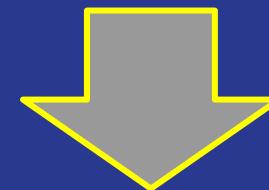


Concepts Objects



Key Words / Mots Clés

- Object Theory
- Object Oriented Programming
 - / Programmation Orientée Objet
 - JAVA (C++)
- Modeling Language
 - UML,
- Design Pattern
 - / Patron de conception



— Simulation Project



Main References

- Courses of
 - Pr. Christine Force,
 - UML
 - Pr. David Hill,
 - Simulation & C++
 - Ferdinand Piette (ex-ISEN),
 - UML
- Books & Websites
 - Design Patterns - Gamma et al.,
 - <http://www.volubis.fr/news/liens/courshtm/Java/objet.html>,
 - <https://openclassrooms.com/>,
 - <https://creately.com/blog/diagrams/uml-diagram-types-examples/>,
 - https://www.ntu.edu.sg/home/ehchua/programming/cpp/cp3_OOP.html,
 - UML & Design Pattern from Eyrolles and DUNOD books.a

Concepts Objets

8 sessions / séances

End of October 2020

Last session / Dernière séance

Sept

Oct

September the 1st 2020

1st session / séance

- Window [13.30 - 17.15] splitted into two parts of 1h45 :
 - [13.30 - 15.30] ; 15 minutes break ; [15.45 - 17.15] ???
- All in one:
 - Eng: Recitation, Lab & Lecture.
 - / TD, TP & CM.

To Write or not to Write ?

- *Diapos disponibles :*
 - Pendant ou après chaque cours ;
- Les corrections (TD & TP) seront données en cours (hors *slides*).

- *Slides available during (e.g. exercises) or after each session*
- Lab et recitation exercises corrections are given during the session.

English or French ?

- *Slides mainly in English*
- *Difficult words given in both languages like this:*
 - *English / French*

Evaluation

“paper test”

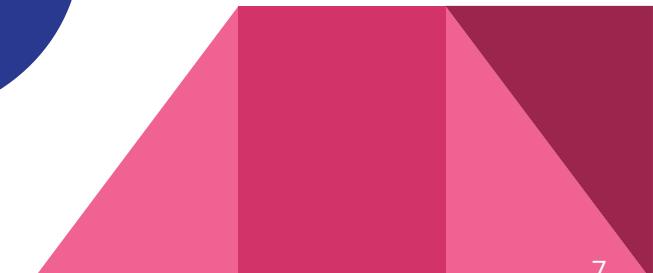
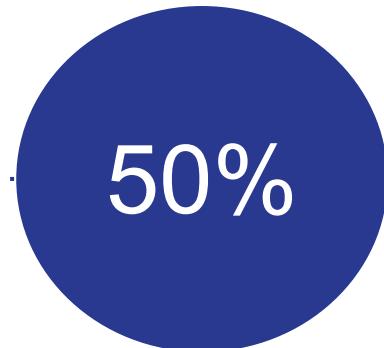
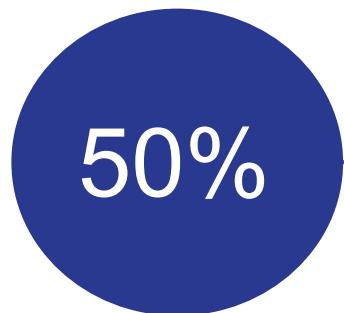
Exam

Simulation mini-Project
/ mini-Projet “Simulation”

JAVA Code (or **C++**) &
short report.

Some time windows
dedicated to it.

Group of 3 or 4 students.



1. Introduction

1.1. First Objects / Premiers Objets

- 1967 : **SIMULA** language
 - First language using the “**Class**” concept,
 - **Discrete event simulation**,
 - / **simulation à événements discrets**,
 - Norwegian Computing Centre of Oslo 
 - 1st authors : Ole-Johan **Dahl** & Kristen **Nygaard** ;
- 1972 : **Smalltalk** language
 - First **object-oriented programming** language
 - / 1er langage de programmation orientée objet



SMALLTALK



- 1967 : **SIMULA**

```
Procedure report (message); Text message; Begin
    OutFix (Time, 2, 0); OutText (" : " & message); OutImage;
End;
```



- 1972 : **Smalltalk**



```
exampleWithNumber: x
    | y |
    true & false not & (nil isNil) ifFalse: [self halt]
    y := self size + super size.
    #($a #a "a" 1 1.0)
        do: [ :each |
            Transcript show: (each class name);
            show: ' '].
    ^x < y
```



Main object-oriented programming languages

The 2 languages you
can use in this course

- C#
- C++
- PHP
- JAVA
- Delphi
- “Python”
 - (Some concepts only)
- Objective-C
- Precursors :
 - SIMULA-67,
 - SMALLTALK-80

In the exercises, we will generally correct the JAVA code, but we
can check together what to do in C++.

First Concepts

One Object

- Data_(attributes) / Données_(attributs)
- Methods / Méthodes

- An object is an entity composed of **data** and code (**methods**) acting generally on this data.
- The data characterize the **state** of an object and the methods make it possible to pass **from one state to another**.

First Concepts

A Class / Une Classe

- **Model for creating an object**
 - / Modèle de création d'objet.
- **A class describes the structure of an object: data and actions on these data (methods)**
 - / Une classe décrit la structure d'un objet : les **données** et les actions possibles sur ces données (**méthodes**).

First Concepts

An Instance

- **Object obtained from the instantiation process based on a class**
 - / Objet obtenu par **instanciation** à partir d'une **classe**.



Example

Class Lion

Data

Name
Sex
Coordinates
Activity
Biotope
Size
Age
Hierarchical position

Methods

Drink
Sleep
Eat
Hunt
Move

Instantiation
1 Object
1 object

Instance

Léonne
Feminine
70 ; 200
Huting
Marigot
1.1 m
3 years
Leader

Instance

Léon
Masculin
12 ; 17
Sleeping
Grove
1.5 m
2,5 years
Subordin ate

First Concepts

Example and Questions



?



?

- Among the 3 first concepts, “Who does what” in the kitchen?
 - / Parmi les 3 concepts de base, “qui fait quoi” en cuisine ?

First Concepts

Example and Questions



Muffin pan **Class**
/ Classe Moule à muffins



These 12 *muffin* objects
are all instances of the
Muffin Pan class.

/ Ces 12 **objets** *muffin* sont
autant d'**instances** de la **classe**
Moule à muffins.

Write by hand what this would give in Java (or C ++).
/ Ecrivez à la main ce que cela donnerait en Java (ou C++).

First Concepts Example - Java



```
class MouleAMuffins{  
  
    String nomGateau ;  
    ...  
}
```

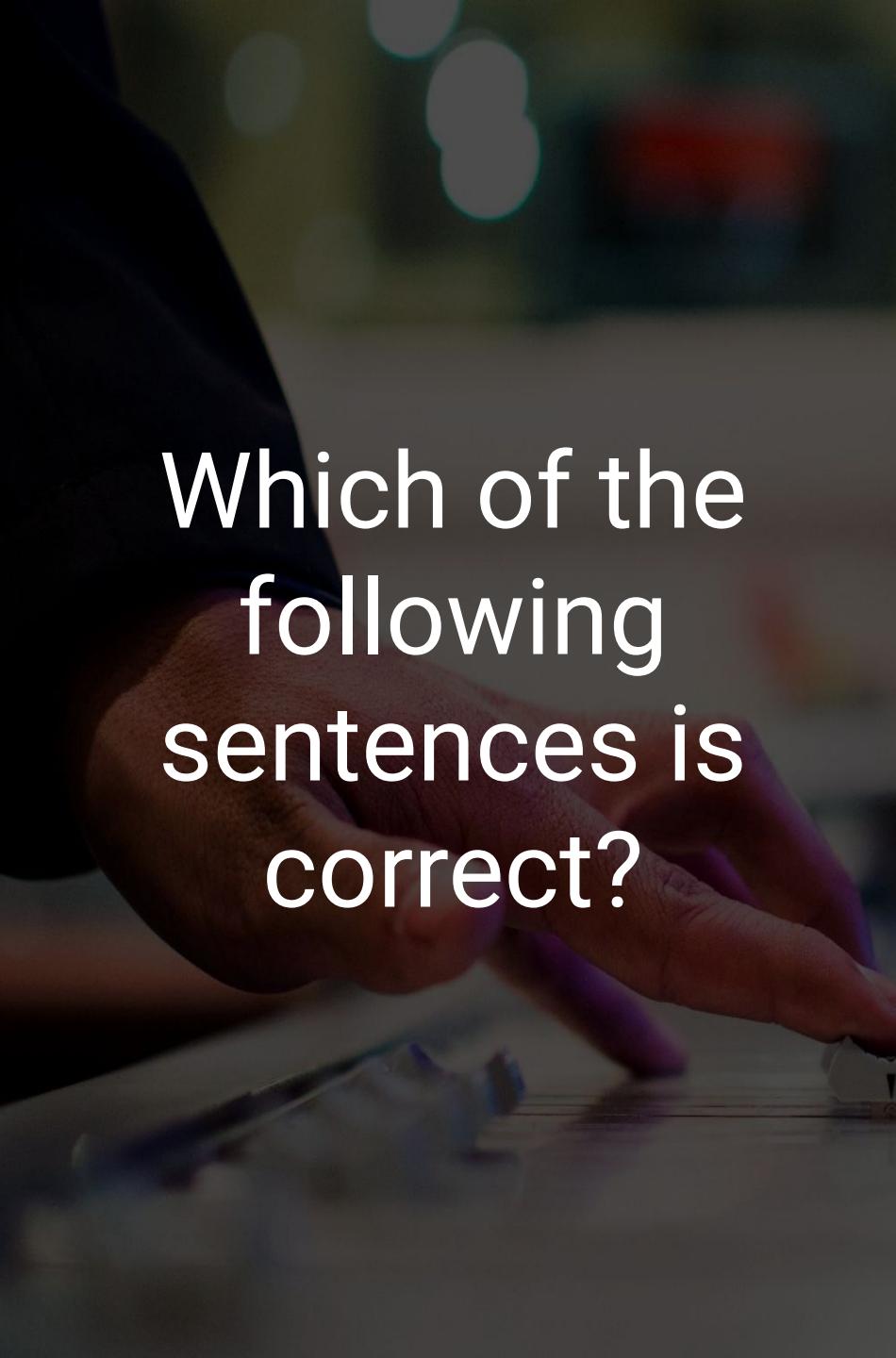


```
MouleAMuffins muffinsChoco  
= new MouleAMuffins();
```



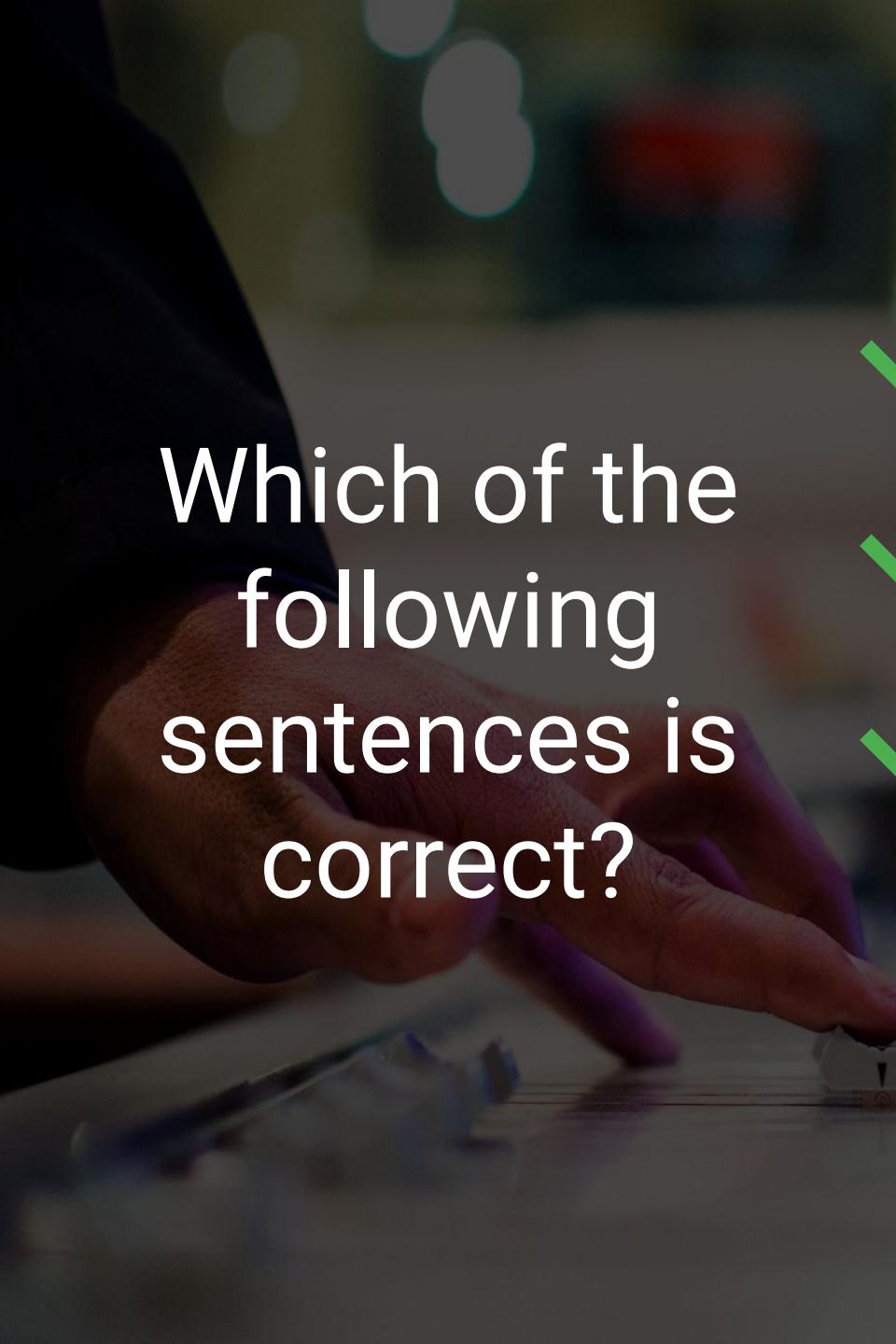
We get a **muffinsChoco object** which is an **instance** of the **MouleAMuffins class**.

