented graphically.
- There are a number of tools for reverse engineering program code into class diagrams automatically.

Code Generation – Example (1/6)



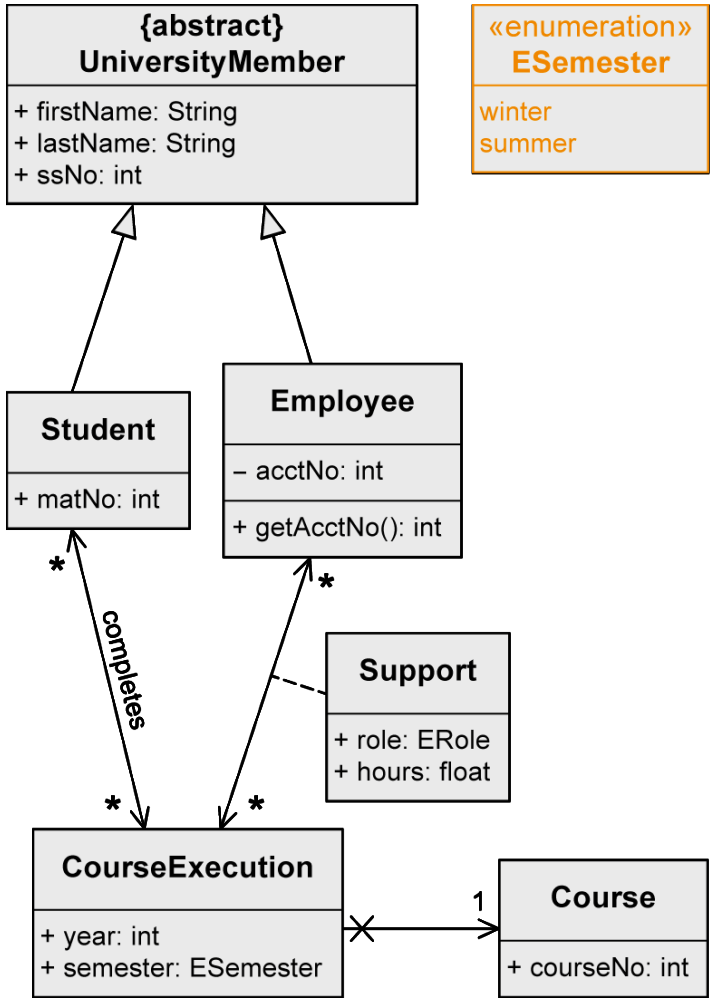

```
class Course {
    public int courseNo;
}
```

Code Generation – Example (2/6)




```
abstract class UniversityMember {
   public String firstName;
   public String lastName;
   public int ssNo;
}
```

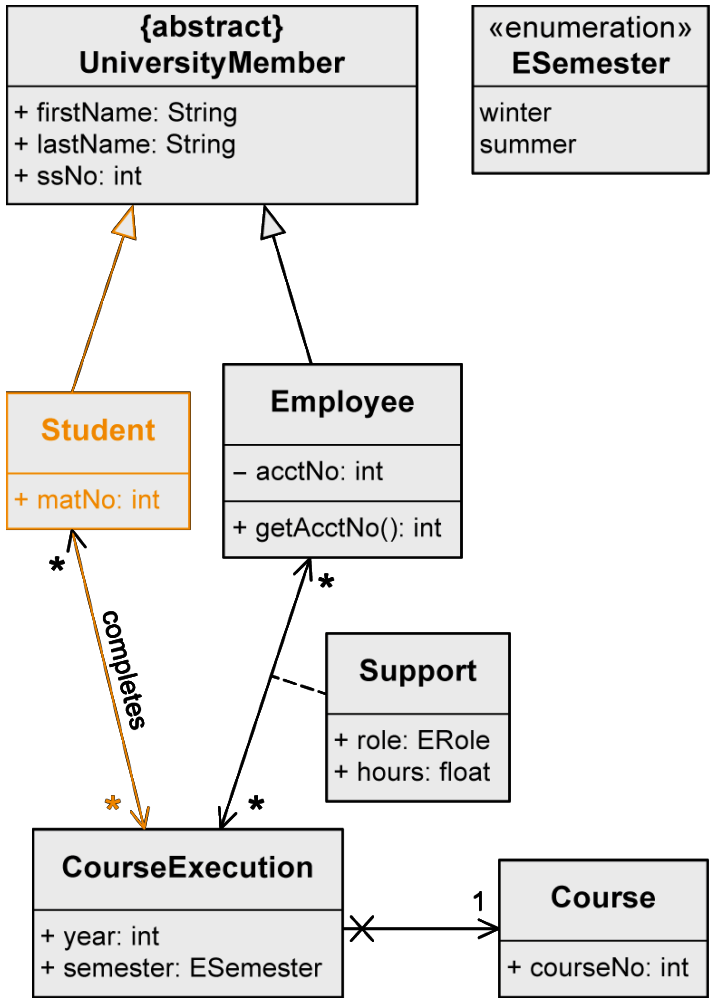
Code Generation – Example (3/6)




```
Enumeration ESemester {
    winter,
    summer
}
```

