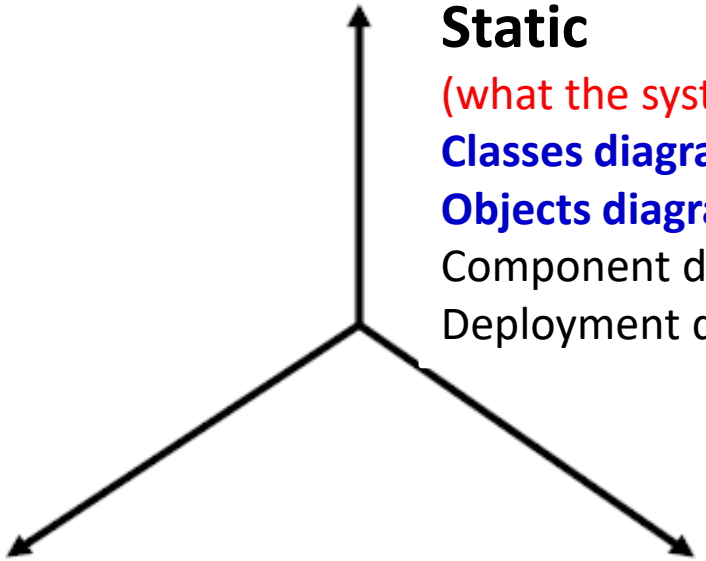
Software Engineering Course

Chapter 4
UML Class and Object Diagrams: Static View
(Class diagram, Object Diagram)

2022-2023

Dr. Sofia Kouah

✓Three modeling axes:



In this chapter, we are interested in:

Class and Object Diagrams

Class diagram

- ✓Representation of the **internal structure** of software
- ✓ The **predominantly utilized** UML diagram.
- ✓Can be applied **in numerous stages of the software development process (**Mainly used in **design** but can be used in **analysis and implementation phases)**.
- ✓Is a **structure diagram** which **represents the structure** of the designed system **at the level** of classes and interfaces,
- ✓It shows their features, relationships and constraints, such as : generalizations, associations, dependencies, etc.

Class and Object Diagrams

Object diagram

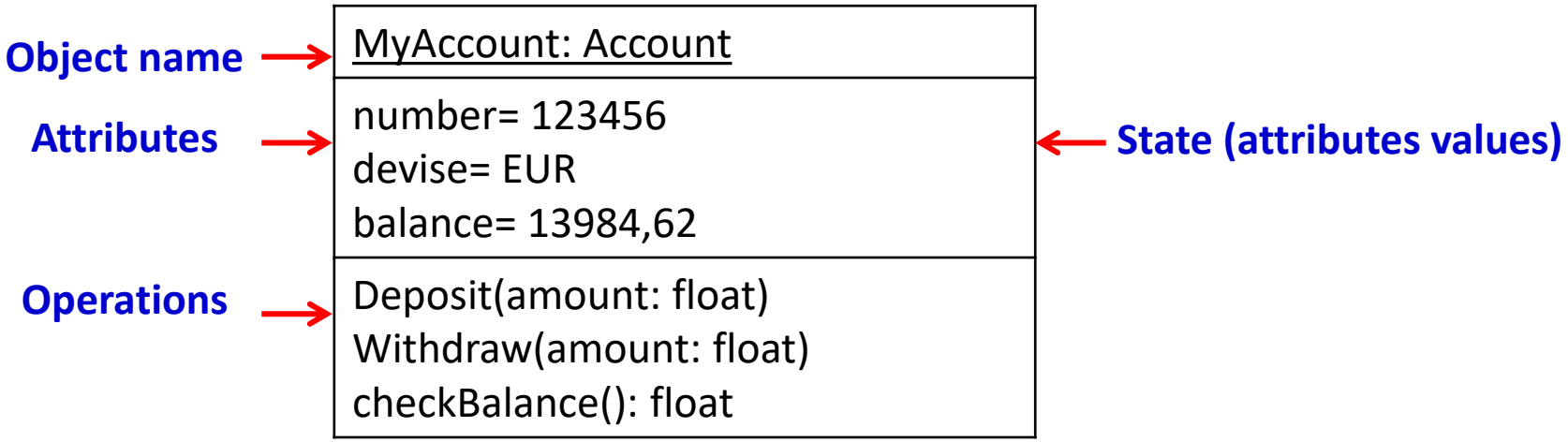
- ✓ Representation of the **software state** (Object + Relations)
- ✓ Allows **depicting concrete objects** that appear in a system at a **specific point in time**.
- ✓ Diagram **evolving with the execution** of the software
  - ✓ **Creating** and **deleting objects**
  - ✓ **Modification** of the **state of objects** (values of attributes)
  - ✓ **Editing relationships** between **objects**
- ✓ Can be considered **as instance level class diagram.**
- ✓ It shows instance specifications of classes and interfaces (objects), properties with assigned values, and links (instances of association).
- ✓ **It Visualizes classes instances** that are modeled in a class diagram.

Object

✓**Abstract** or **concrete entity** of the application domain.

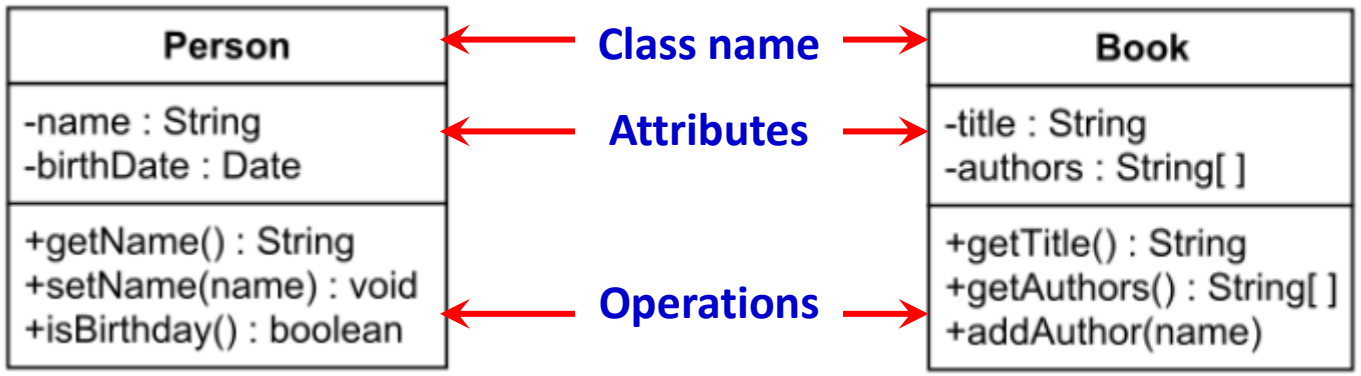
✓Described by the following:

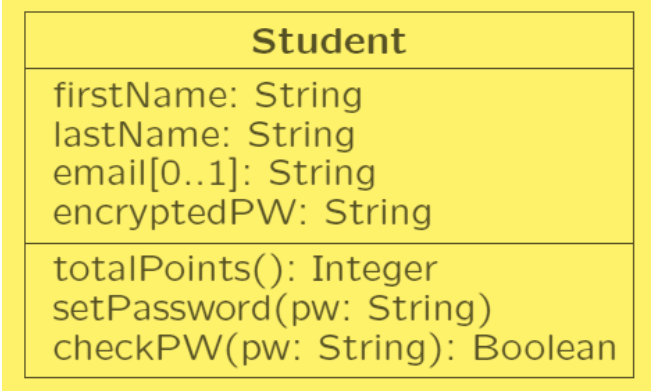
Identity (memory address) + **state** (attributes) + **behavior** (operations)



Class

- ✓A class is a **collection of objects** of the same nature.
- ✓Same nature = **same attributes** + **same operations**.
- ✓A class is **represented by a rectangle**, divided into three parts:
  - ✓The class **name**.
  - ✓The **attributes names and types (Fields)**.
  - ✓The **names, return types, and parameters** of the **operations**.
- ✓**Examples:**



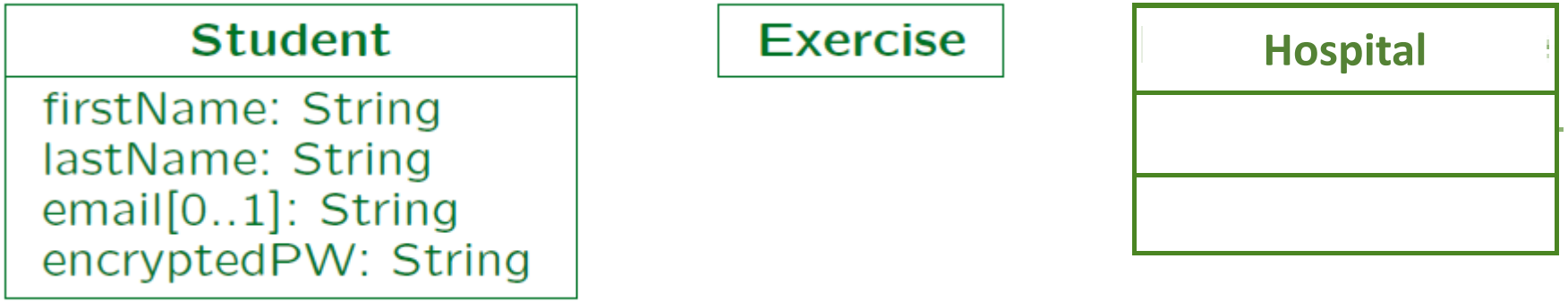


Class

✓**Either** or **both** of **attributes and operations parts** may be deleted.