/ On obtient un **objet muffinsChoco** qui est une **instance** de la **classe MouleAMuffins**.



Which of the following sentences is correct?

- a > An object is always an instance.
 - b > An instance is always an object.
 - c > An object and an instance are the same thing.
-



Which of the following sentences is correct?

- a > An object is always an instance.
 - b > An instance is always an object.
 - c > An object and an instance are the same thing.
-

2. Basic concepts

2.1. Encapsulation

2.2. Inheritance / Héritage

2.3. Polymorphism

Encapsulation... meaning?

...and how it works in Java/C++?

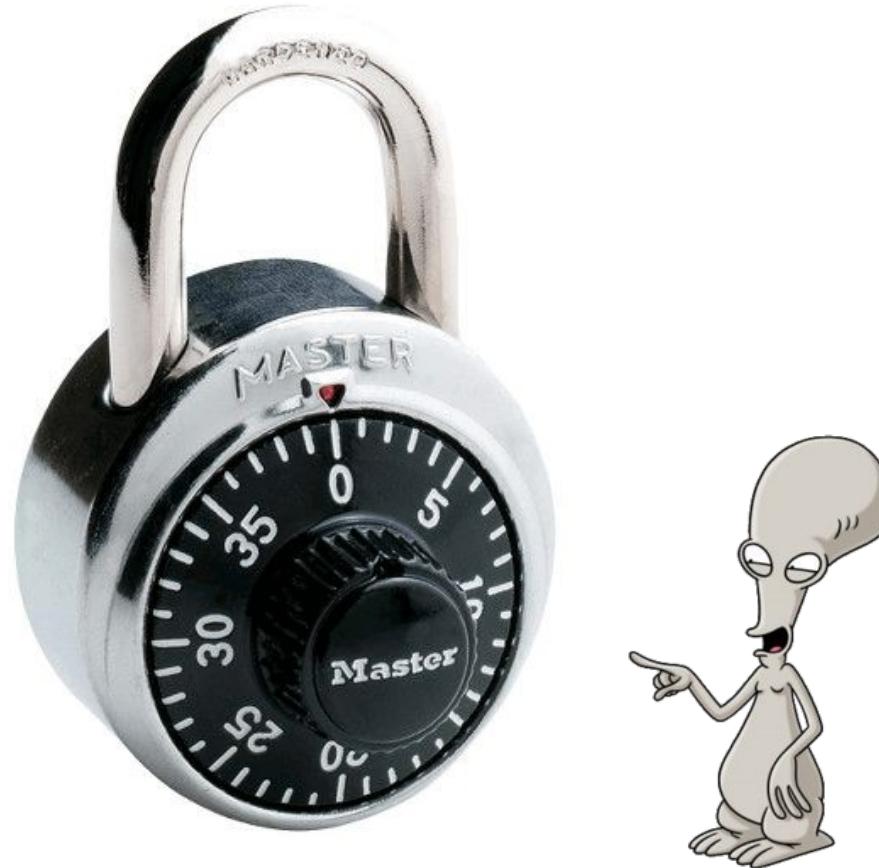
The Three Pillars of OOP



*(OOP : -eng- Object-oriented
programming.)*

2.1.
1st Fundamental Concept

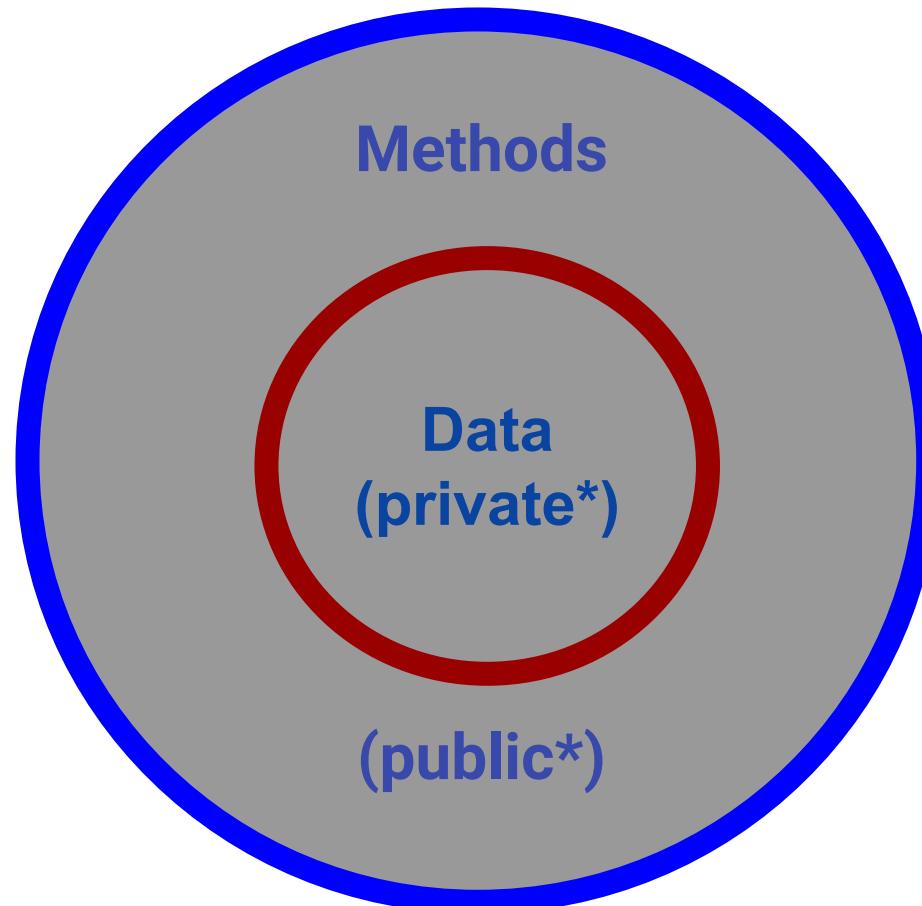
Encapsulation



(of the data)



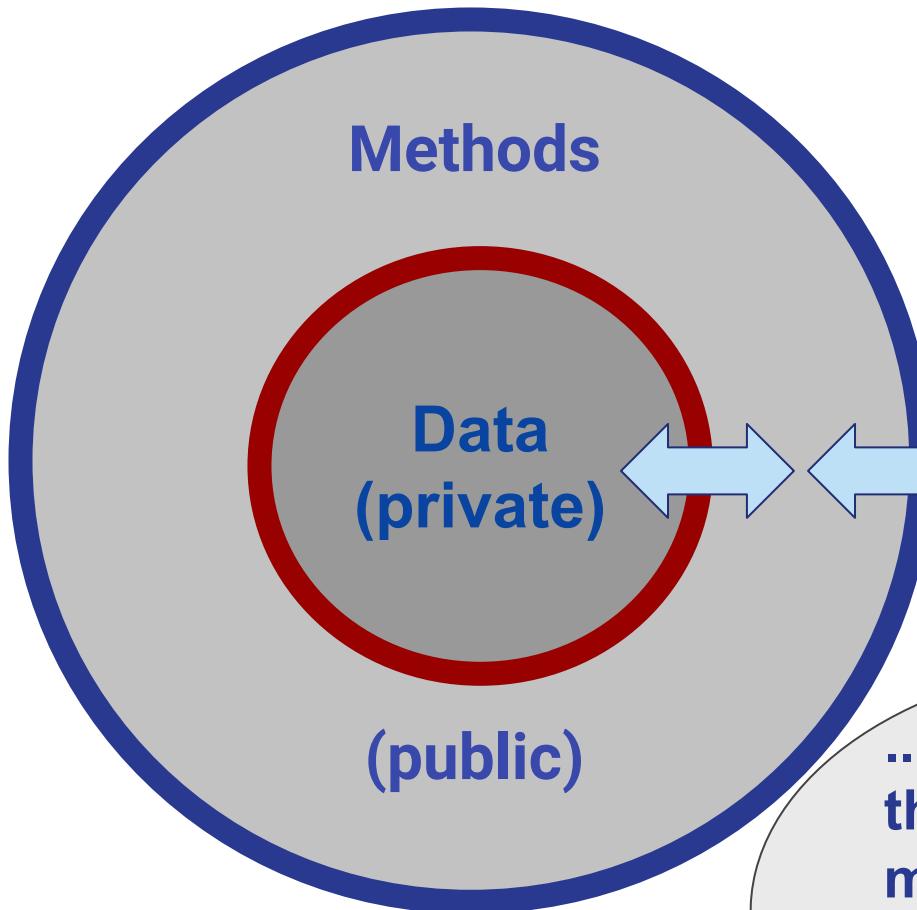
1st Fundamental Concept Encapsulation



* from the point of view outside the object
/ du point de vue extérieur à l'objet

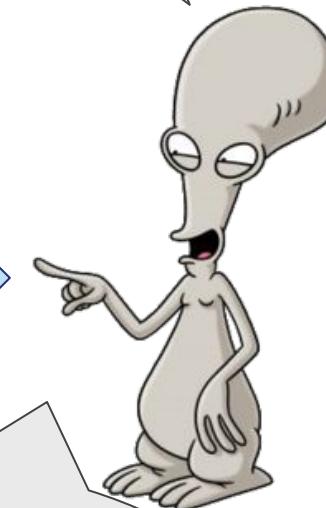
Remark: Some data and methods can be **static**: an attribute or a method can be linked to a class and not to an instance (e.g. an integer "numberDInstances").

Concept fondamental Encapsulation



I want to
modify
private data
...

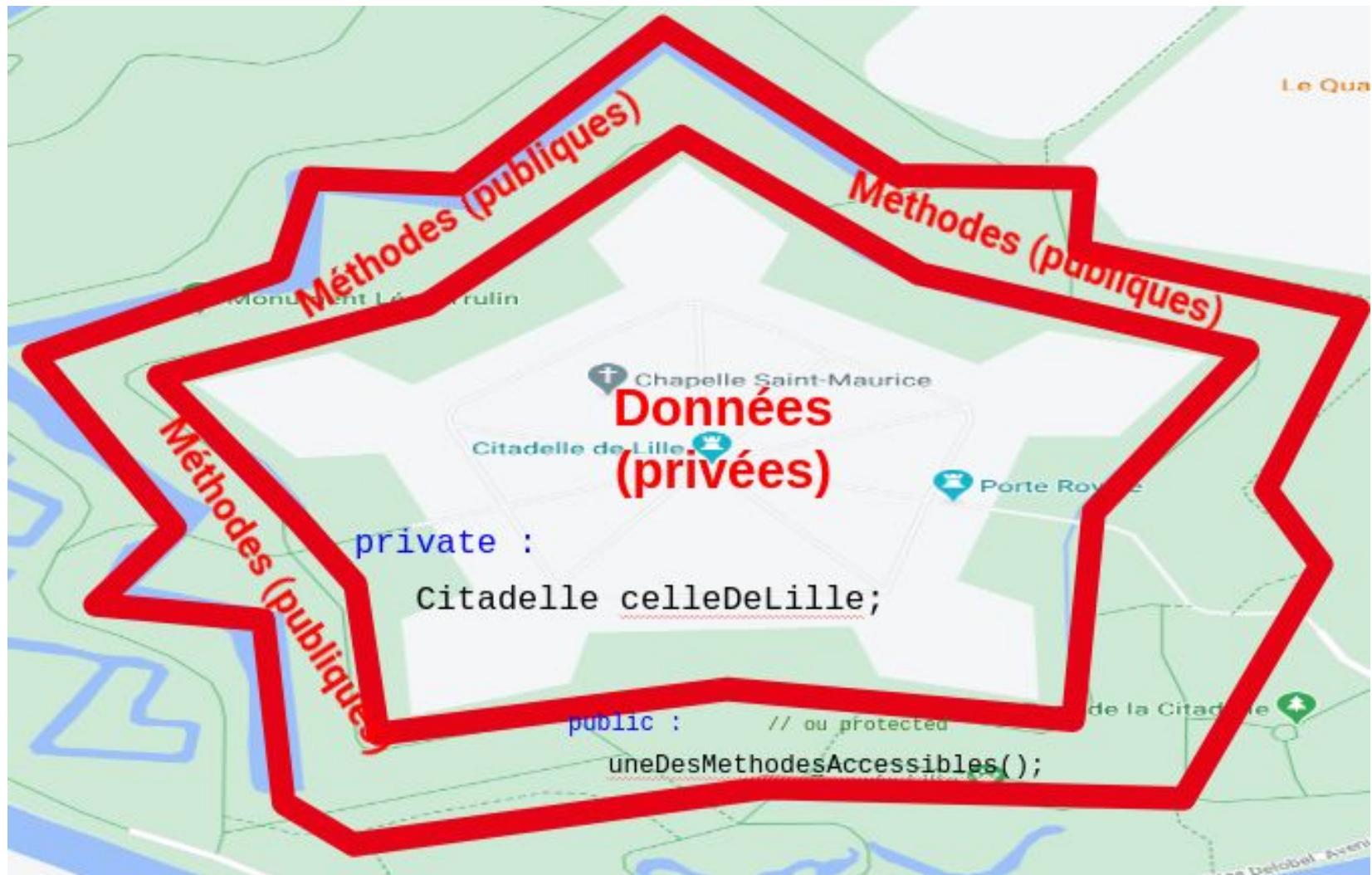
/ Je veux
modifier
une donnée
privée...



... then I go
through a public
method which has
access to the
private data of its
own class.