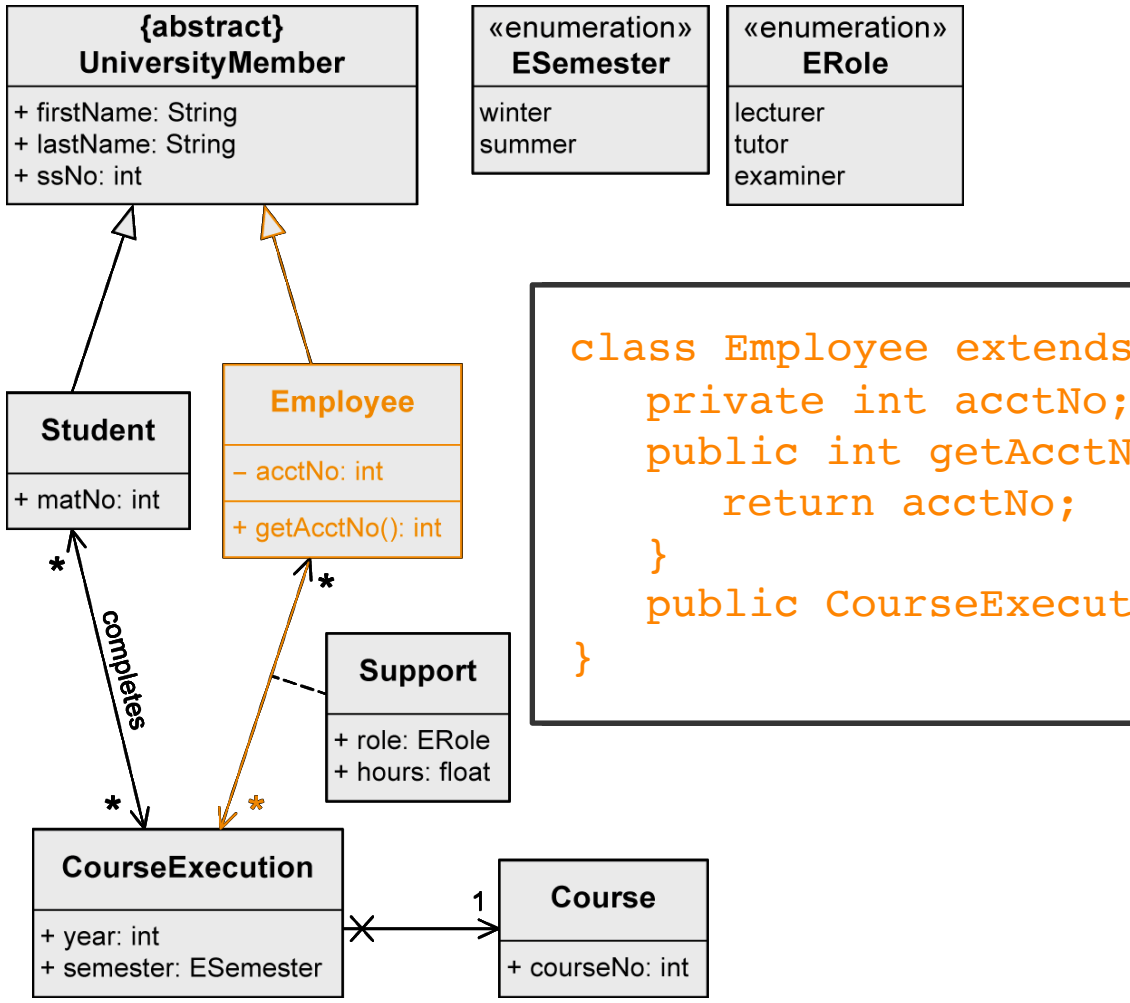
```
Enumeration ERole {
    lecturer,
    tutor,
    examiner
}
```