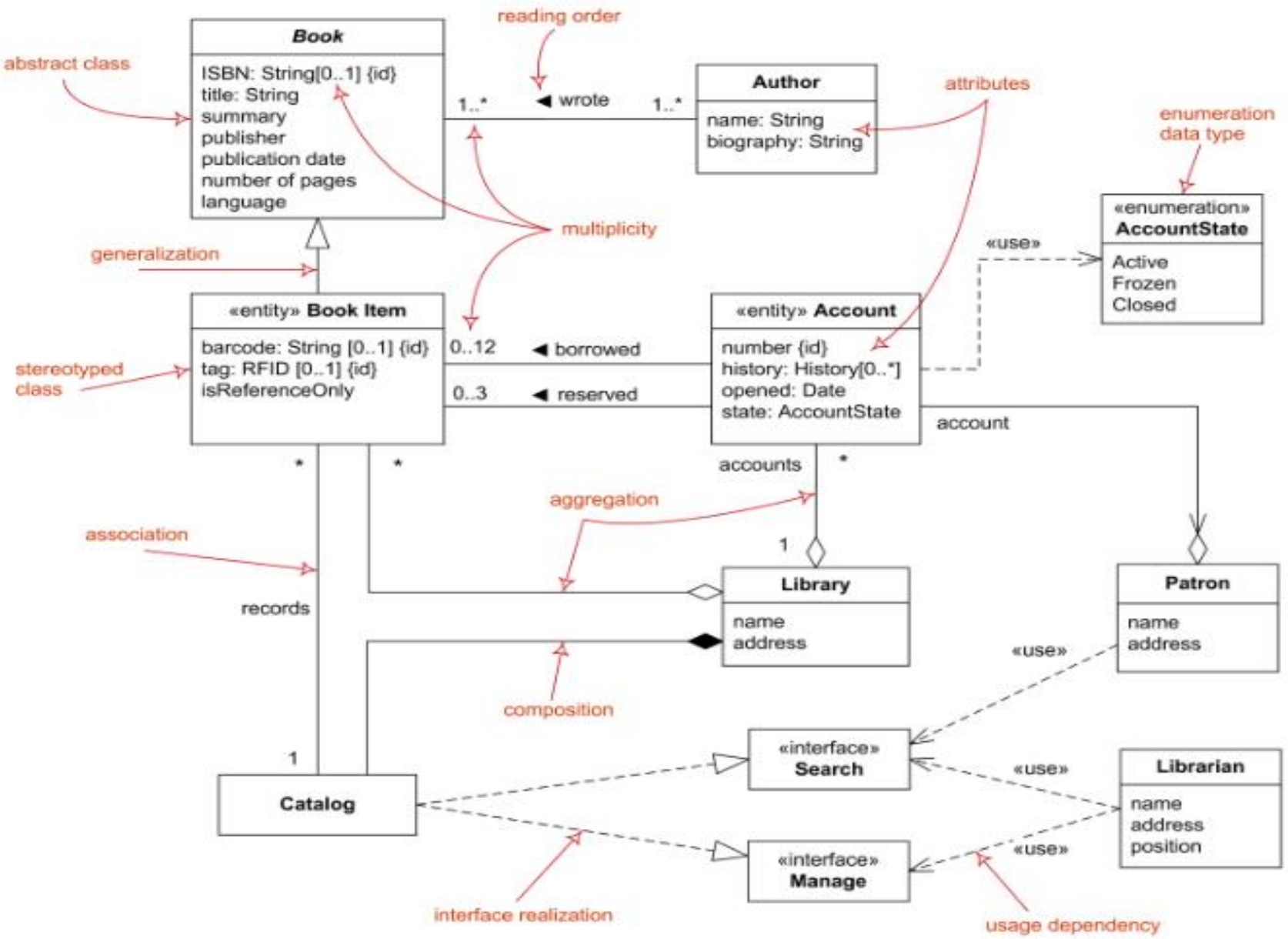
✓It is possible to show **only attributes**, **only operations**, or **none of the two**.

✓**Example**:



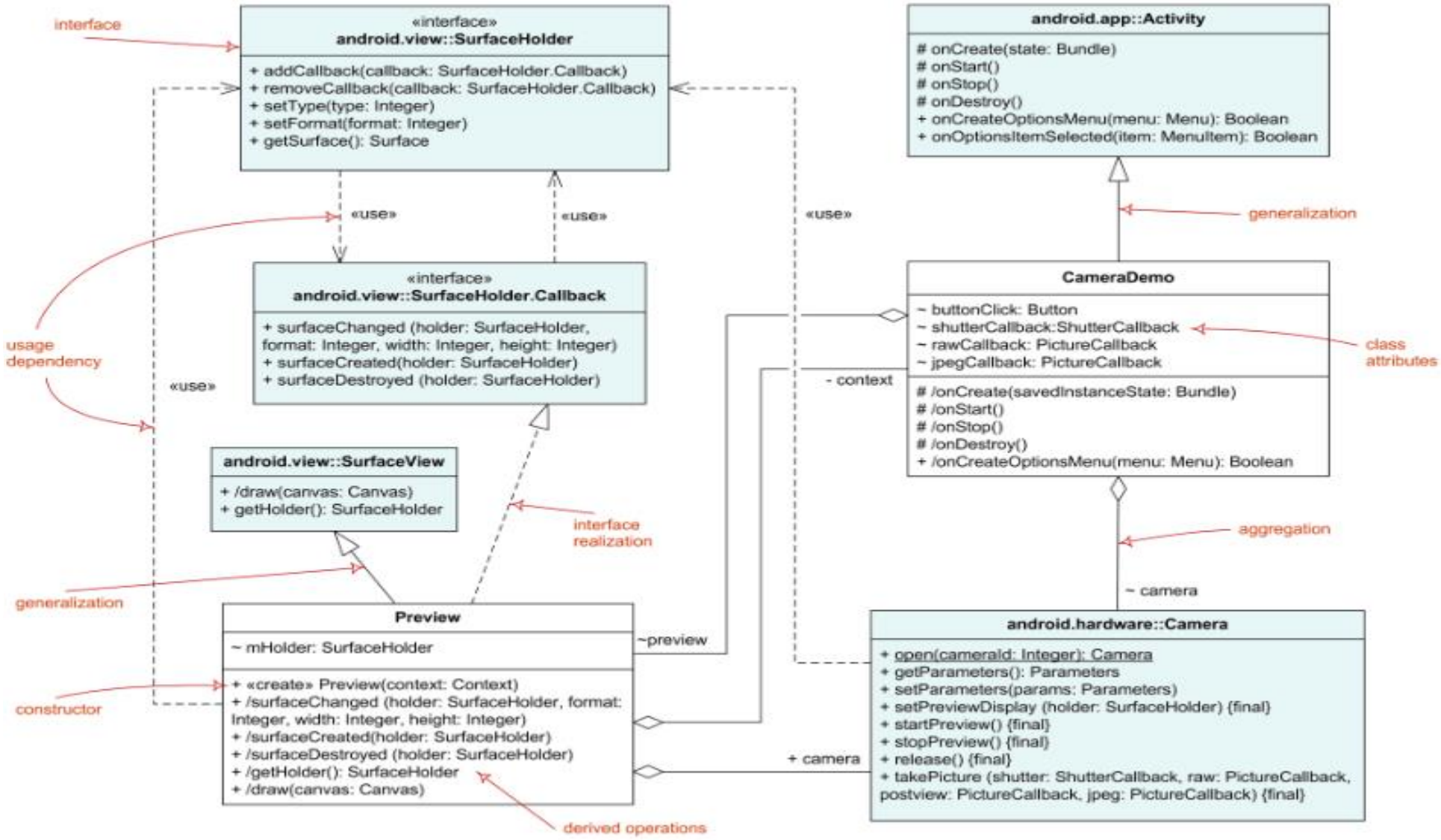
Example 1 of Class Diagram

This figure presents an example of UML class diagram for domain modeling, with the main elements of the diagram.

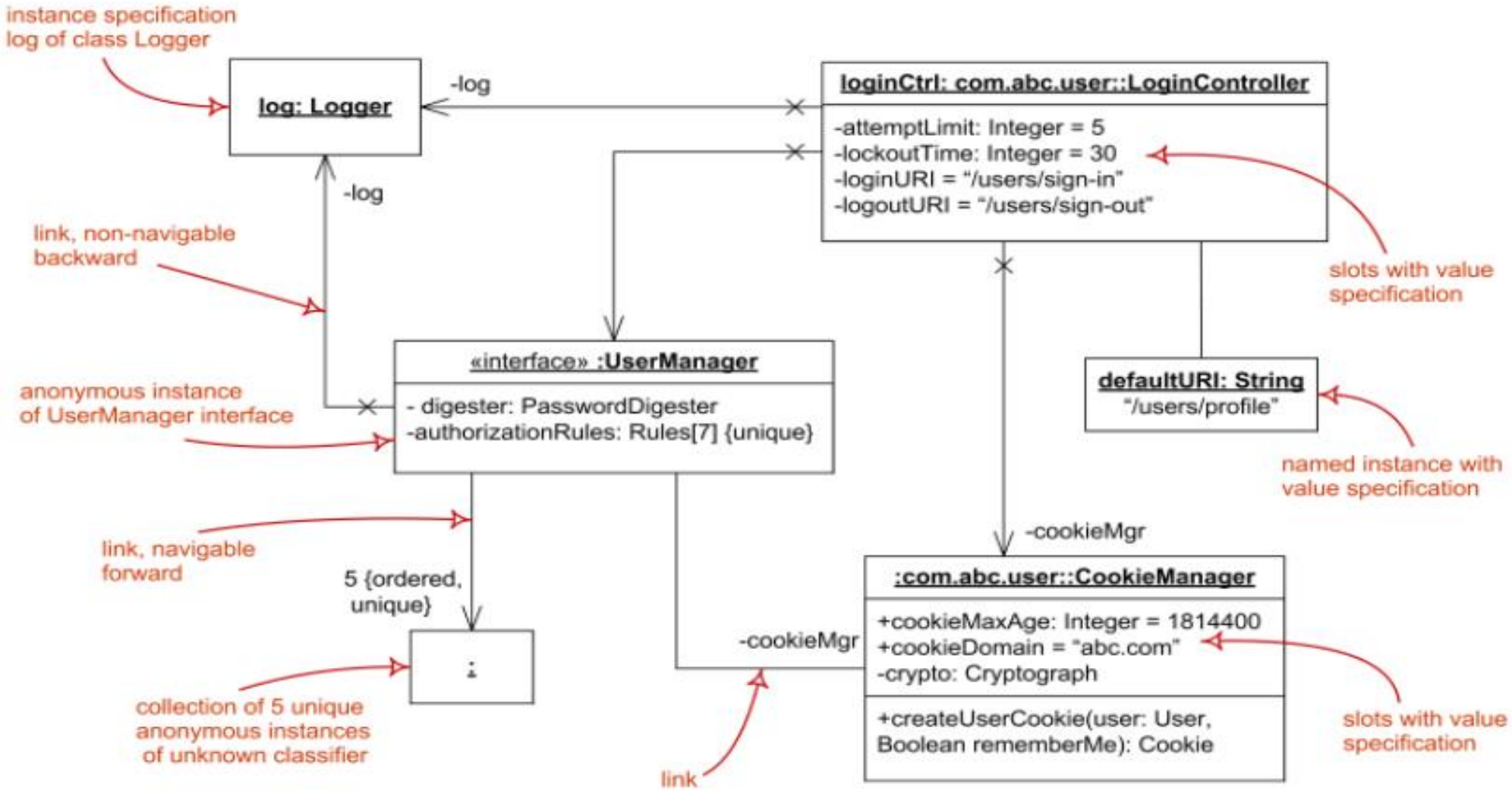


Example 2 of Class Diagram

This figure presents an example of UML class diagram for Implementation Classes, with the main elements of the diagram.