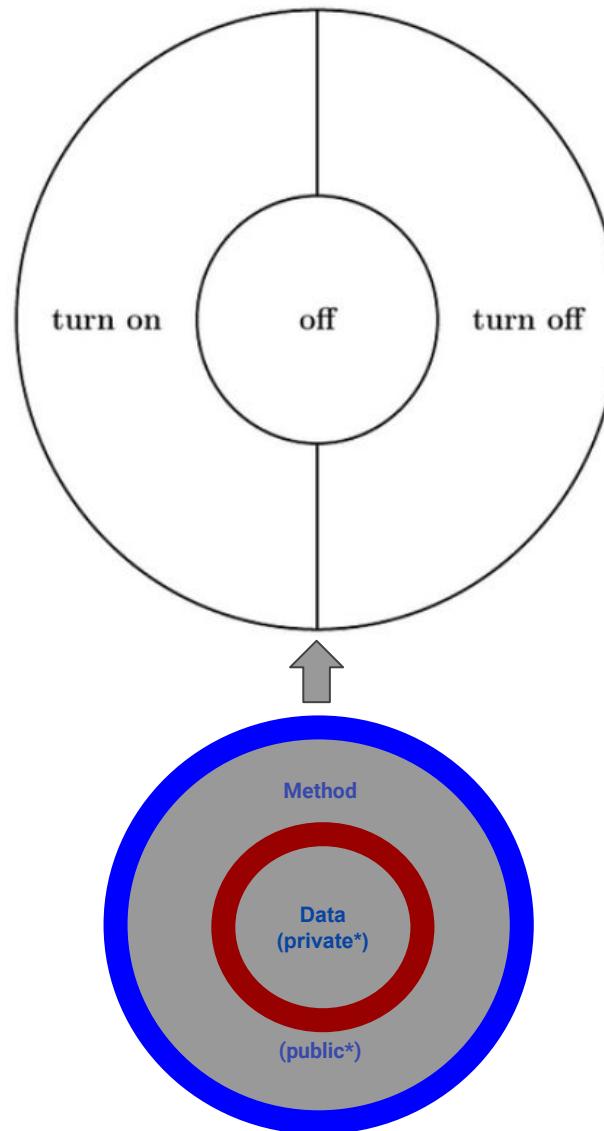
/ ...alors je passe
par une méthode
publique qui a
accès aux
données privées
de sa propre
classe.

Main Goal: the security (c++ example)



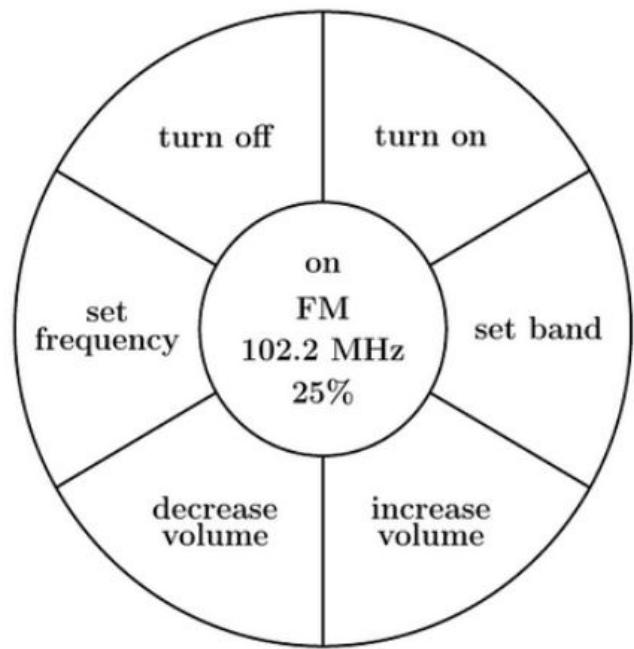
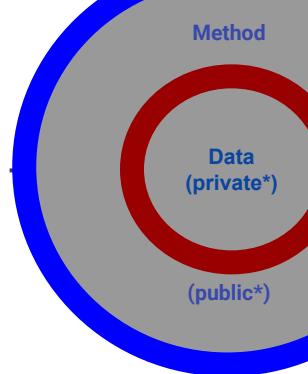
Encapsulation & Data Access

LAMP Example



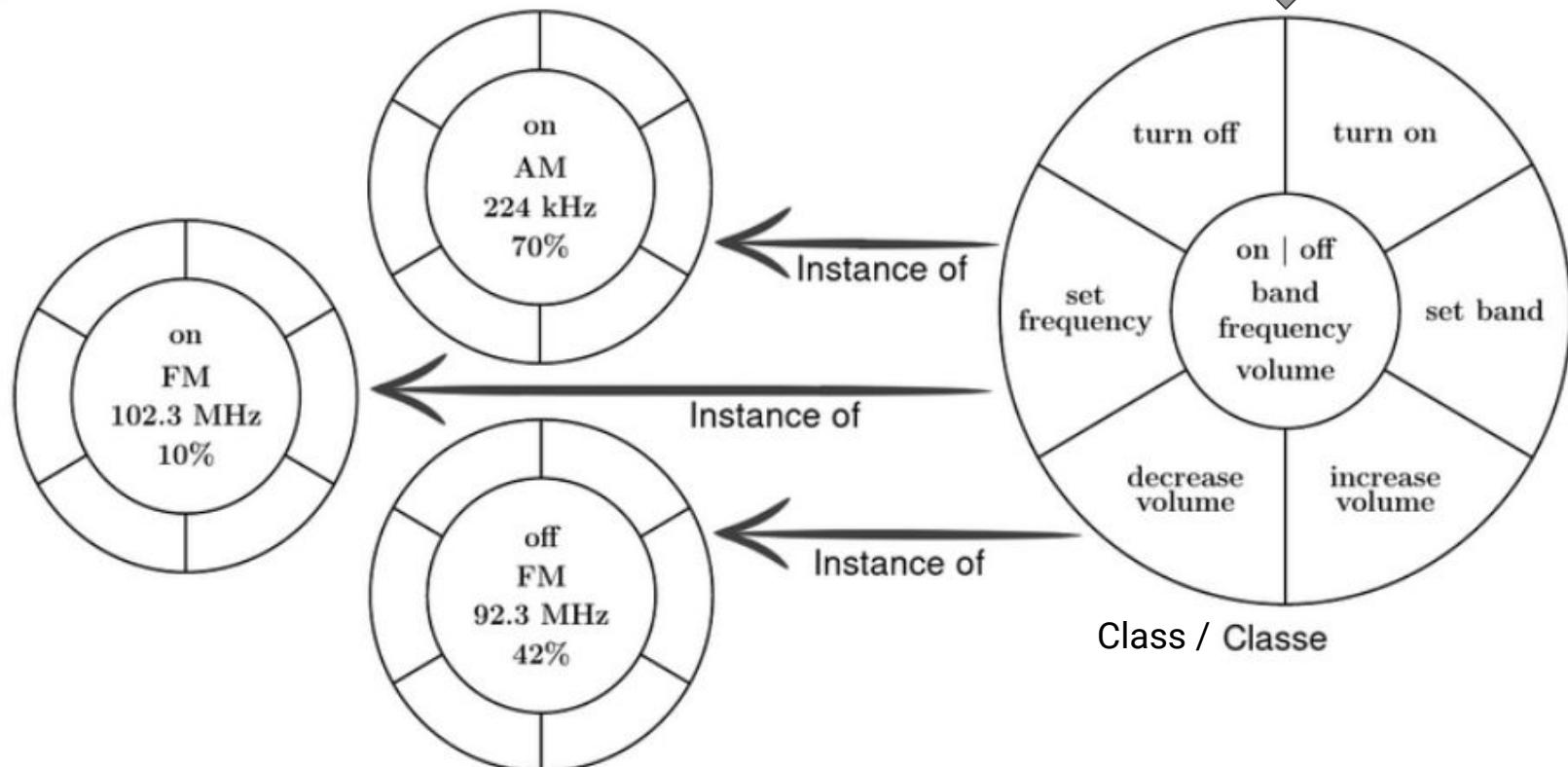
Encapsulation & Data Access

Example: Radio Instance



**1 instance of
Radio class**

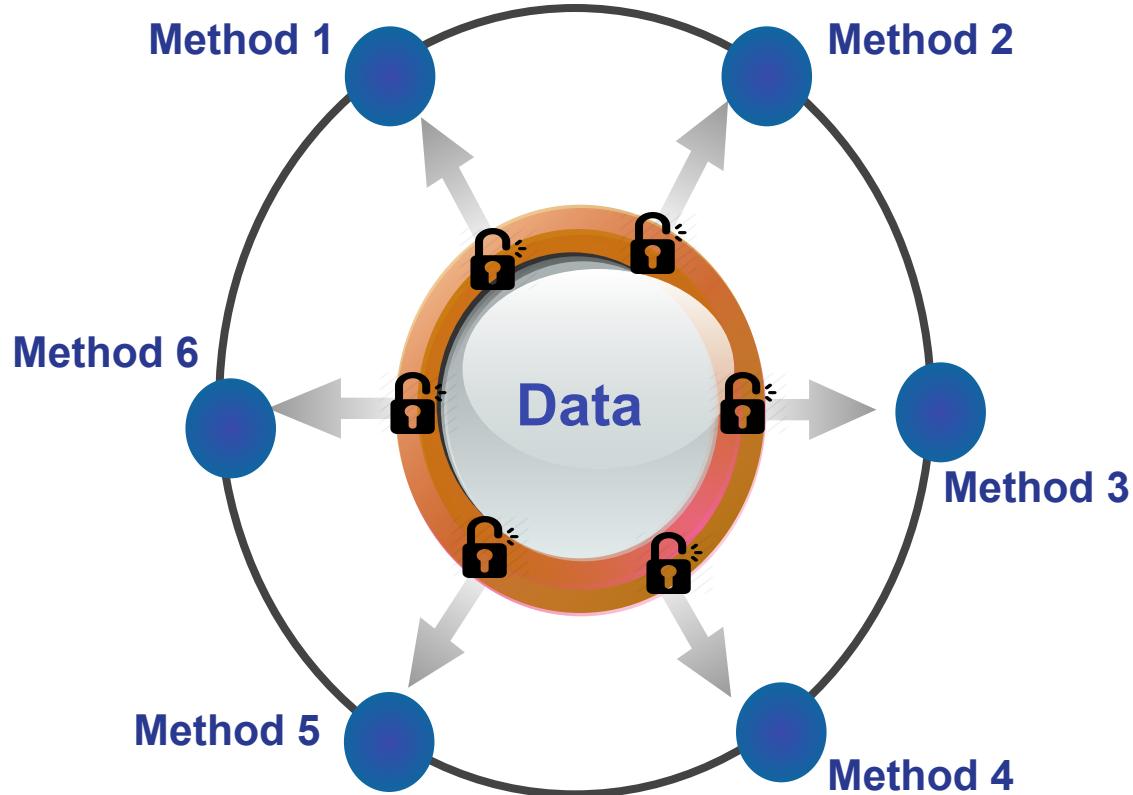
Example : Radio class & 3 instances / objects



KeyWords: **Getters & Setters** (/same in french)

"Getters" = public methods of access to private data.

"Setters" = public methods of modifying data
Private.