Code Generation – Example (4/6)



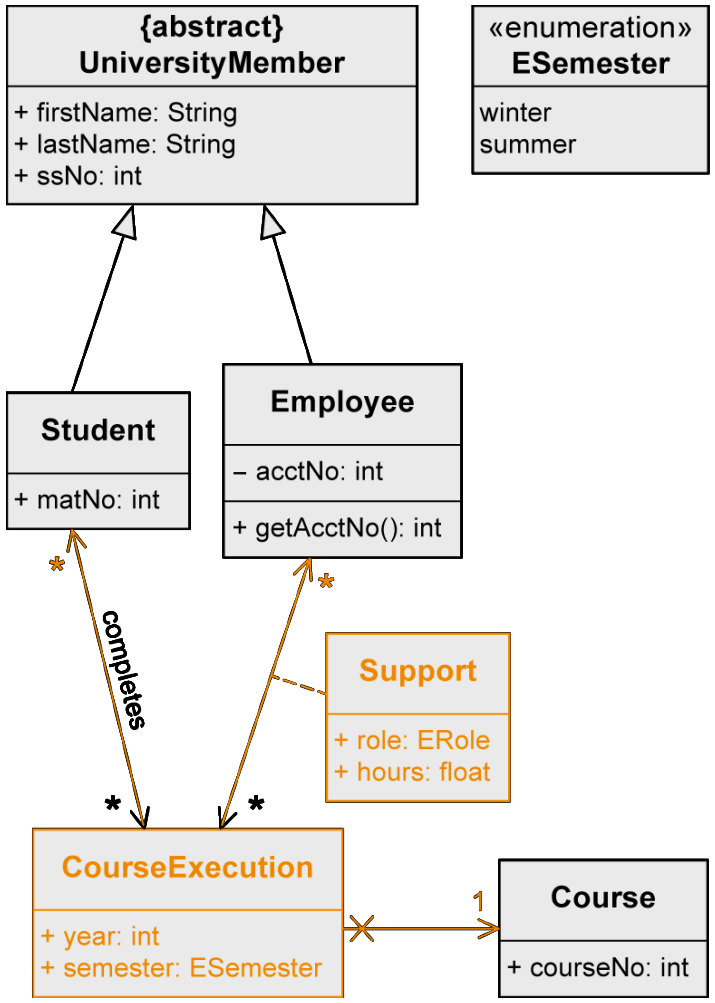

```
class Student extends
                UniversityMember {
   public int matNo;
   public CourseExecution []
                completedCourses;
}
```

Code Generation – Example (5/6)

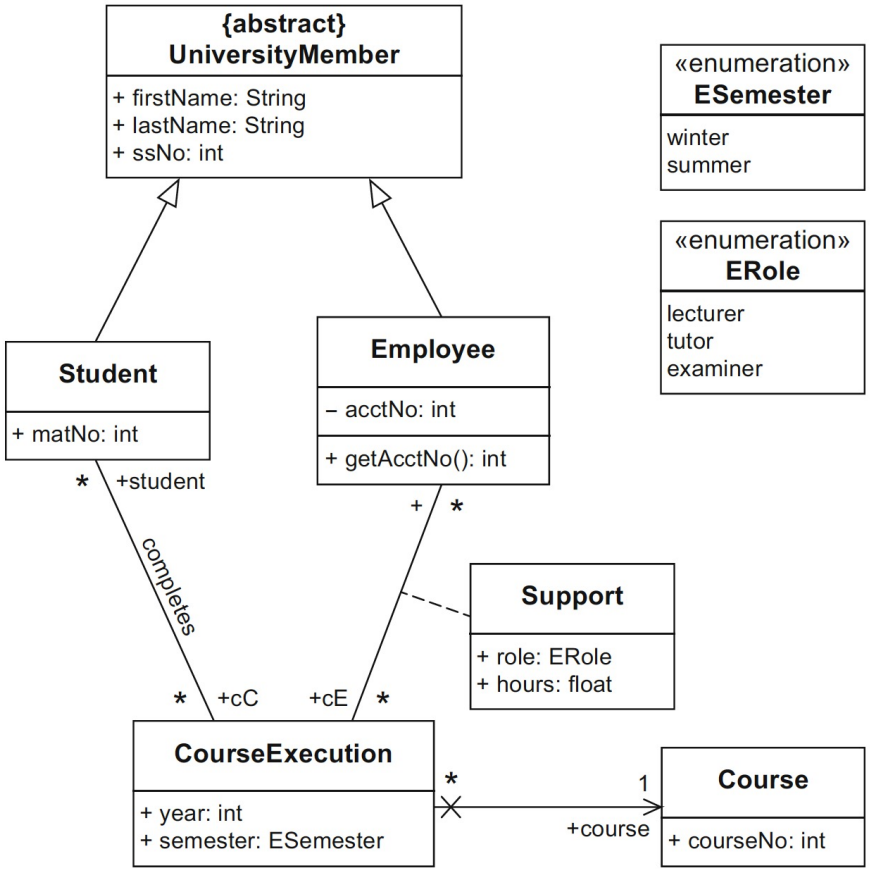



```
class Employee extends UniversityMember {
   private int acctNo;
   public int getAcctNo () {
      return acctNo;
   }
   public CourseExecution [] courseExecutions;
}
```