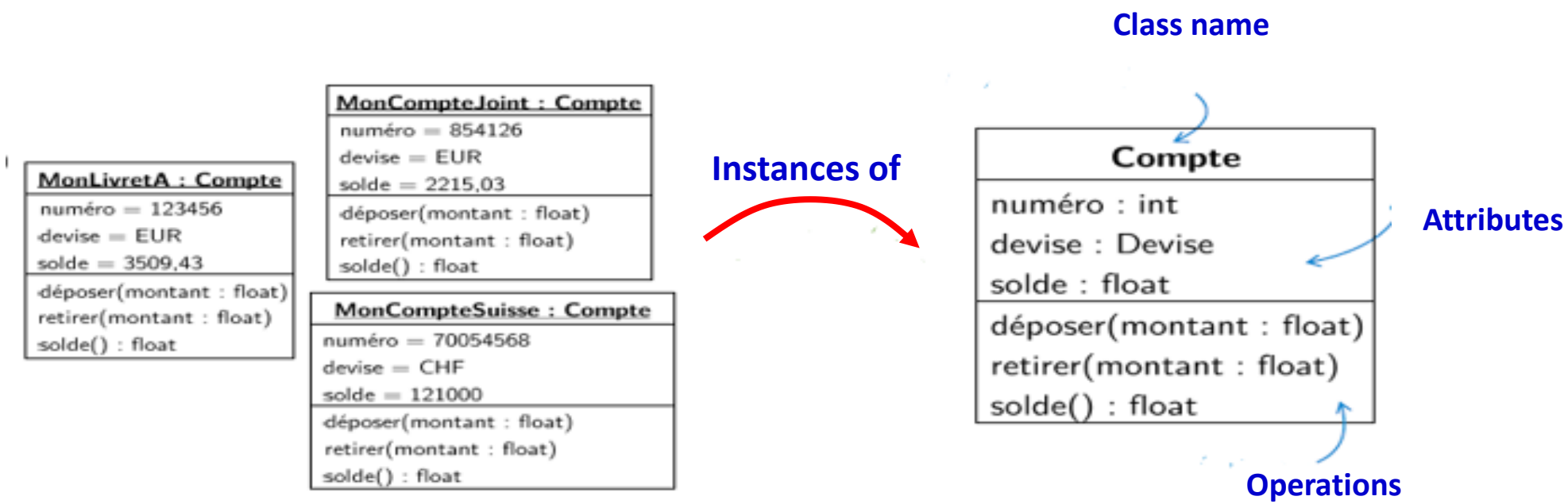
Example of Object Diagram



This figure presents an example of UML object diagram, with the main elements of the diagram.

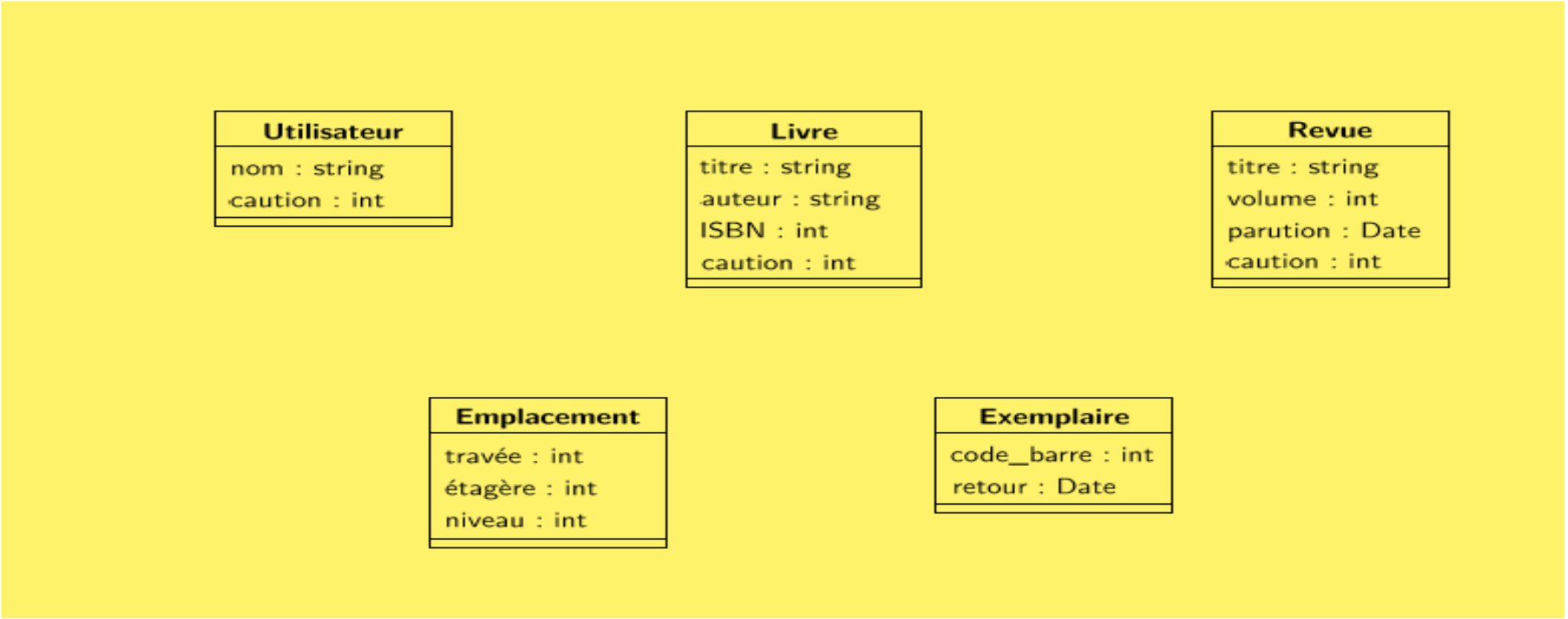
Object & Class

✓ An **object** is an instance of a **class**.