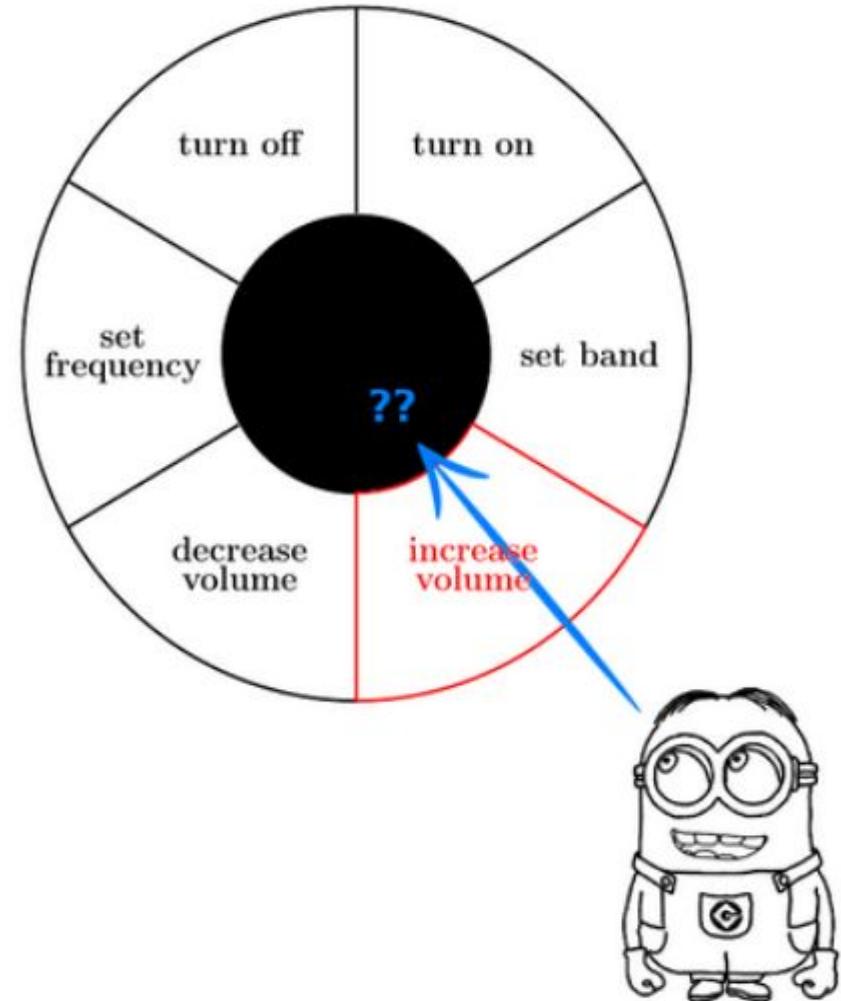


Encapsulation & Data Access

Data visibility / Visibilité des données

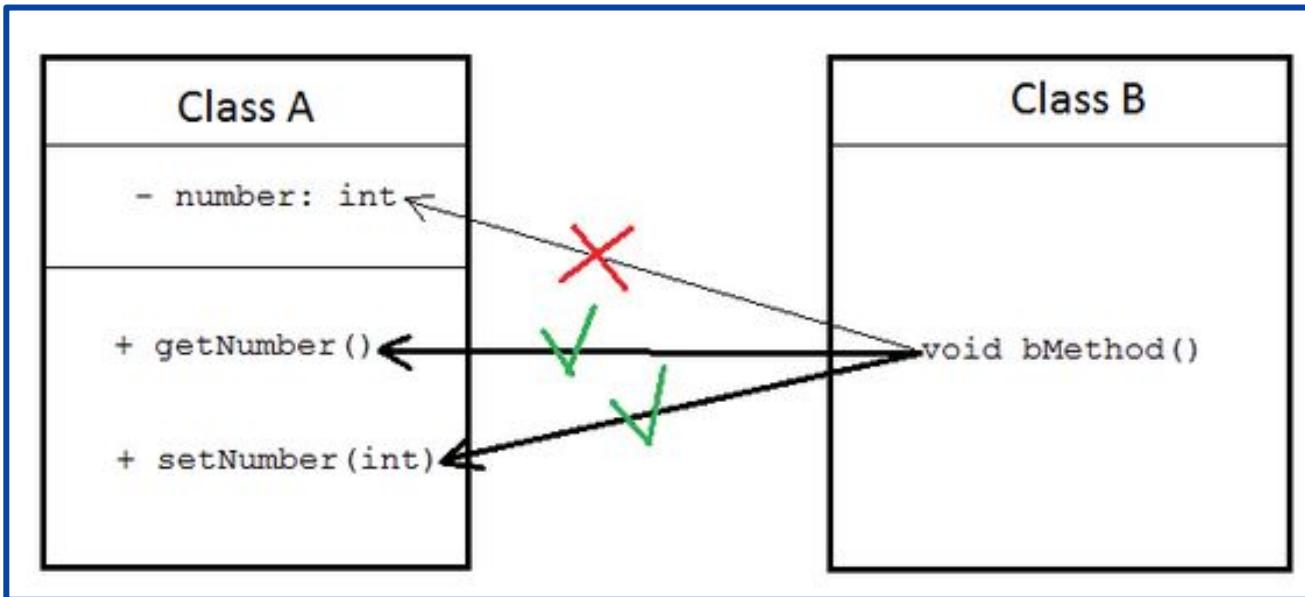
Take a point of view outside the object :

- **Data** are invisible because **Private**;
- **Methods** are the only way to interact with data because **Public**:
 - Access ("**getters**"),
 - Modification ("**setters**"),
 - Other actions on data.



Getters & Setters

- Example in JAVA -



```
1 | public int getNumber() {  
2 |     return this.number;  
3 | }
```

```
1 | public void setNumber(int num) {  
2 |     if (num < 10 || num > 100) {  
3 |         throw new IllegalArgumentException();  
4 |     }  
5 |     this.number = num;  
6 | }
```

JAVA & Muffins



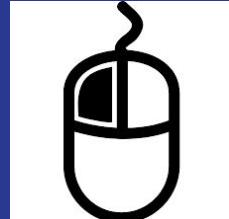
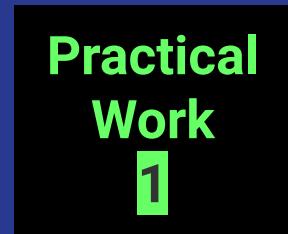
```
/* classe déclarée dans le fichier  
   MouleAMuffins.java */  
public class MouleAMuffins{  
  
    Private String nomGateau ;  
    ...  
  
    Public String getNomGateau();  
    ...  
}
```

```
// e.g., code dans le Main  
MouleAMuffins muffinsChoco  
= new MouleAMuffins() ;  
  
String nomDuGateauInstancie  
= muffinsChoco.getNomGateau();
```

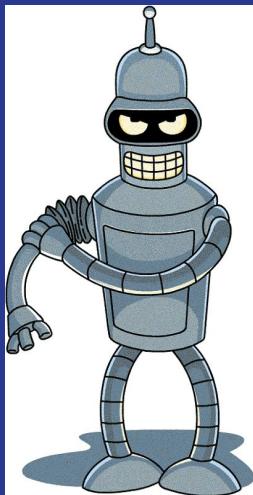
Public ==> Call of the Getter possible outside the methods of the **MouleAMuffins** class.

Practical work (labs)

/ Travaux Pratiques

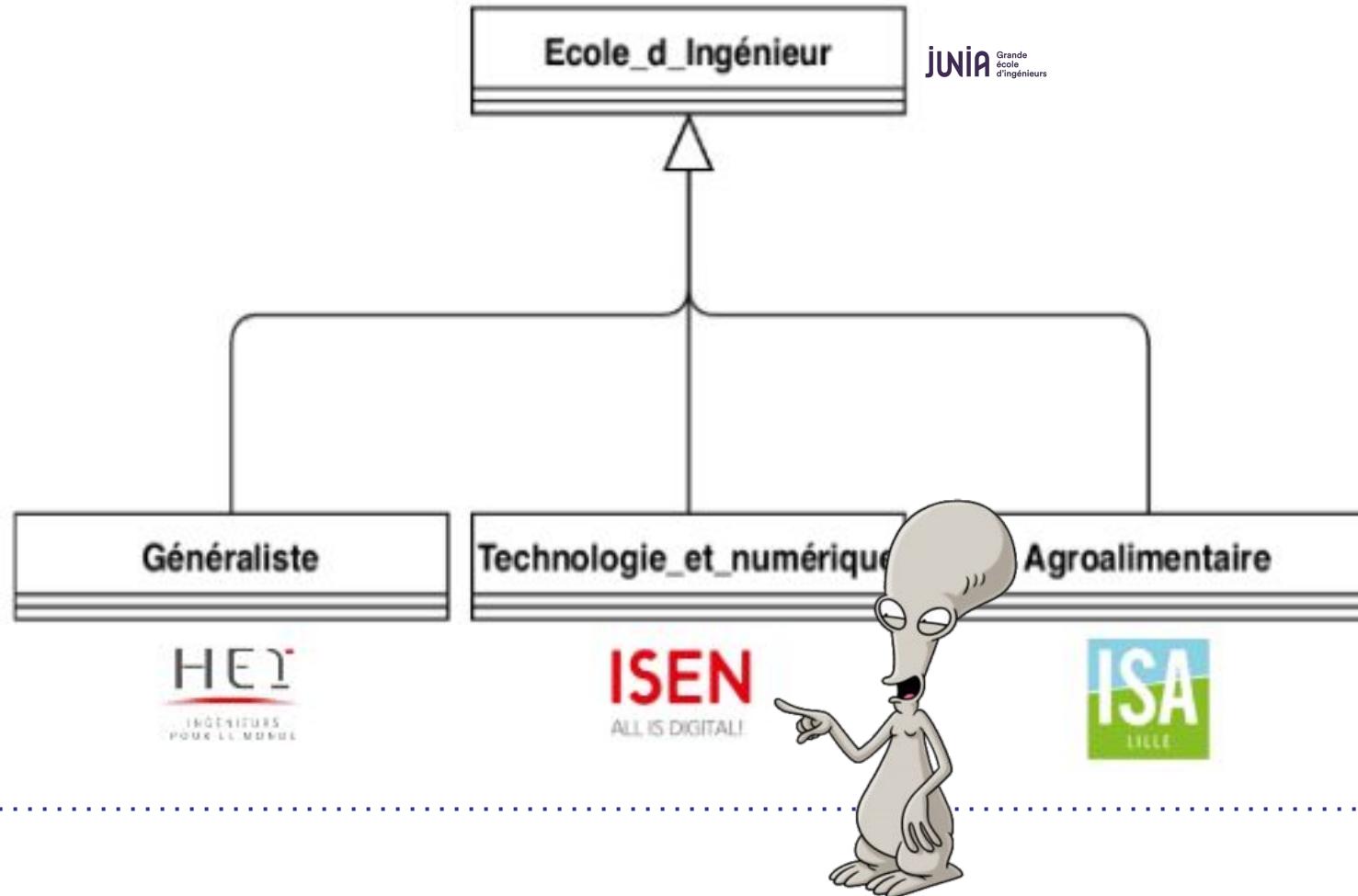


Back to the lecture / Retour au cours



2.2.
2nd fundamental concept

Inheritance / Héritage



Inheritance / Héritage

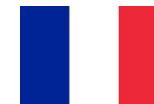
Superclass



Generalization
Spécialization

Subclass

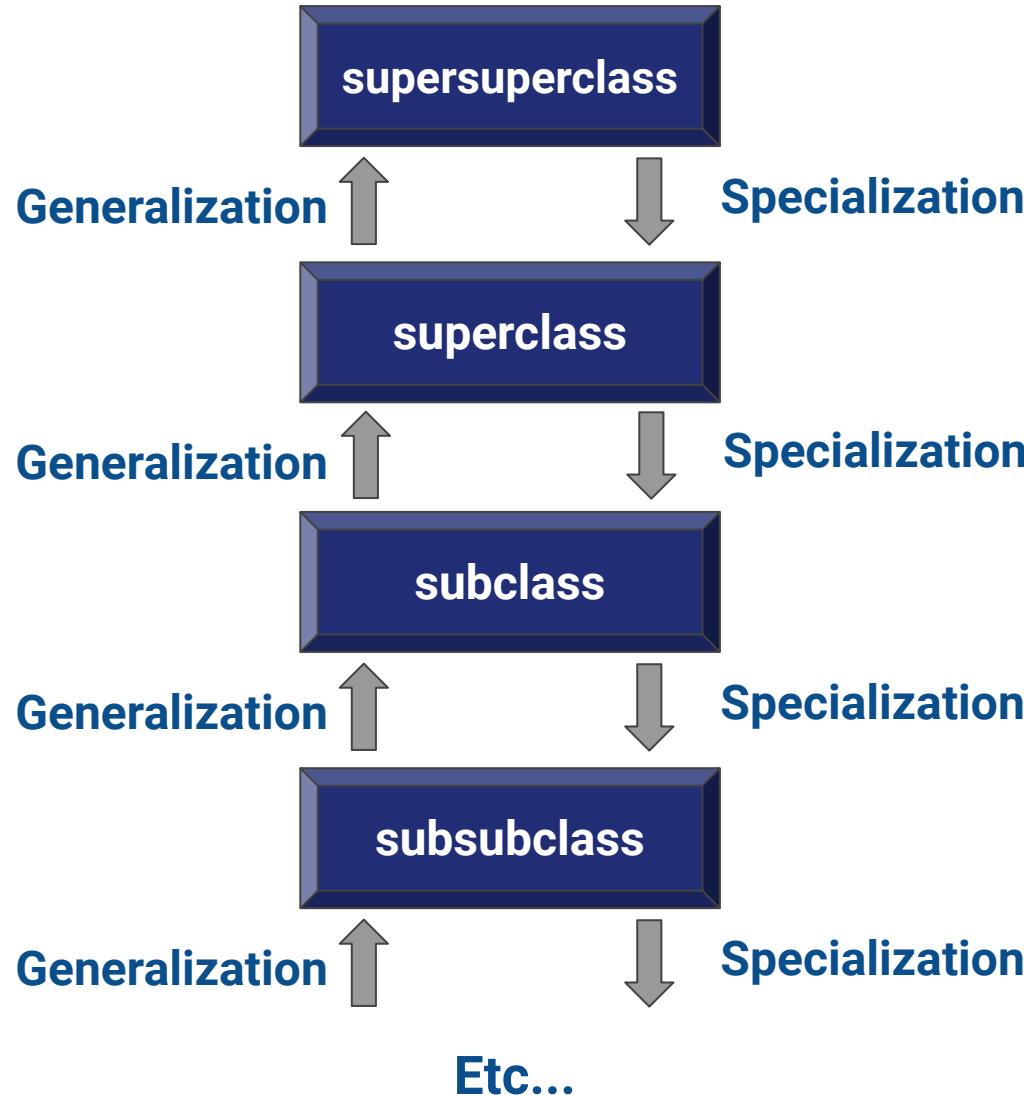
Classe Mère
<=> Classe de Base



Generalisation
Spécialisation

Classe Fille
<=> Classe dérivée

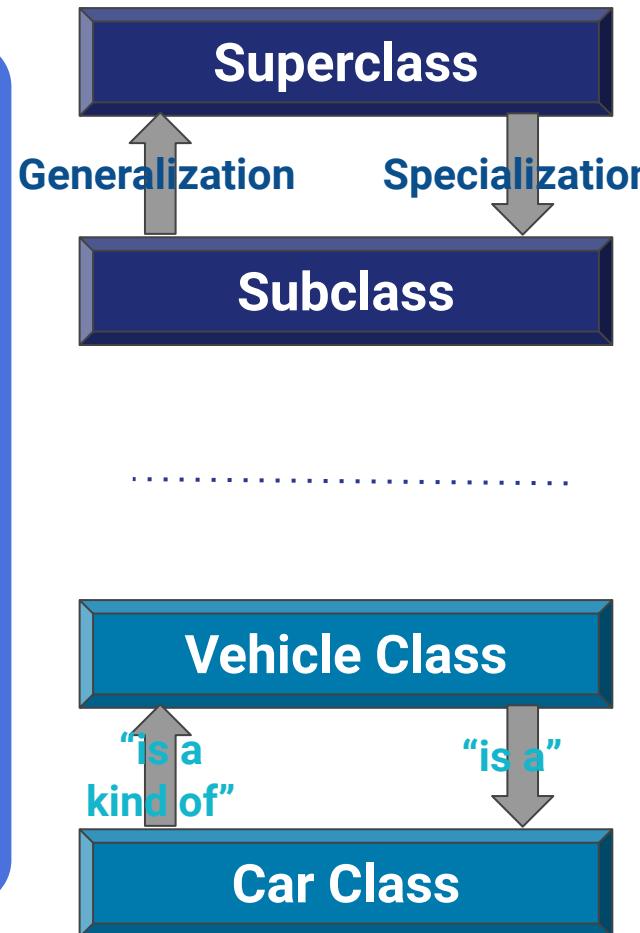
generation after generation



Inheritance / Héritage

Main points (1/2)

- An inheritance link says: "**is a**" or "**is a kind of**",
 - / Une relation d'héritage se dit : "est une" ou "est une sorte de" ;
- Inheritance allows you to build class hierarchies,
 - / L'héritage permet de construire des hiérarchies de classes,
- Inheritance can be single or multiple.
 - L'héritage peut être simple ou multiple.