Code Generation – Example (6/6)

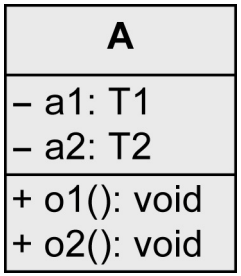
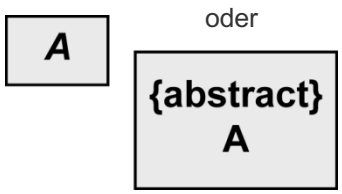
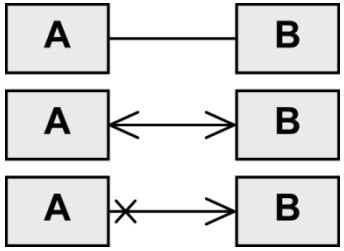



```
class CourseExecution {
   public int year;
   public ESemester semester;
   public Student [] student;
   public Course course;
   public Hashtable support;
      // Key: employee
      // Value: (role, hours)
}
```

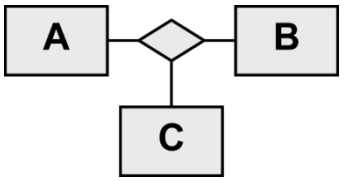
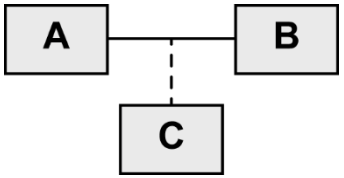
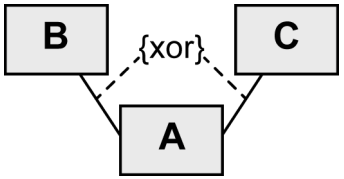



```
abstract class UniversityMember {
        public String firstName;
        public String lastName;
        public int ssNo;
}

class Student extends UniversityMember {
        public int matNo;
        public CourseExecution [] cC; // completed c.
}

class Employee extends UniversityMember {
        private int acctNo;
        public CourseExecution [] cE; // supported c.
        public int getAcctNo { return acctNo; }
}
class CourseExecution {
        public int year;
        public ESemester semester;
        public Student [] student;
        public Course course;
        public Hashtable support;
                // Key: employee
                // Value: (role, hours)
}

class Course {
        public int courseNo;
}

Enumeration ESemester {
        winter;
        summer;
}

Enumeration ERole {
        lecturer;
        tutor;
        examiner;
}
```

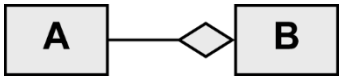
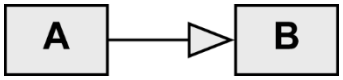
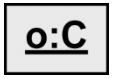
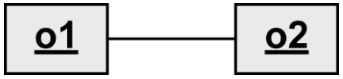
Notation Elements (1/3)

Name	Notation	Description
Class		Description of the structure and behavior of a set of objects
Abstract class		Class that cannot be instantiated
Association		Relationship between classes: navigability unspecified, navigable in both directions, not navigable in one direction

Notation Elements (2/3)

Name	Notation	Description
n-ary association		Relationship between n (here 3) classes
Association class		More detailed description of an association
`xor` relationship		An object of **`C`** is in a relationship with an object of **`A`** or with an object of **`B`** but not with both

Notation Elements (3/3)

Name	Notation	Description
Shared aggregation		Parts-whole relationship (**A** is part of **B**)
Strong aggregation = composition		Existence-dependent parts-whole relationship (**A** is part of **B**)
Generalization		Inheritance relationship (**A** inherits from **B**)
Object		Instance of a class
Link		Relationship between objects