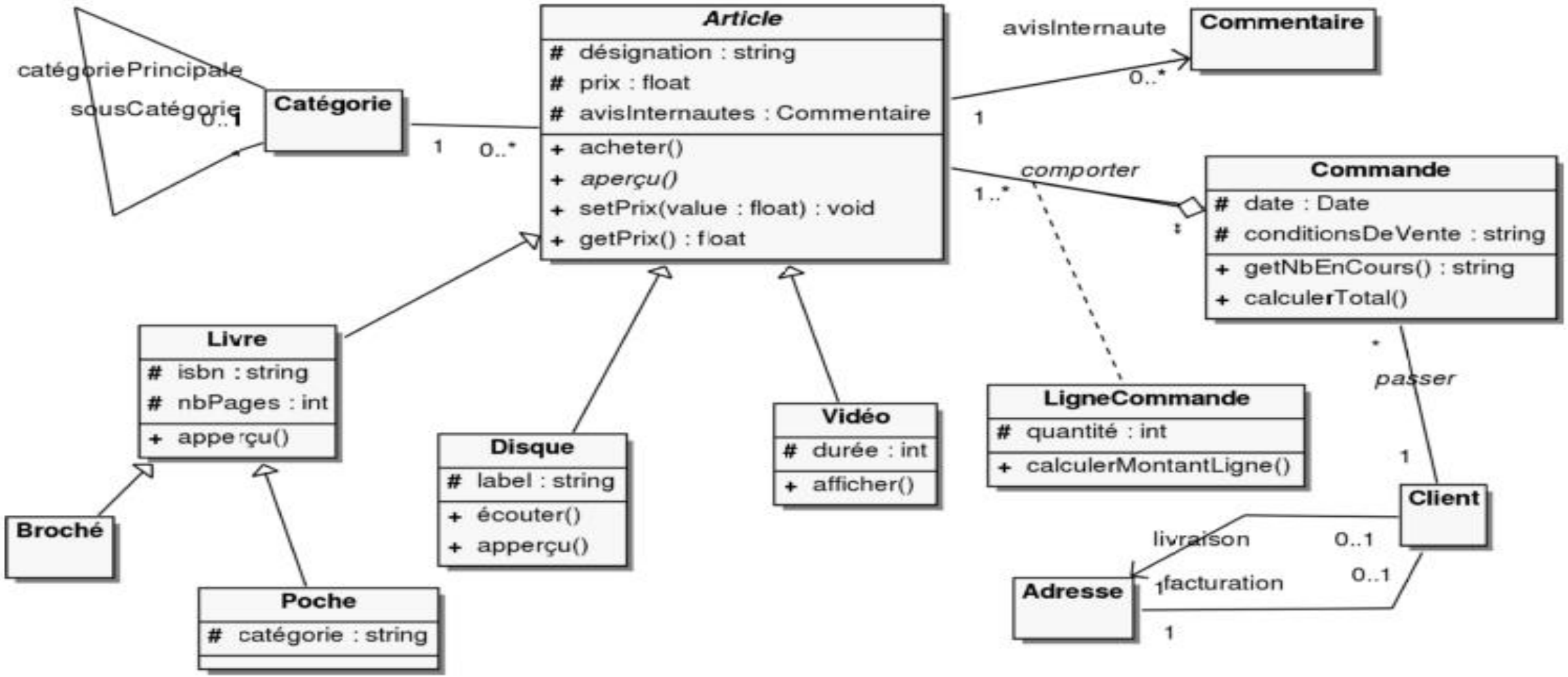


Class

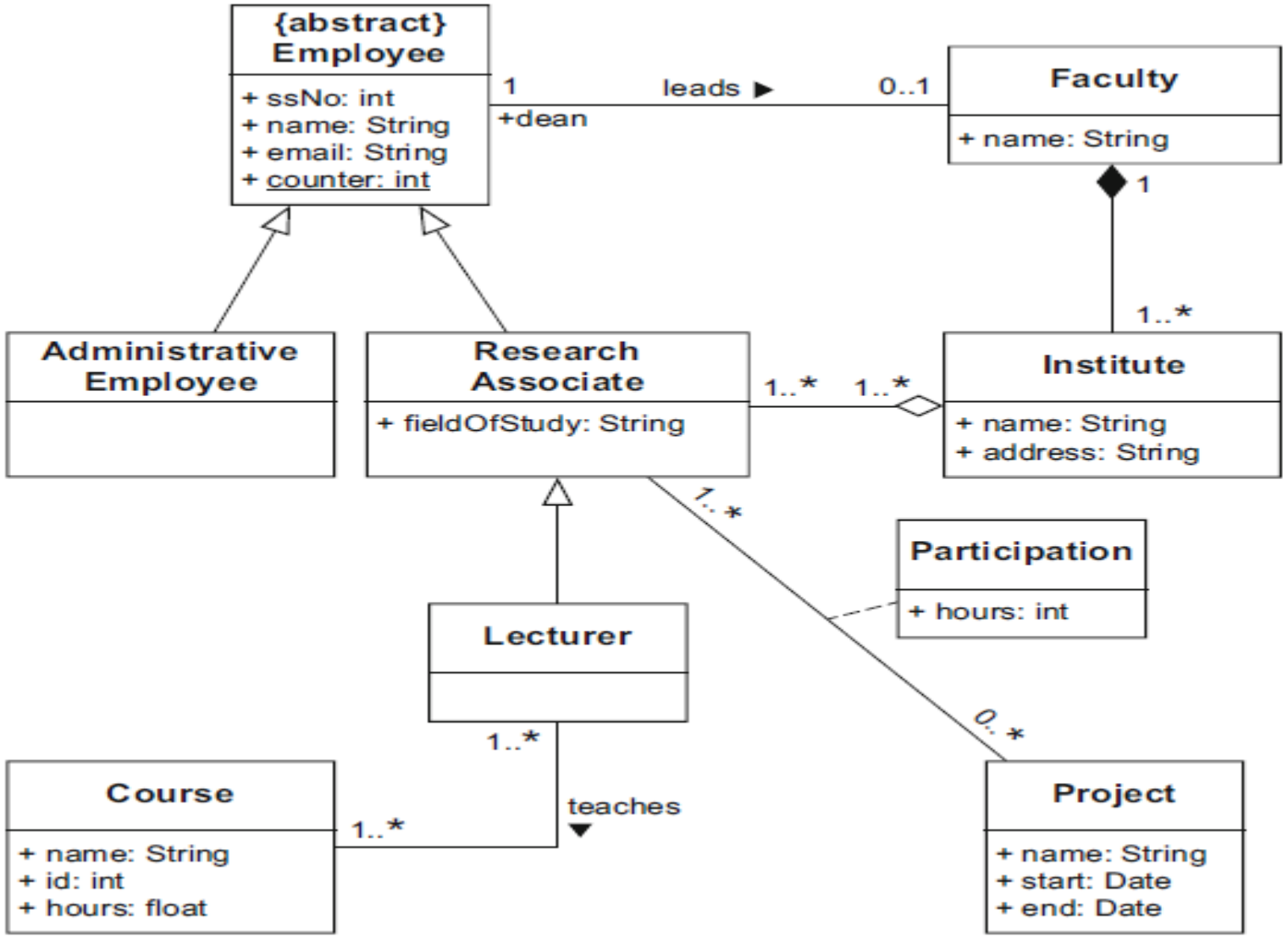
✓**Some other class examples**



Example of **Classes** Diagram



Example of Classes Diagram



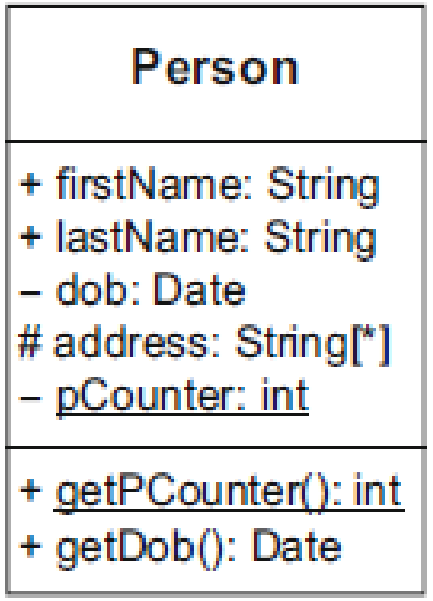
Property

- ✓ **Property = Attribute + Operations**
- ✓ **Attributes** and **operations** are **properties** of a class.
- ✓ Their **name** begins with a lowercase letter.
- ✓ An **attribute** describes a class data.
- ✓ **Attribute types** and their **initializations** as well as **access modifiers** can be **specified in the model**.
- ✓ **Attributes** take **values** when the class is instantiated: they are sort of variables attached to objects.
- ✓ An **operation** is **a service offered** by the **class** (a process that the corresponding objects can perform).

Visibility

✓The attributes and operations visibility specifies who is permitted to access them.

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

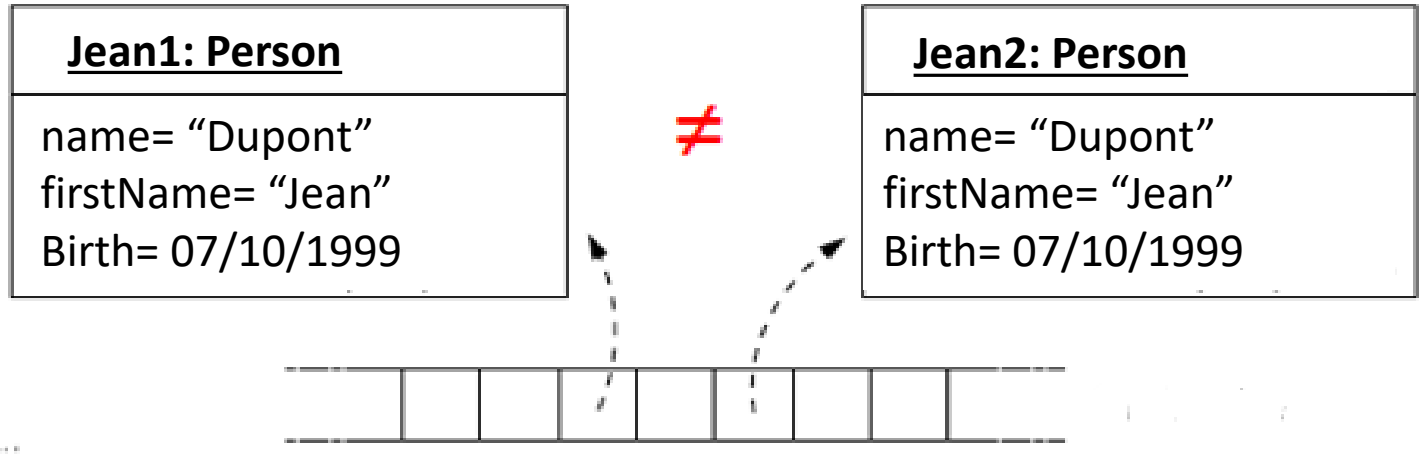


Visibility

- ✓**Private**: Only an object itself knows the values of attributes.
- ✓**Public**: Anyone can view attributes .
- ✓**Protected**: Access to protected attributes is reserved for the class itself and its subclasses.
- ✓**Package**: If a class has a package attribute, only classes that are in the same package as this class may access this attribute.
- ✓**Visibility of an operation:** specifies who is allowed to use the functionality of the operation.

Attributes

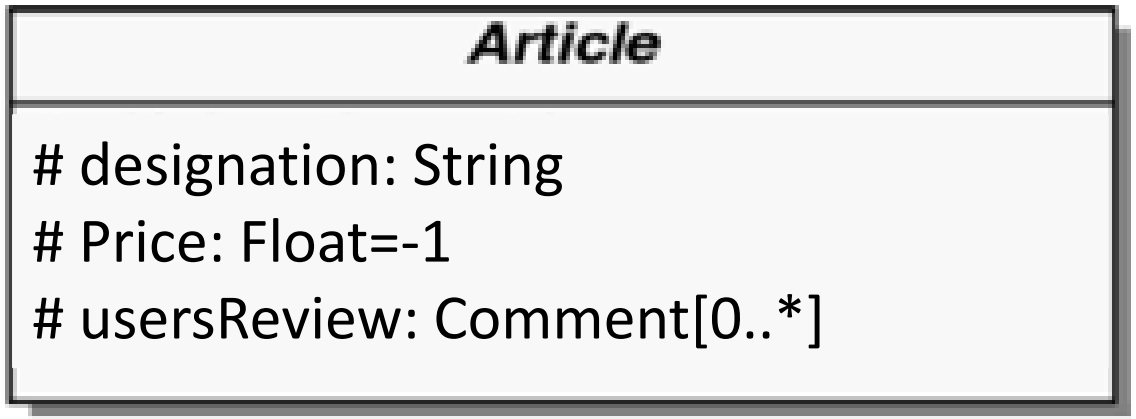
- ✓"An attribute represents some property of the thing you are modeling that is shared by all objects of that class." [Booch et.al.: UML User Guide, 1999, p. 50]
- ✓Attribute Values = Object State.
- ✓Different objects (i.e. different identities) can have the same attributes.
- ✓**Example**: Jean1 and Jean2 are two different objects having the same attributes.