Multiple Inheritance / Héritage Multiple



Car



Boat

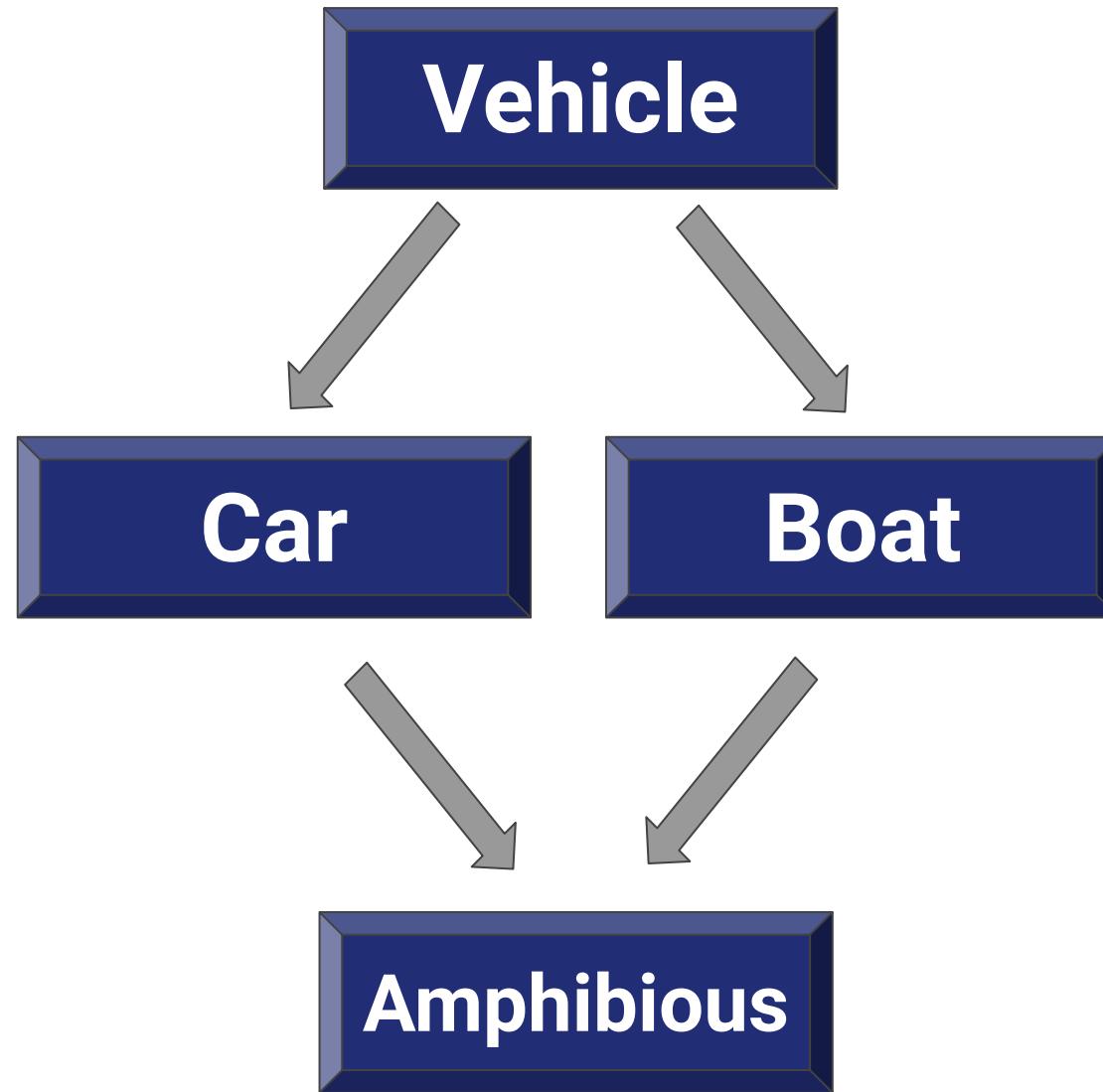
Double
Specialization

Amphibious



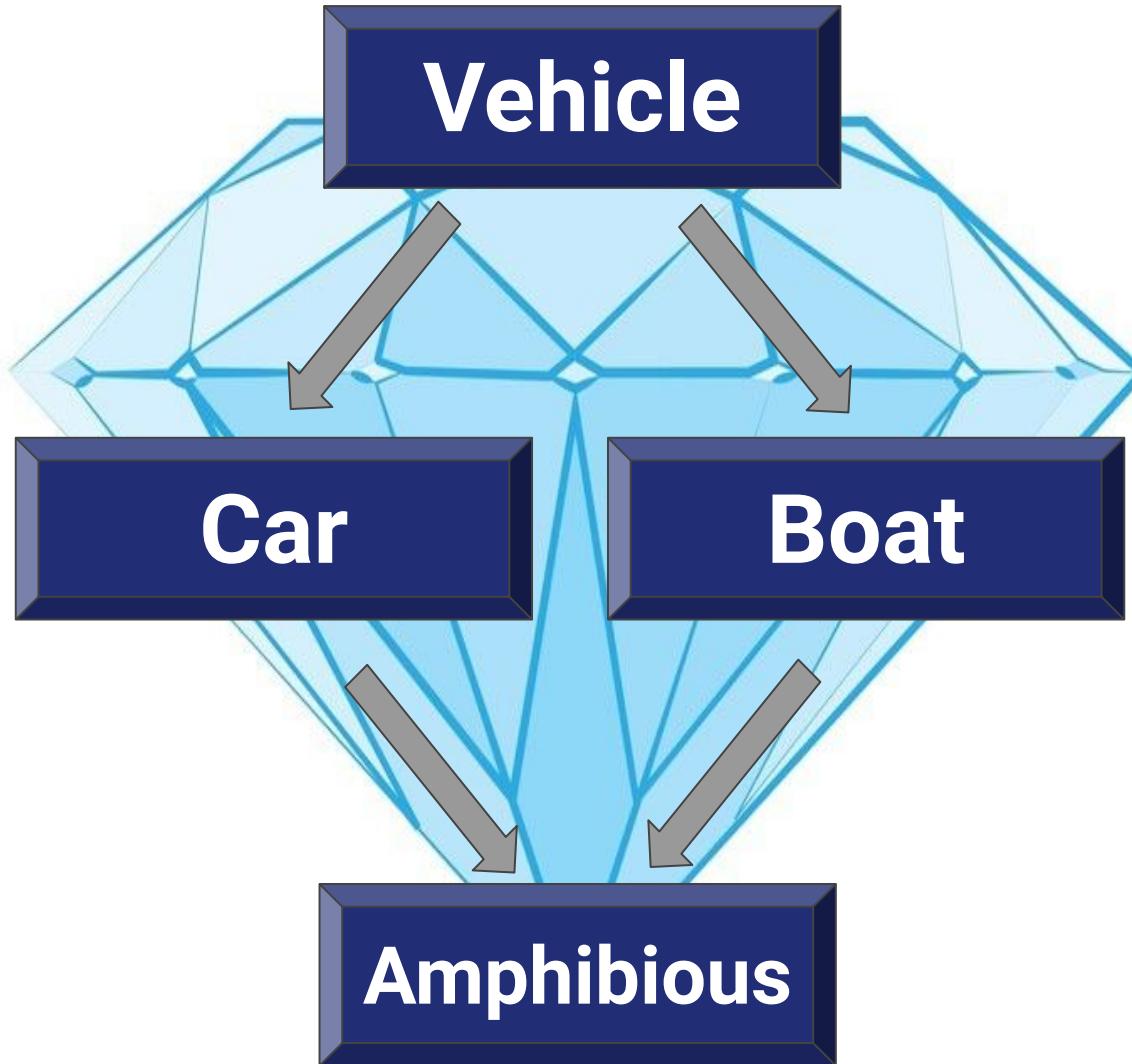
Note: we're not using UML class diagram here where inheritance relations are using an arrow targeting the super class.

Multiple Inheritance and its problems...



Question: What problem are we going to have with Amphibious?

Multiple Inheritance and its problems...

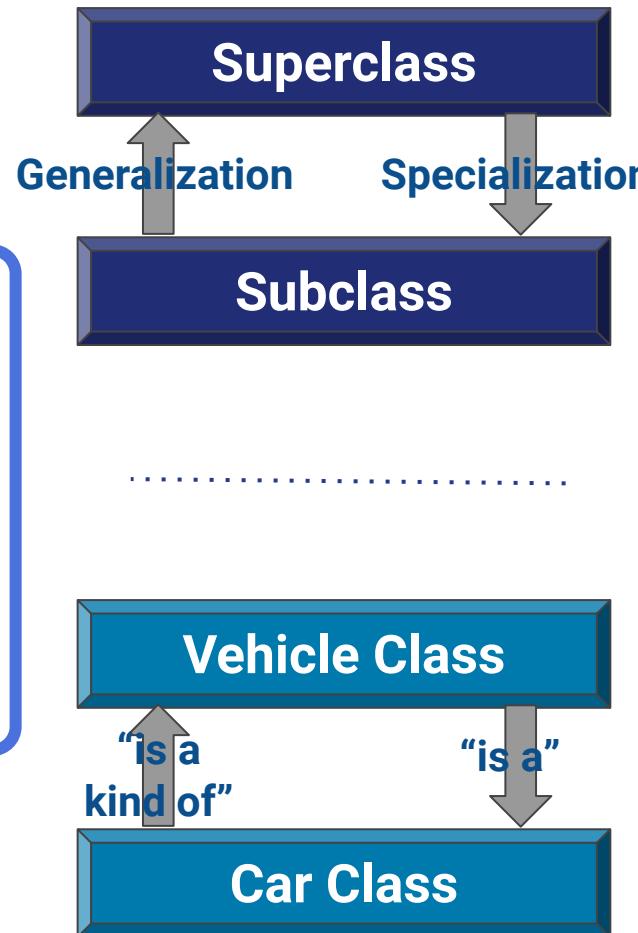


Pb: Amphibian receives **two copies** of Vehicle (data and methods).
We call it: the **diamond inheritance** / héritage en diamant.

Inheritance

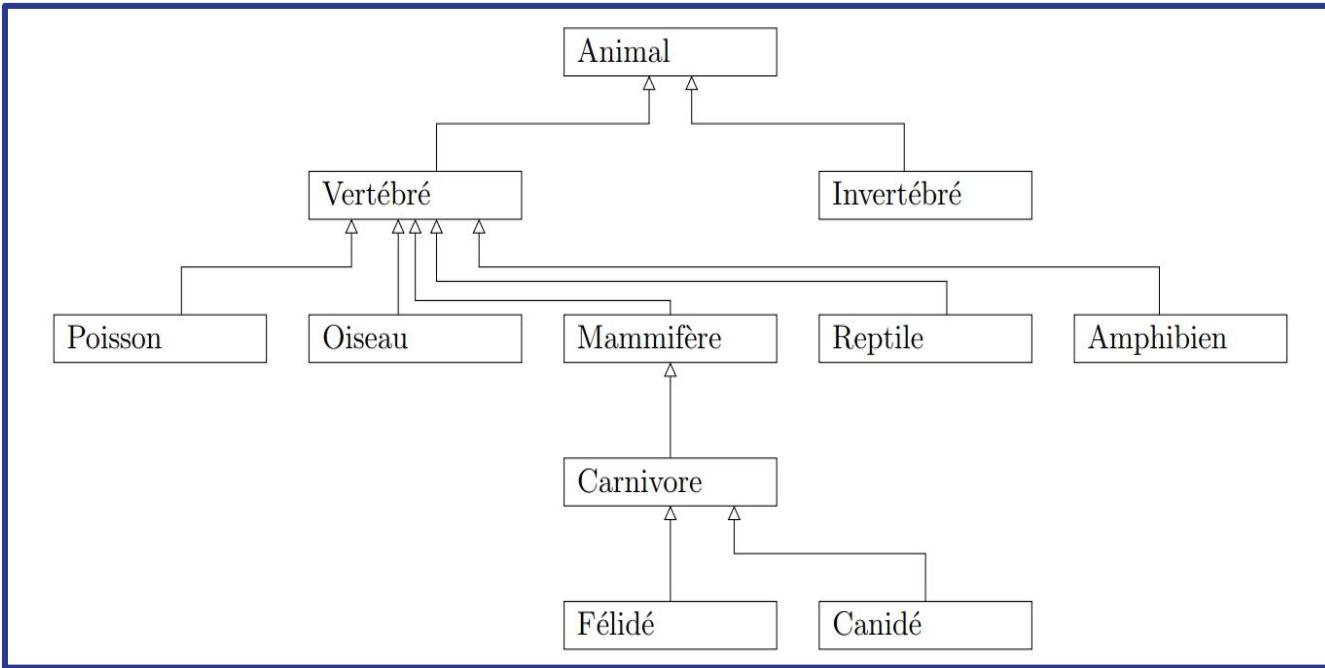
Main points (2/2)

- **There is no partial inheritance:**
 - The state and behavior (i.e., all data and methods) of the superclass are inherited in the subclass.

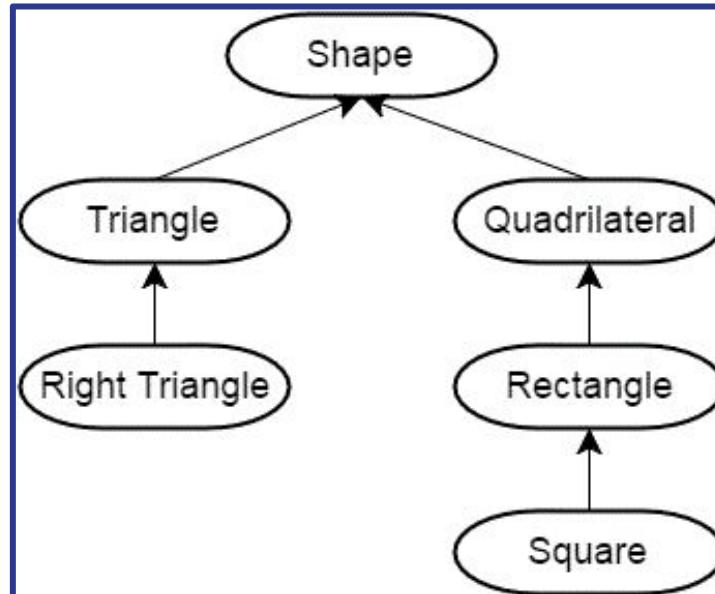


Inheritance - Examples

Generalization



Specialization



Note. In UML, the right type of arrow is the one above.



Back on Data Accessibility in JAVA



Private

Protected

Public

Data accessible in the class itself.

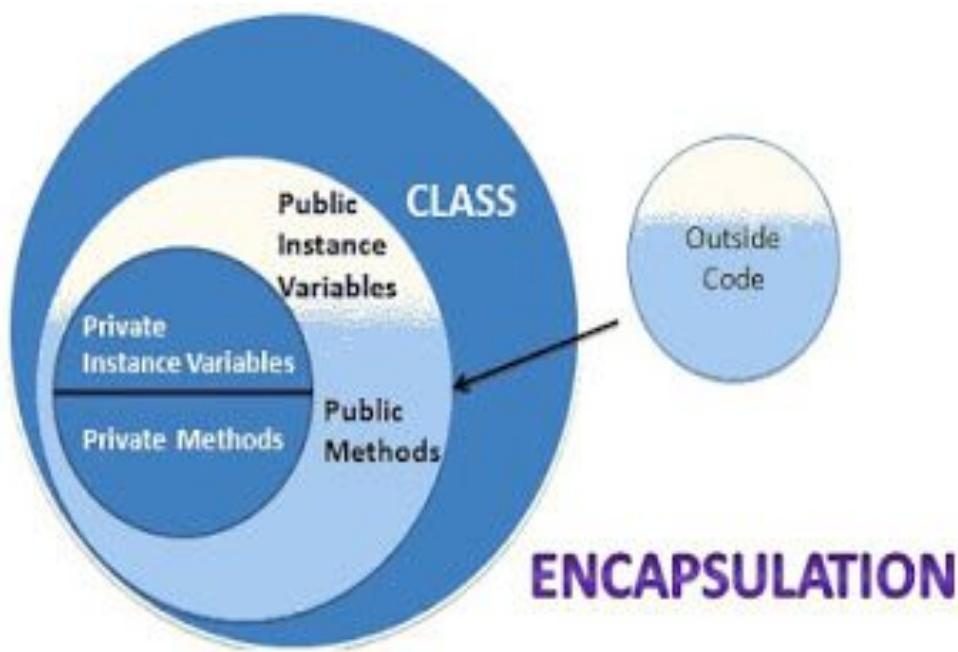
“Private” rules + accessibility for the subclasses.

Data accessibility for all

Package

(“*Friendly*”) By default but to be avoided. (\approx friend en C++)

Back on Data Accessibility in JAVA

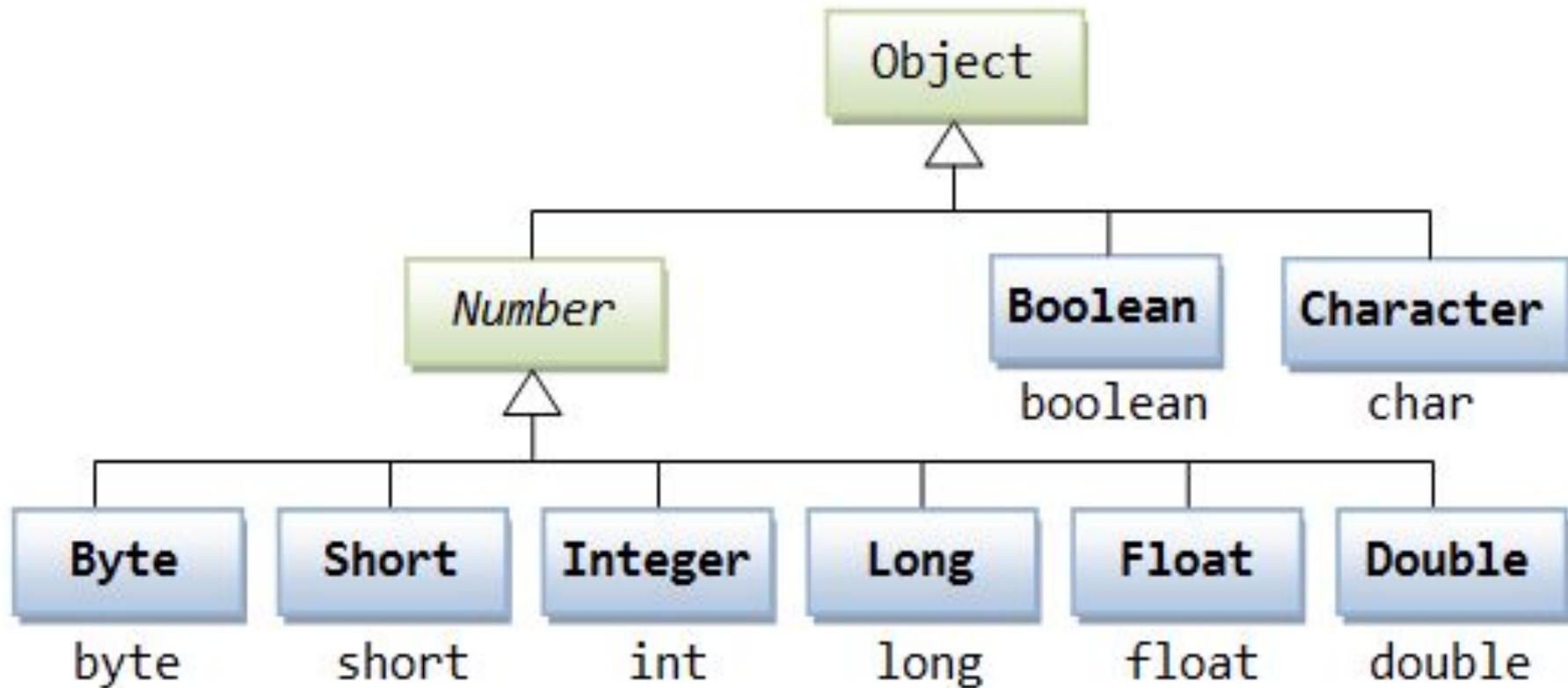


"Protected Data and Public Methods" is a good way to go ...
But the world is more complicated than that!

Examples:

- Some superclass data must remain private
- Some constructors (which are methods) are private.
 - cf. Design Pattern Singleton (later in this course).
- Non secured if inheritance.

Inheritance & JAVA



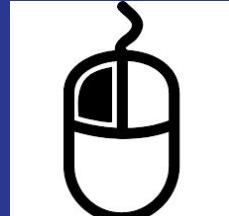
Documentation on `java.lang.Object` : <https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/Object.html>
Pic source : https://www.ntu.edu.sg/home/ehchua/programming/java/J3c_OOPWrappingUp.html

Practical Work

/ Travaux Pratiques

Practical
Work
2

(Maybe the most important one for the simulation project)



Back to the lecture



2.3.
3rd Fundamental Concept

Polymorphism /e



“Easy” (static) Polymorphism

Polymorphic Methods

- Literally:
 - (which can take) several forms.
- Methods with the same name and the same number of parameters. E.g.:
 - `sum(int a, int b, int c)` & `sum(double a, double b, double c)`,
 - `divide(int a, int b)` & `divide(double a, double b)`.

(more complicated...)

Inheritance & Polymorphism

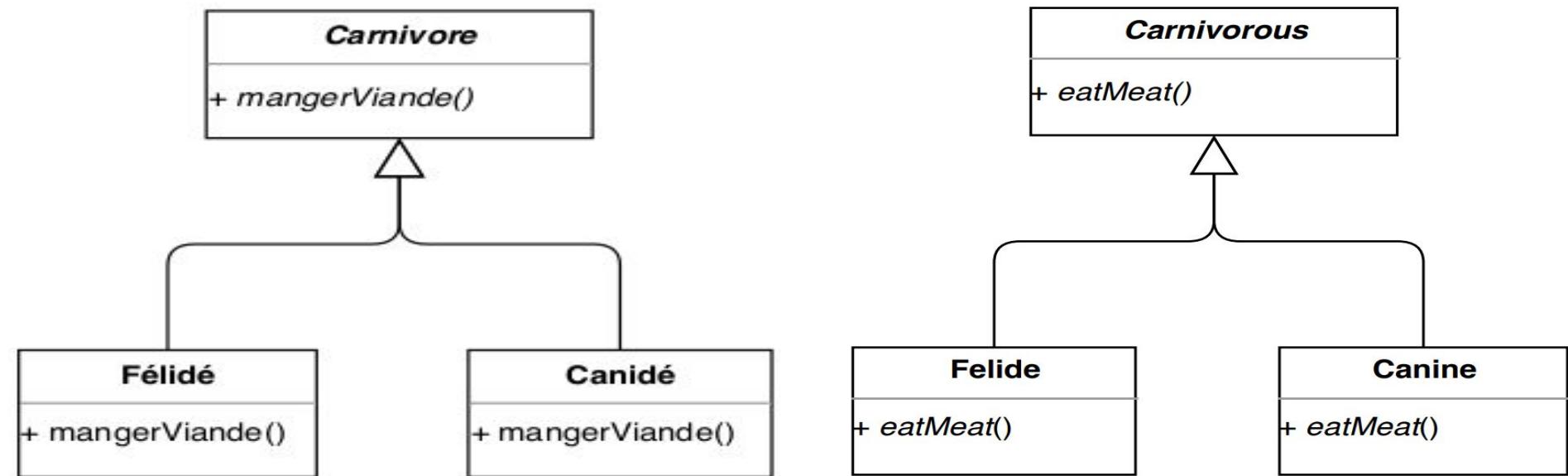
Abstract methods and classes

- **Abstract Method / Méthode abstraite**
 - Signature known / Signature connue,
 - (name & parameters) ;
 - Unknown algorithm
 - The subclasses define the algorithm (according to the class particularities);
- **Abstract Class / Classe abstraite**
 - A class is abstract if it has at least one abstract method.
 - An abstract class cannot be instantiated.

(more complicated...)