Attributes

- ✓An attribute can be **initialized** and its **visibility** is defined during its declaration.
- ✓**Syntax of the declaration of an attribute**:

modifAcces nameAtt: nameClasse[multiplicity]= valueInit

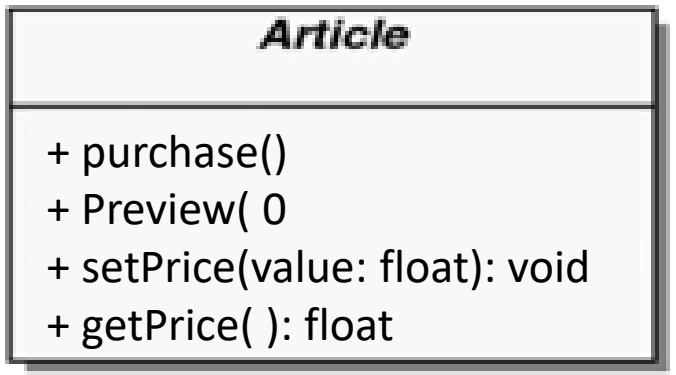


Operation

- ✓An **operation** is defined by its **name** as well as its **parameter types** and the **type of its return value**.
- ✓Syntax of the declaration of an operation:

  modifAcces nameOperation(parameters): ClassReturn
- ✓Syntax of the declaration of parameters :

  nameClass1 nameParam1, …, nameClassN nameParamN
- ✓**Example**:



Association

- ✓An **association** is a structural relationship between objects.
- ✓An **association** is often used to represent the possible links between objects of given classes.
- ✓It is represented by a line between classes.
- ✓**Example**:

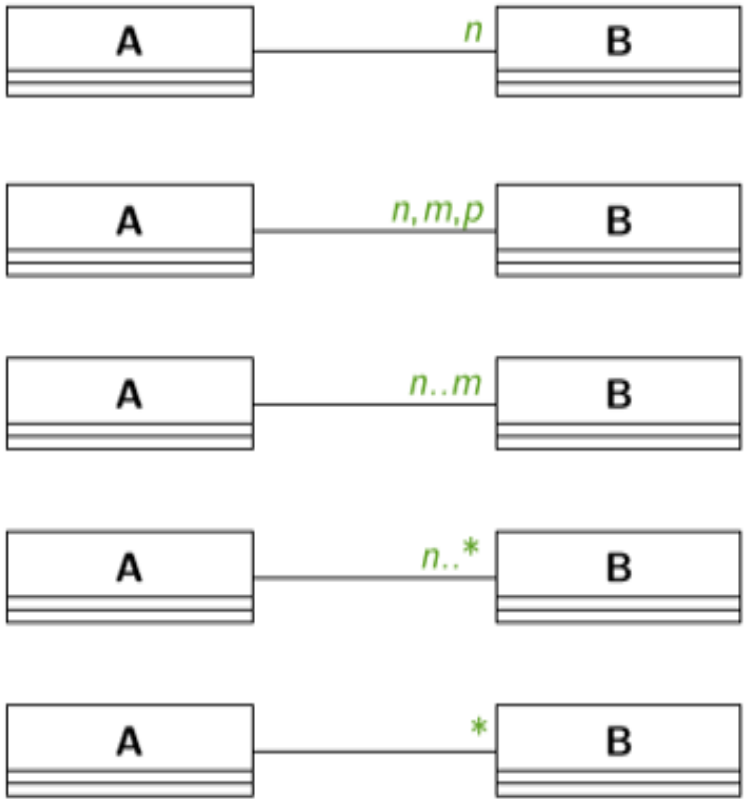
Multiplicities

- ✓The **notion of multiplicity** makes it possible to **control the number of objects involved in each instance of an association.**
- ✓**Example**: an article belongs to only one category (1); a category concerns more than 0 articles, without maximum(*).

- ✓The syntax is **MultMin** .. **MultMax**.
- ✓"*" instead of **MultMax** means "**several**" without specifying a number.
- ✓"n .. n" is also denoted "n", and "0 .. *" is denoted "*".

Multiplicities

Number of objects of the class B associated with an object of the class A:



Exactly n

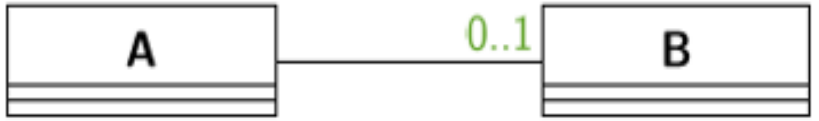
Exactly n or m or p

Between n and m

At least n

Many (0 or more)