Inheritance & Polymorphism

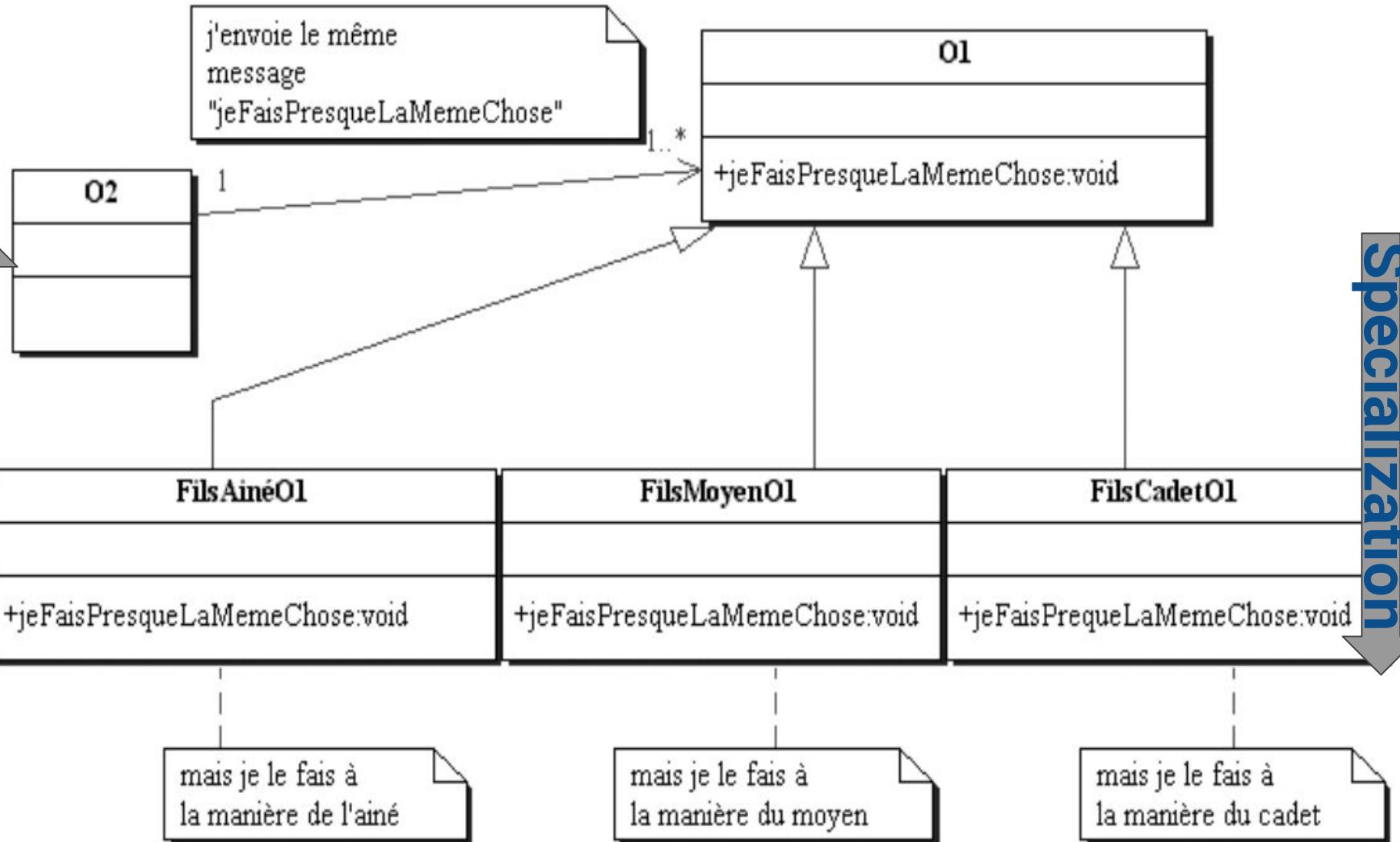
Example



Inheritance & Polymorphism - Example

Generalization

Specialization



Polymorphism & JAVA

```
MainClass.java
1 public class MainClass {
2     public static void main(String[] args) {
3         System.out.println("Hello Abstract.");
4     }
5 }
```

```
Vehicle.java
1 abstract class Vehicle {
2     /*
3      * Default Constructor
4      */
5     public Vehicle()
6     {
7         km = 0;
8     }
9     /*
10    * Methods
11   */
12     abstract void move();
13     /*
14    * Attributes
15   */
16     protected int km;
17 }
```

```
Car.java
1 public class Car extends Vehicle {
2     /*
3      * Default Constructor
4      */
5     public Car()
6     {
7         super();
8     }
9     /*
10    * Methods
11   */
12     public void move()
13     {
14         super.km += 10;
15     }
16 }
17
18 }
```

```
Bike.java
1 public class Bike extends Vehicle {
2     public Bike()
3     {
4         super();
5     }
6     /*
7      * Methods
8      */
9     public void move()
10    {
11        super.km++;
12    }
13 }
14 }
```

- **super** : call to the superclass from the subclass,
- **super()** : call of the constructor of the superclass,
- **abstract** : to declare an abstract method and an abstract class,
- **extends** : to inherit from a class.

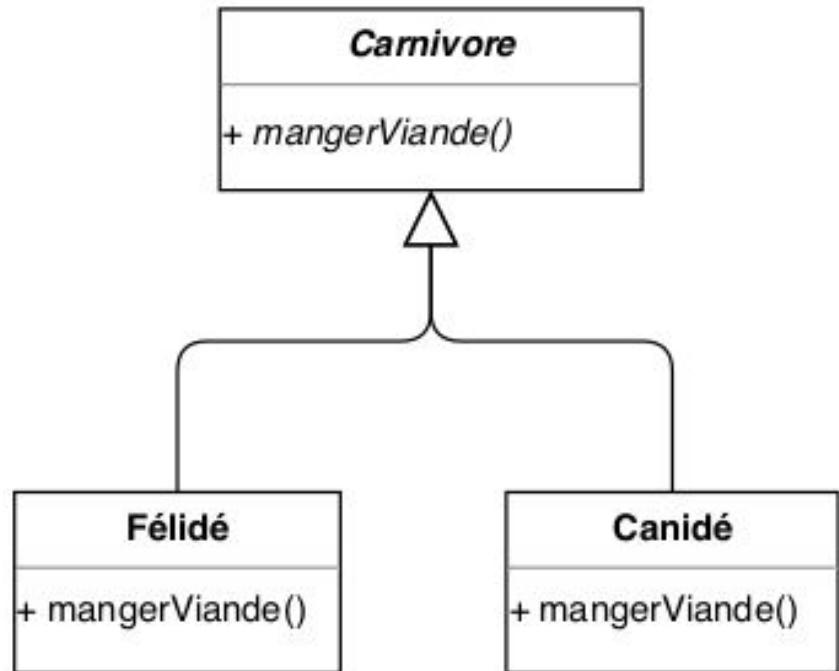
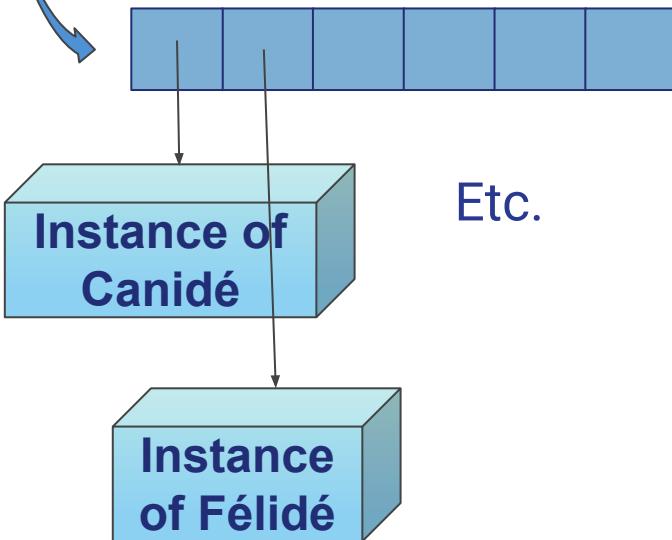
```
public final class ClasseFinal
{
    ...
}
```

Note: The **final** keyword can be used to forbid inheritance of the class.

Polymorphism - C/C++ Example

```
Carnivore ** tabAnimal;  
// [...]  
  
for (uint i = 0 ; i < AnimalNumber ; i++)  
{  
    tabAnimal[i]->mangerViande();  
}
```

tabAnimal is a pointer table on Carnivores which can be either Felide or Canide.



Polymorphism ensures that the correct `eatMeat()` method will be called (either Felide's or Canide's).

Inheritance & Polymorphism & C++

```
class Personne {  
...  
virtual void afficher(void) const  
{ cout << nom << " " << prenom; }  
...  
};
```

```
class Etudiant : public Personne {  
...  
void afficher(void) const {  
    Personne::afficher();  
    cout << " " << ecole;  
}  
...  
};
```

- **superclass::** : call to the superclass from the subclass,
- **subclass : public superclass** : to inherit from a class,
- **virtual** : the method must be redefined in the **subclasses**,
- **Abstract method** : called “pure virtual function” (**ends with ‘=0’**).

About the `@Override` keyword in Java

- '`@Override`' is written right before a child class method signature for **Dynamic Polymorphism**.
- If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, BUT if an object of the subclass is used to invoke the '`Overridden`' method, then the version in the child class will be executed. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

```
class SuperClass {
    void show()
    { System.out.println("SuperClass's method"); }
}

// Inherited class
class SubClass extends SuperClass {
    @Override
    void show()
    { System.out.println("Child's method"); }
}
```

class Main {
 Run | Debug
 public static void main(String[] args) {
 // If a Parent type reference refers
 // to a Parent object, then Parent's
 // show is called
 SuperClass obj1 = new SuperClass();
 obj1.show();
 // If a Parent type reference refers
 // to a Child object Child's show()
 // is called. This is called RUN TIME
 // POLYMORPHISM. object
 SuperClass obj3 = new SubClass();
 obj2.show();
 } }

Reference
(not the object)

Question in C++

```
class EcoleIngenieur
{
public:
    void M1() { cout << "M1
                    EcoleIngenieur" << endl; }
    virtual void M2() { cout << "M2
                    EcoleIngenieur" << endl; }
};

class EcoleJUNIA :
    public EcoleIngenieur
{
public:
    void M1() {
        cout << "M1 EcoleJUNIA" << endl; }
    void M2() {
        cout << "M2 EcoleJUNIA" << endl; }
};
```

```
int main()
{
    EcoleIngenieur a;
    a.M1();
    a.M2();

    EcoleJUNIA b;
    b.M1();
    b.M2();

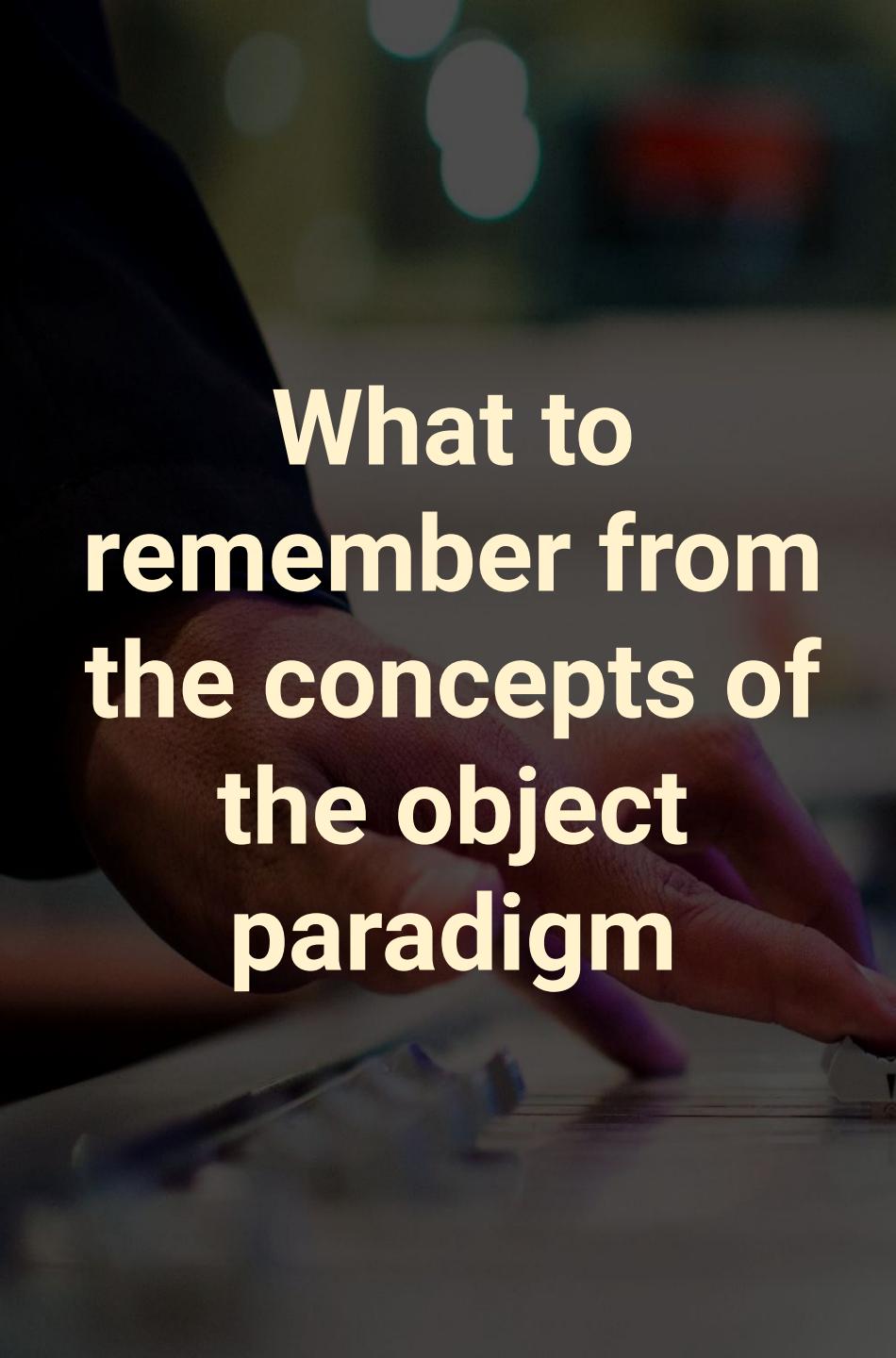
    EcoleIngenieur * pa
    = new EcoleJUNIA;
    pa->M1();
    pa->M2();
}
```

— What the “cout”
will display?

```
class EcoleIngenieur
{
public:
    void M1() { cout << "M1
                    EcoleIngenieur" << endl; }
    virtual void M2() { cout << "M2
                    EcoleIngenieur" << endl; }
};

class EcoleJUNIA :
    public EcoleIngenieur
{
public:
    void M1() {
        cout << "M1 EcoleJUNIA" << endl; }
    void M2() {
        cout << "M2 EcoleJUNIA" << endl; }
};
```

```
int main()
{
    EcoleIngenieur a;
    a.M1(); // M1 de EcoleIngenieur
    a.M2(); // M2 de EcoleIngenieur
    EcoleJUNIA b;
    b.M1(); // M1 de EcoleJUNIA
    b.M2(); // M2 de EcoleJUNIA
    EcoleIngenieur * pa
        = new EcoleJUNIA;
    pa->M1(); // M1 de EcoleIngenieur
    pa->M2();
    // M2 de EcoleJUNIA ("virtual")
    EcoleIngenieur * Ecl = &b;
    Ecl->M1(); // M1 of EcoleIngenieur
    Ecl->M2();
    // M2 of EcoleJUNIA (car "virtual")
}
```



What to remember from the concepts of the object paradigm

- Definition of :
 - **object,**
 - **instance,**
 - **class;**
 - The 3 fundamental concepts:
 - **Encapsulation,**
 - **Inheritance,**
 - **Polymorphism.**
-

3.