Exactly 1

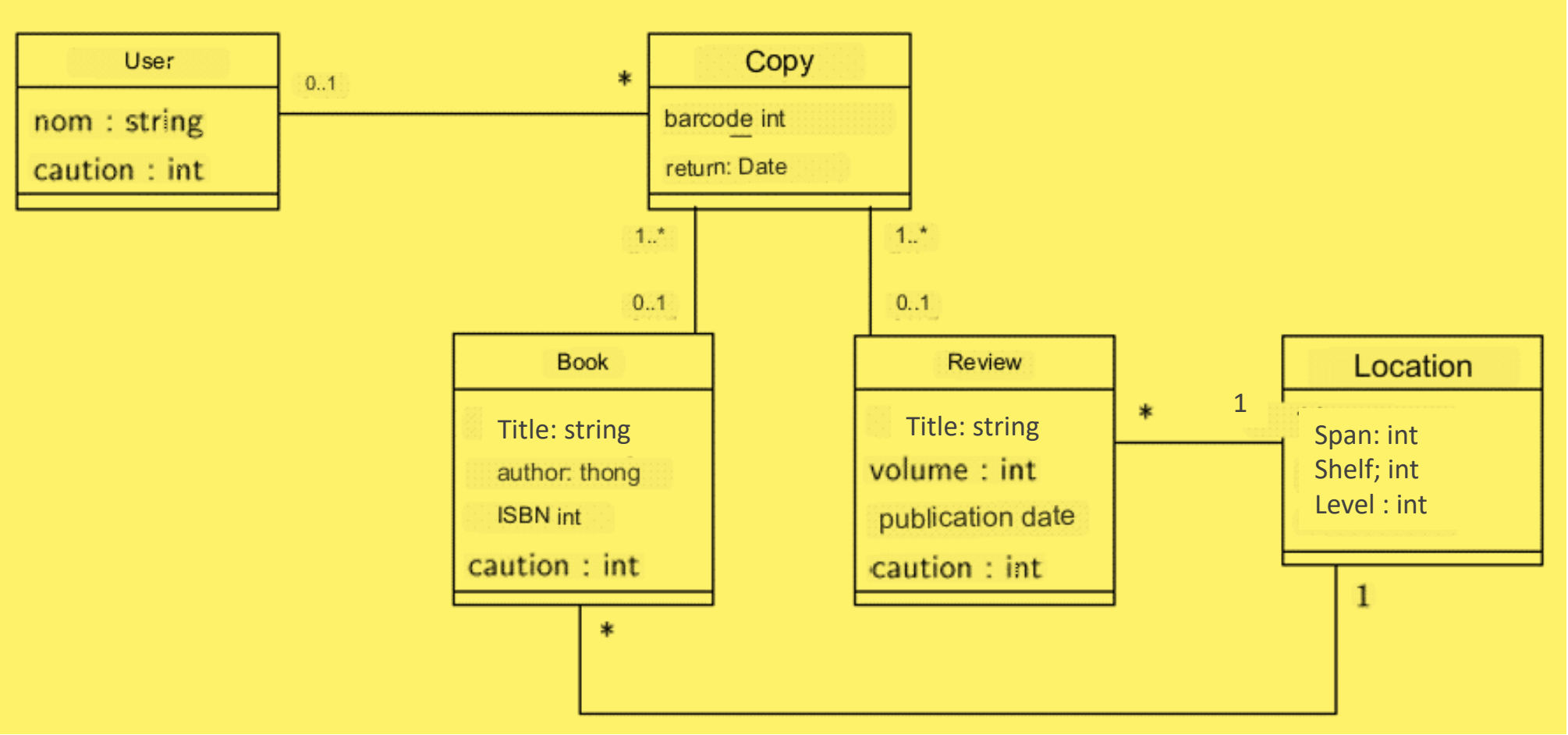


At most 1

At least once

0 or more

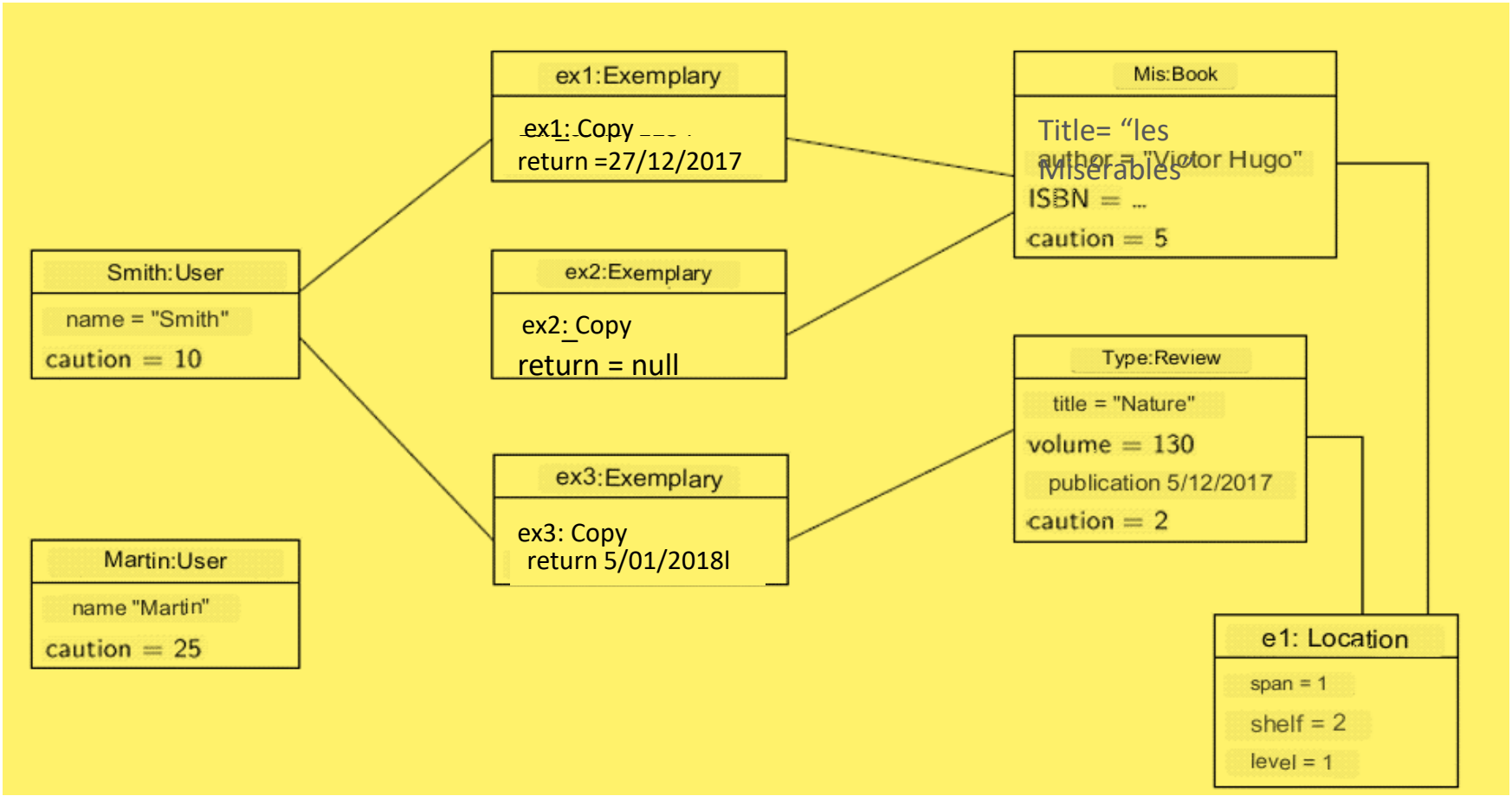
Multiplicities



Example of Class Diagram with multiplicities

Multiplicities

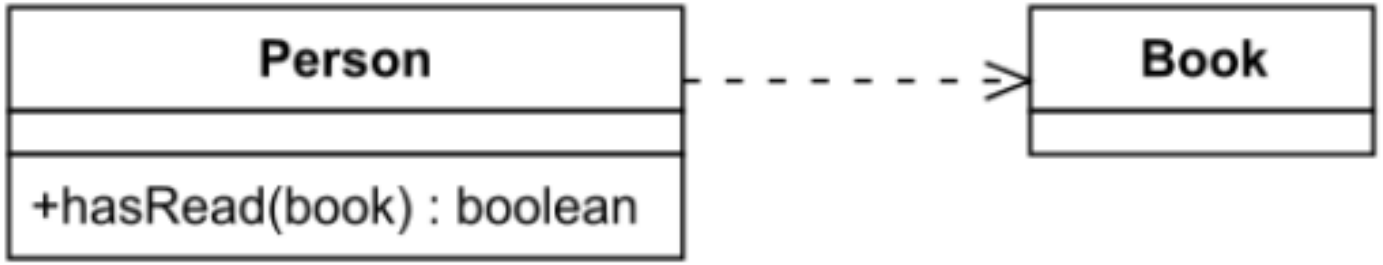
- This object diagram respect the multiplicities of the above class diagram, thus it is a valid object diagram.



Example of Object Diagram

Dependency

- An object of one class **can utilize an object of another class** in the method code.
- If the object is **not stored in any attribute**, then this is **modeled as a <u>dependency</u> relationship.**
- Example: **Person class** may have a **hasRead** method with a **Book parameter** that returns true whenever person has read the book.

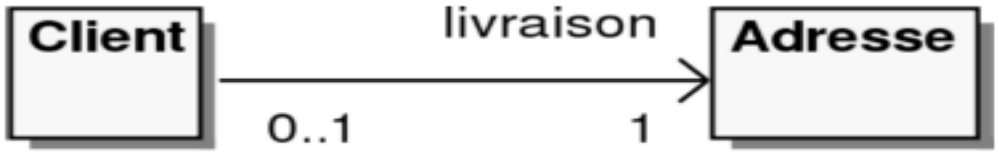


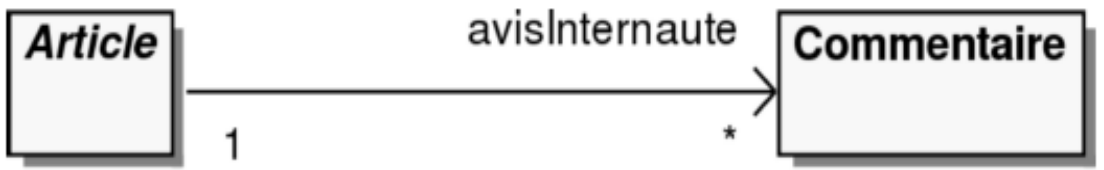
Unidirectional Association

- ✓An object may **store another object in a attribute**.
- ✓**Example**: this relation is modeled by the owns attribute (person own books).

But, a book may be owned by several persons, so the reverse direction may not be modeled.

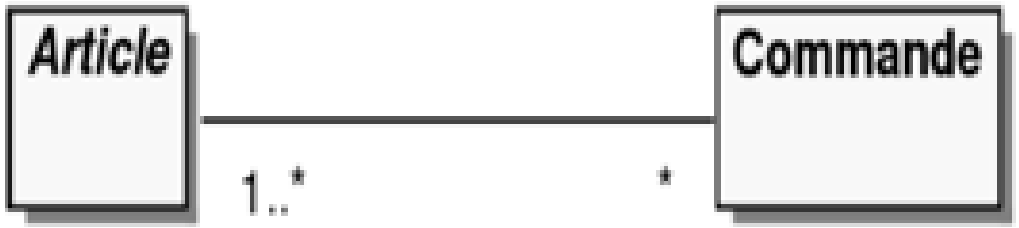
- ✓The *'s in the figure designate that a book may be owned any number of people, and that a person can own any number of books.
- ✓Other examples:





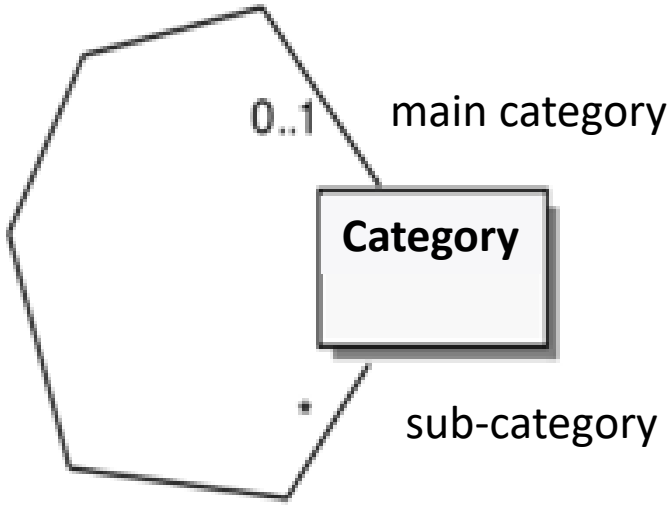
Binary Association

- A **Bidirectional** (**Binary**)**Association** allows us to associate the instances of two classes with one another.
- Two objects might store each other in fields (attributes).
- **Example**: in addition to a Person object listing all the books that the person owns, a Book object might list all the people that own it.



Reflexive Association

- ✓The most used association is the binary association (linking two classes)
- ✓Sometimes both **ends of the association point to the same class**.
- ✓In this case, the association is called "**reflexive**".
- ✓**Example**:



Association Class

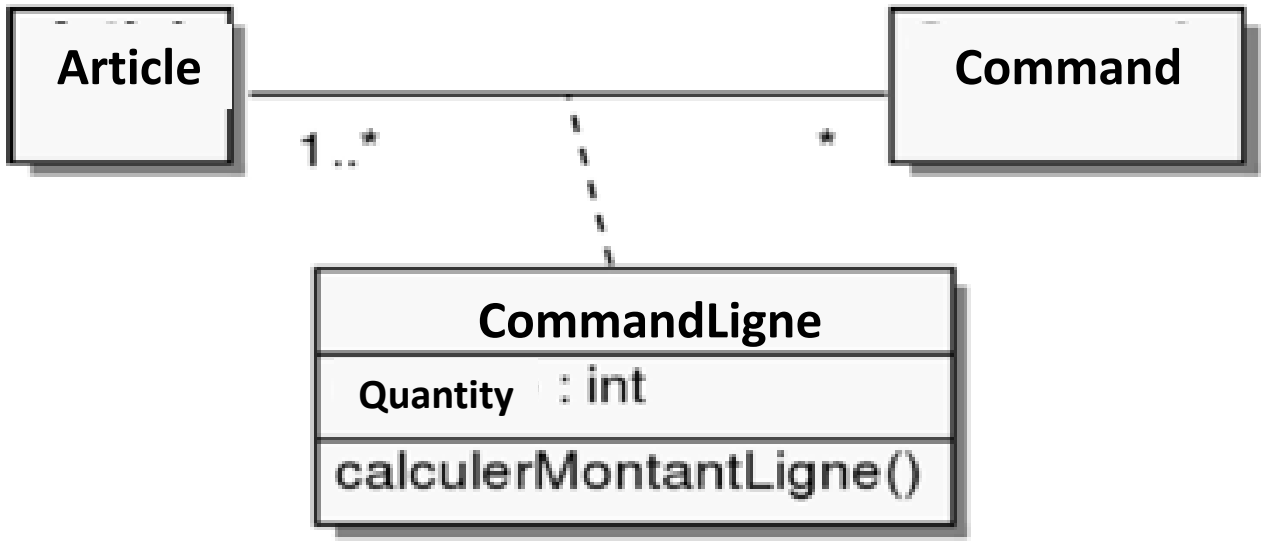
- An association can be **refined** and have its own attributes, which are **not available in any of the classes** it links.

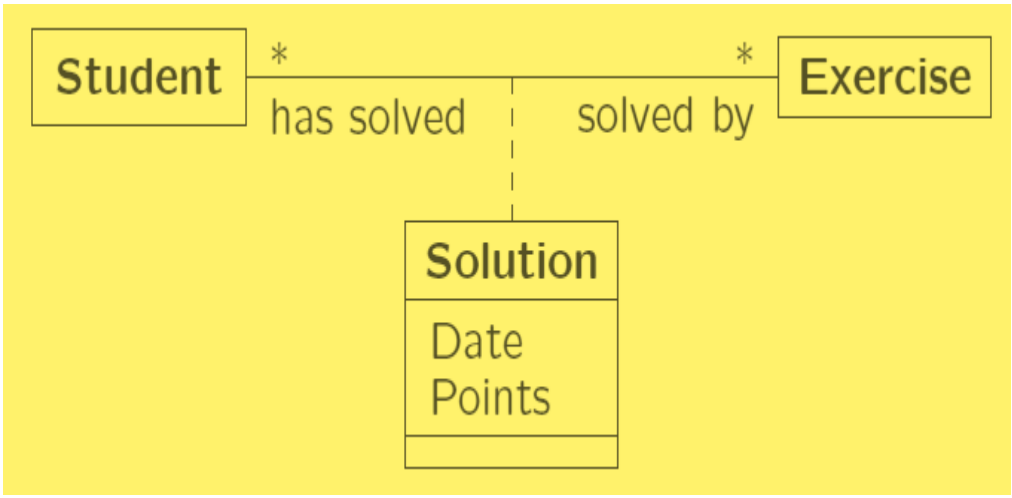
✓As, in the object model, **only classes can have attributes**, then this association becomes a class called "**association class**".

✓Thus, if an association has attributes (or operations), an “association class” must be used.

✓An association class is shown as a class that is linked by a dashed line to an association.

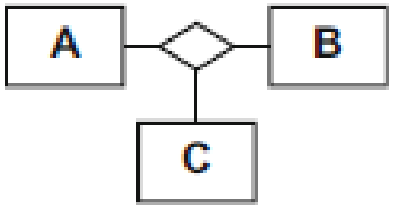
✓**Example**:

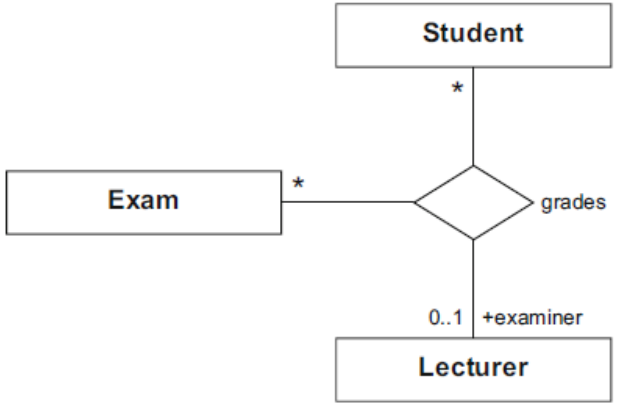


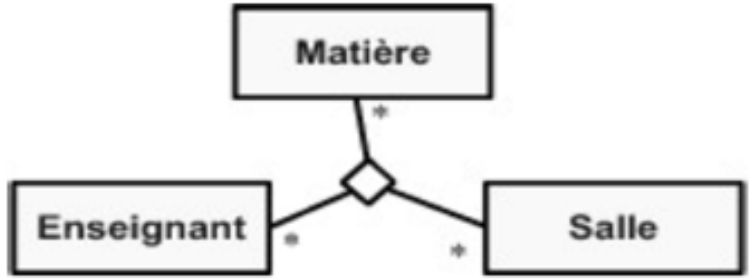


N-Ary Association

- ✓An **N-Ary association** links **more than two classes**.
- ✓Represented by **a central diamond** that can optionally accommodate an association class.
- ✓The **multiplicity** of each class applies to an instance of the diamond.
- ✓**N-ary associations** are **infrequent** and mainly concern cases where the multiplicities are all "*".
- ✓In most cases, it will be more advantageous to use association-classes or several binary relations.
- ✓**Examples**:

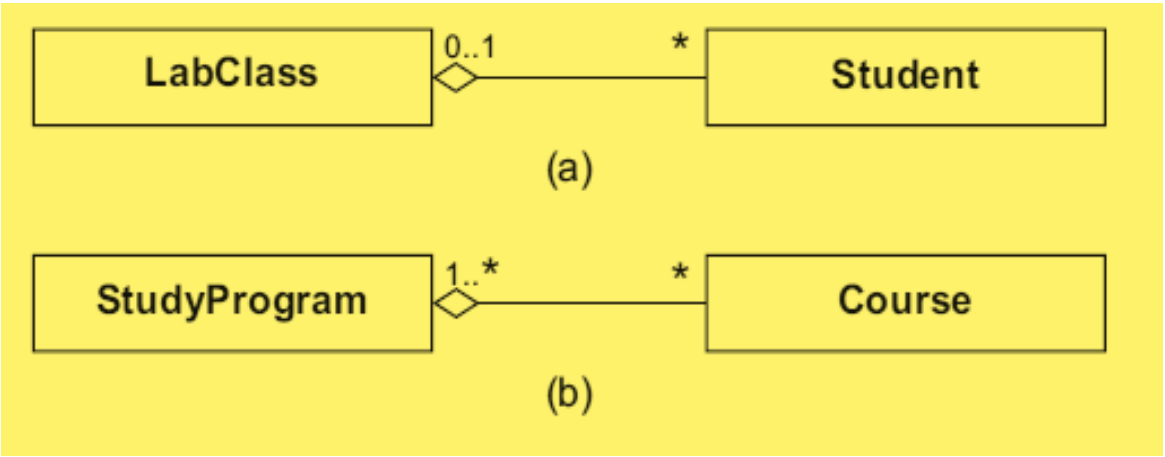


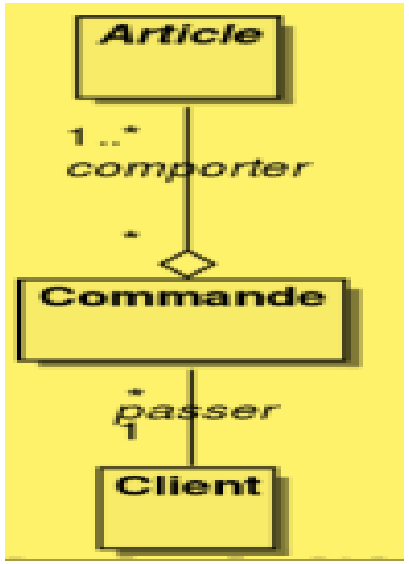




Aggregation

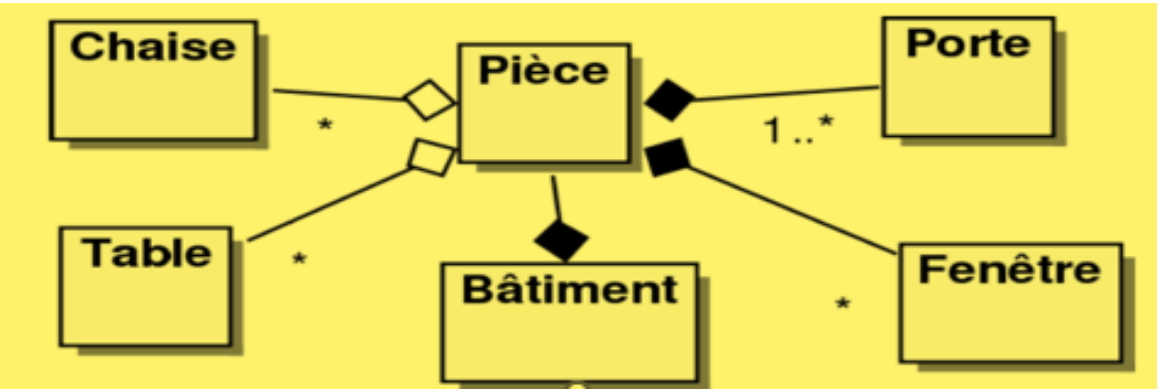
- ✓An **aggregation** is a **special form of association**. It represents the **inclusion relationship** of an element in a set.
- ✓The **aggregation** is represented by an empty diamond on the side of the aggregate.
- ✓An aggregation denotes a relationship of a set to its parts. The set is the **aggregate**, and the part is the **aggregated**.
- ✓**Examples**:





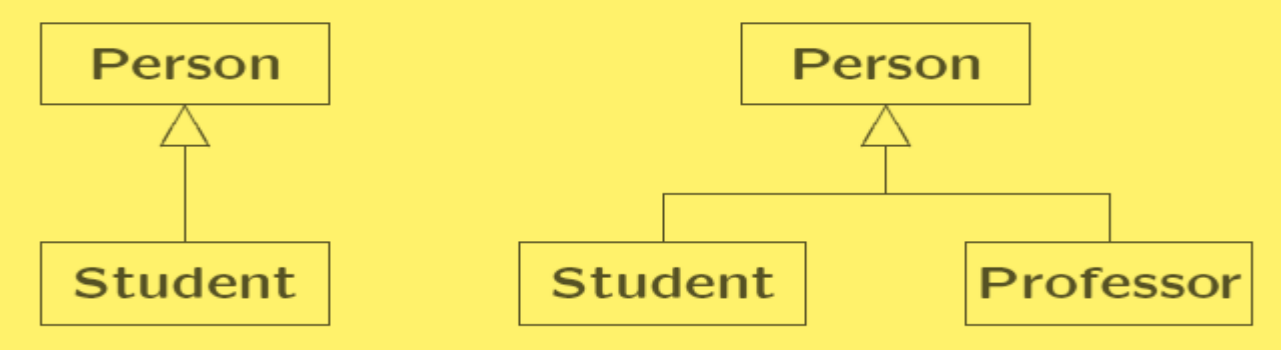
Composition

- ✓The **composition relationship** describes a **structural containment** between instances.
- ✓We use a **solid diamond**.
- ✓**Destroying** and **copying** the **composite object** (the whole or the set) involves respectively **destroying** or **copying** its components (the parts).
- ✓An **instance of the part** never belongs to more than one instance of the composite element.
- ✓**Examples**:



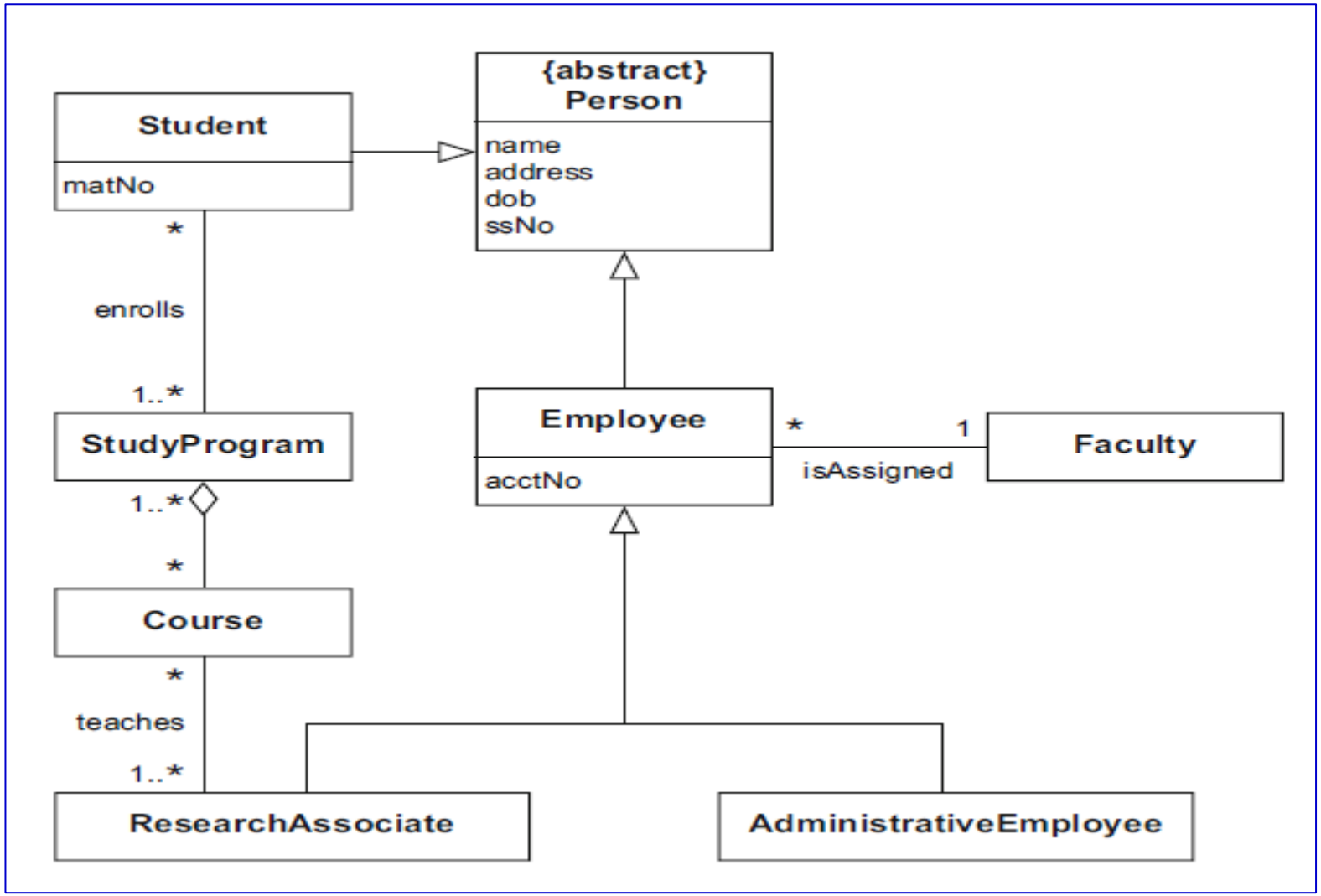
Generalization

- ✓"A **generalization** specifies a relationship **between a general class** (superclass or parent) and a **more specific class** (subclass or child).
- ✓**Generalization** is also named an **"is- a "** relationship."
- ✓**Generalization** is represented by an arrow (with a large open triangle at the end) directed from the subclass to superclass (in the "is a" direction).
- ✓If a class has several subclasses, we can adopt the two notations:



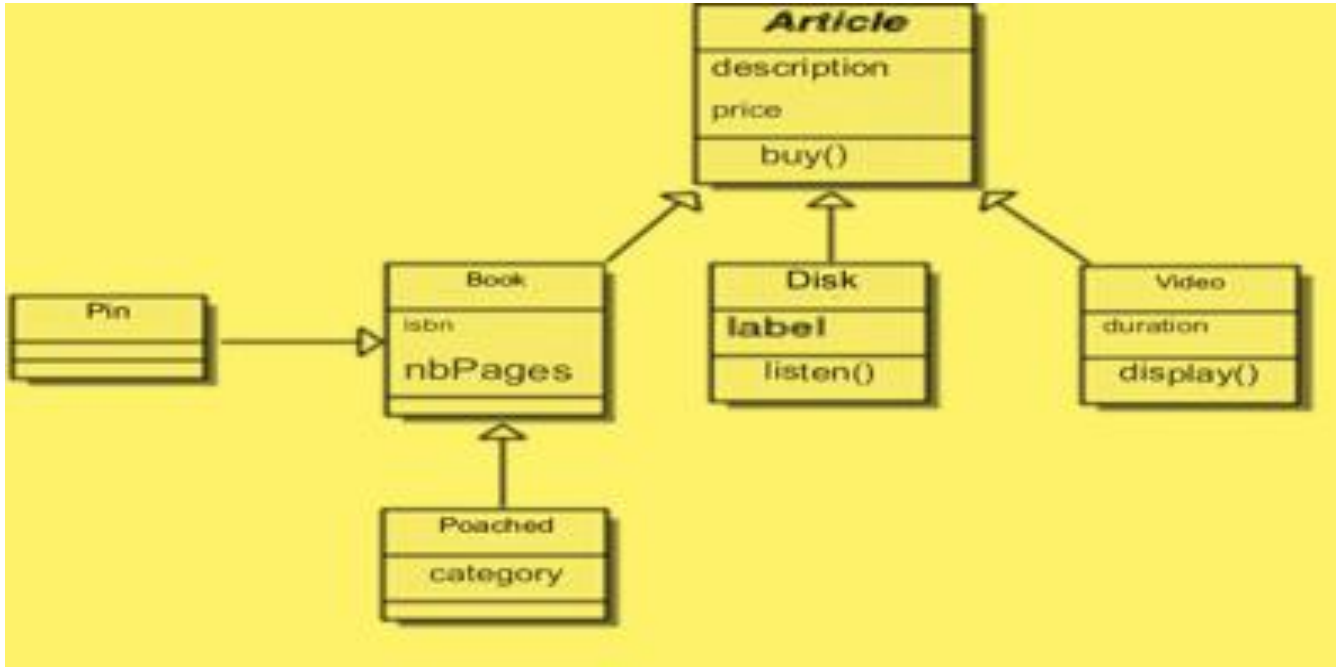
Generalization

- ✓This figure presents a classes diagram with generalization.
- ✓Generalization for example from Student to Person, from Employee to Person, …etc



Inheritance

- ✓**Inheritance** a **specialization / generalization relationship**.
- ✓**Specialized elements** inherit **structure and behavior** from **more general elements** (attributes and operations)
- ✓**Example**: by article inheritance, a book automatically has a price, a designation and a buy( ) operation, without it being necessary to specify it.

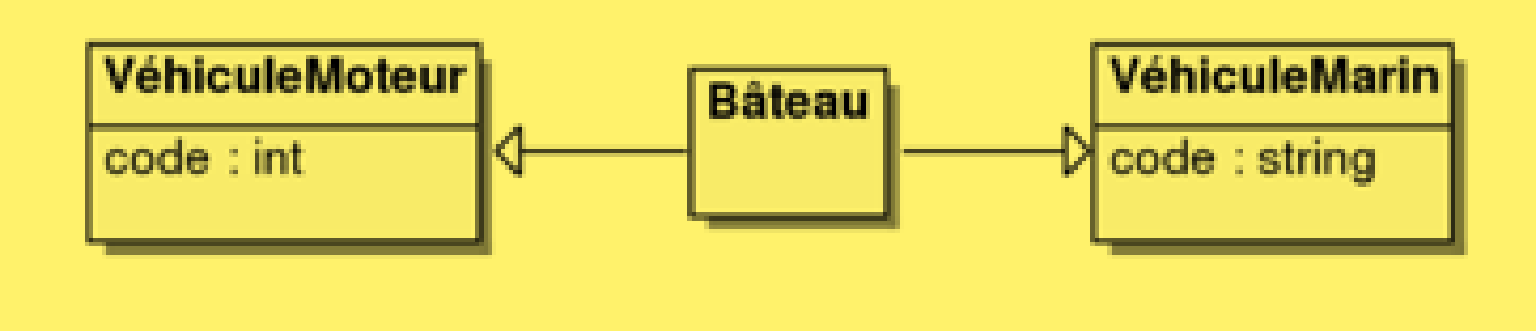


Inheritance

- ✓The **child class** has all the properties of its parent classes (attributes and operations).
  - ✓The **child class** is the **specialized class**.
  - ✓The **parent class** is the **generalized class**.
- ✓However, it does not have access to **private properties**.

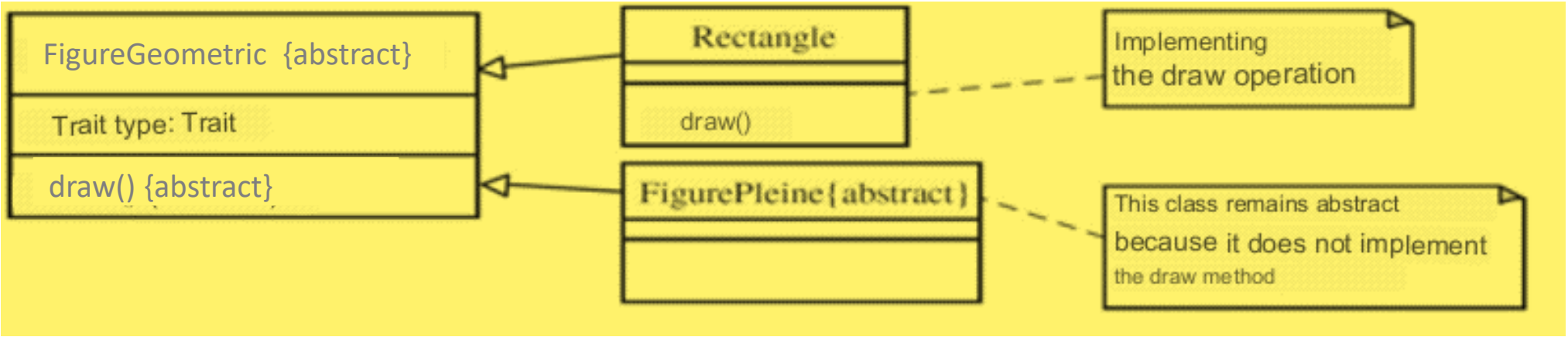
Multiple Inheritance

- A class can have several parent classes. This is called multiple inheritance.
- The C++ language is an example of a language that allows its effective implementation.
- Java does not allow this.



Abstract Class

- ✓A method is said to be abstract when we know its header but not the way in which it can be implemented.
- ✓It is up to the child classes to define the abstract methods.
- ✓A class is said to be abstract when it defines at least one abstract method or when a parent class contains an abstract method that has not yet been implemented.



Interface

- ✓**The role of an interface** is to **group together a set of operations** ensuring a consistent service offered by a workbook (classifier) and a class in particular.
- ✓An **interface** is defined as a class, with the same compartments.
- ✓We add the "**interface**" stereotype before the interface name.
- ✓We use a "**realization**" type relationship between an interface and a class that implements it.

- ✓**Classes that implement an interface** must implement all the operations described in the interface.