Unified Modeling Language (UML)

3.1. History and Overview

Once upon a time...the

UML

SECRETS
d'HISTOIRE


- Early 80' : Objects escape from laboratories
 - C++ is born;
- 1980 --> 1994 : Birth of object-oriented methods
 - e.g. OMT (Object Modeling Technique) ;
- 1994: Beginning of Unified Methods
 - UM: Unified Methods (by Rational Software) ; 
- 1995: UML 0.8;
- 1996: UML 1.0 normalised by the Object Management Group ; 
OBJECT MANAGEMENT GROUP
- ...
- Since 2017 (Déc) : UML 2.5.1





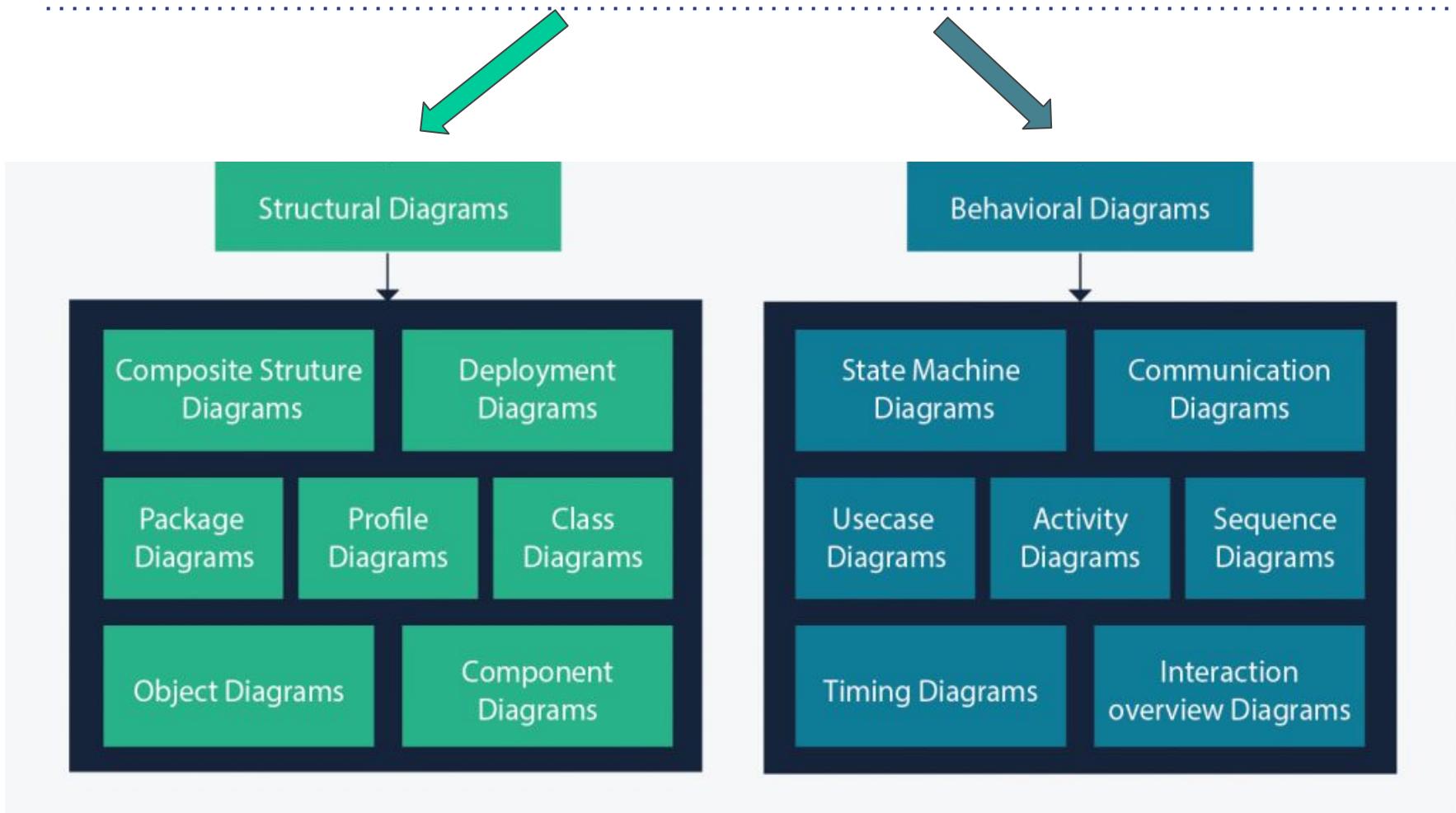
Why the ... UML ?

Unified
Modeling
Language

- For modeling software systems (and other)
 - To decompose **large scale** system
 - To help **the communication between developers**
 - e.g., with the use of different programming languages
 - Use of object-oriented software systems
 - Evolving “before, during, after” the development process.
-

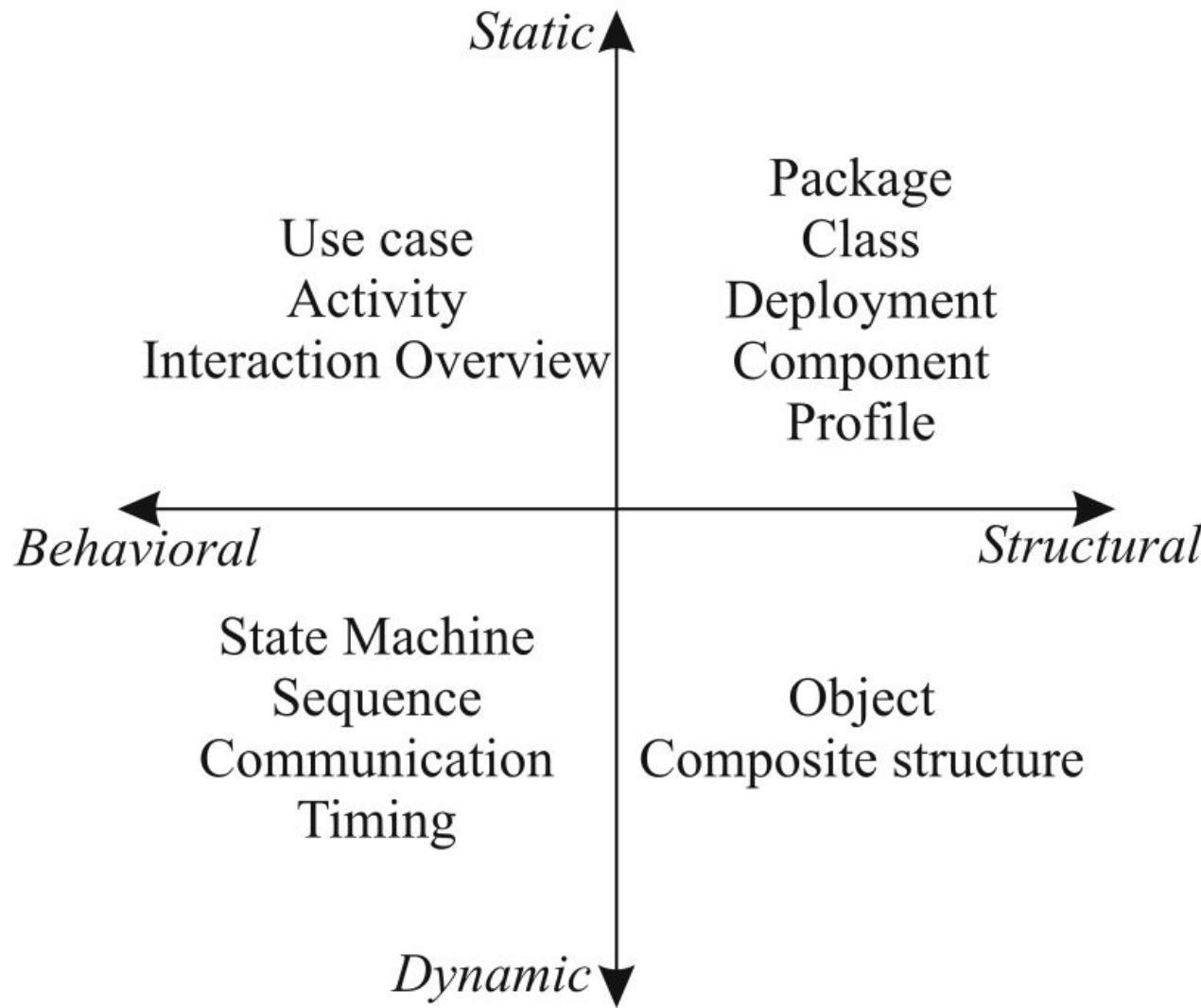
The UML today (Version 2.5)

14 diagrams, 2 types: **structural or behavioral**



The UML today (Version 2.5)

14 diagrams, 2 types: **structural** or **behavioral**



UML in this course

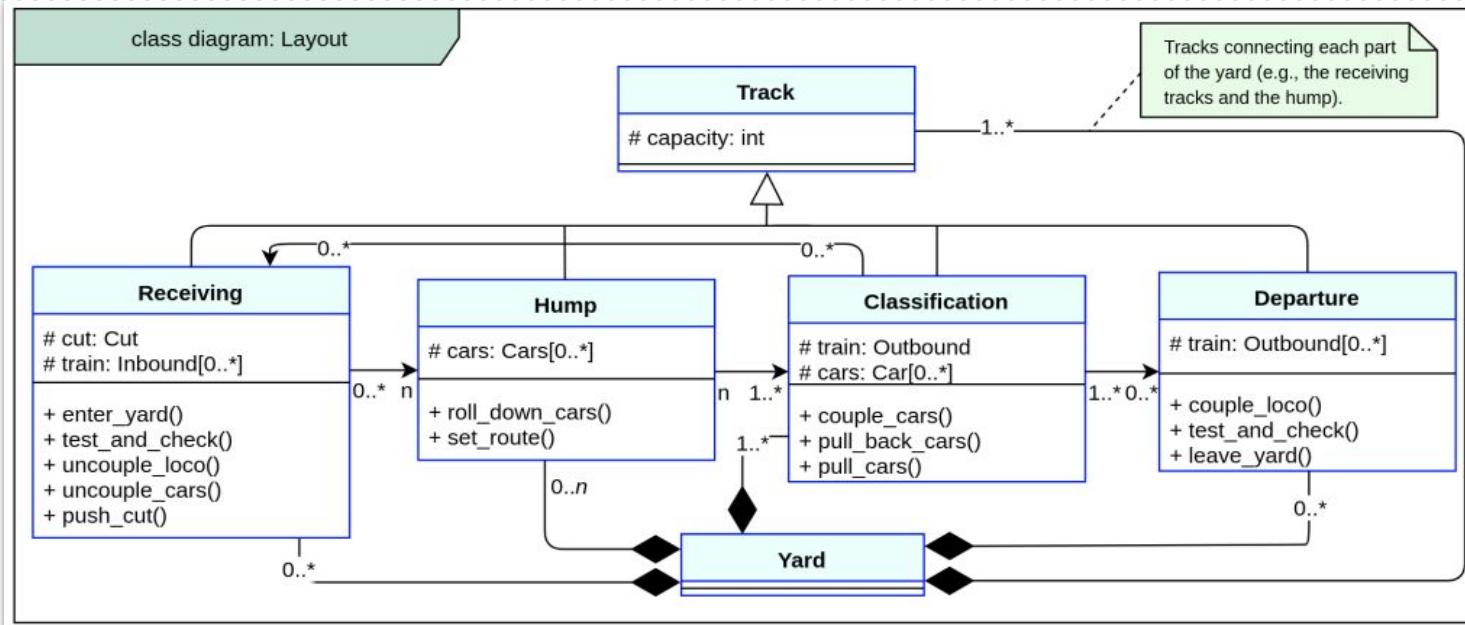


We will study (more or less in detail) the following diagrams:

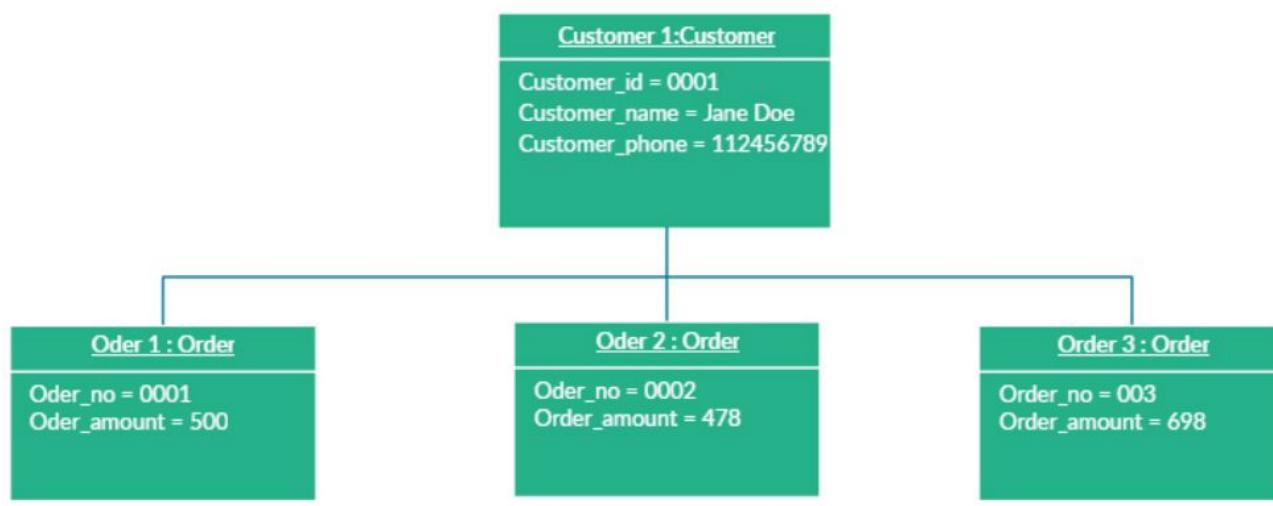
- ***class diagrams*** / diagrammes de **classes**
- ***object diagram*** / diagrammes d'**objets**
- ***state machine diagram*** / diagramme d'**états-transitions**
- ***Use case diagram*** / diagramme de **cas d'utilisation**
- ***sequence diagram*** / diagramme de **séquence**
- ***activity diagram*** / diagramme d'**activités**

Quick overview of the 14 UML diagrams

Class Diagram

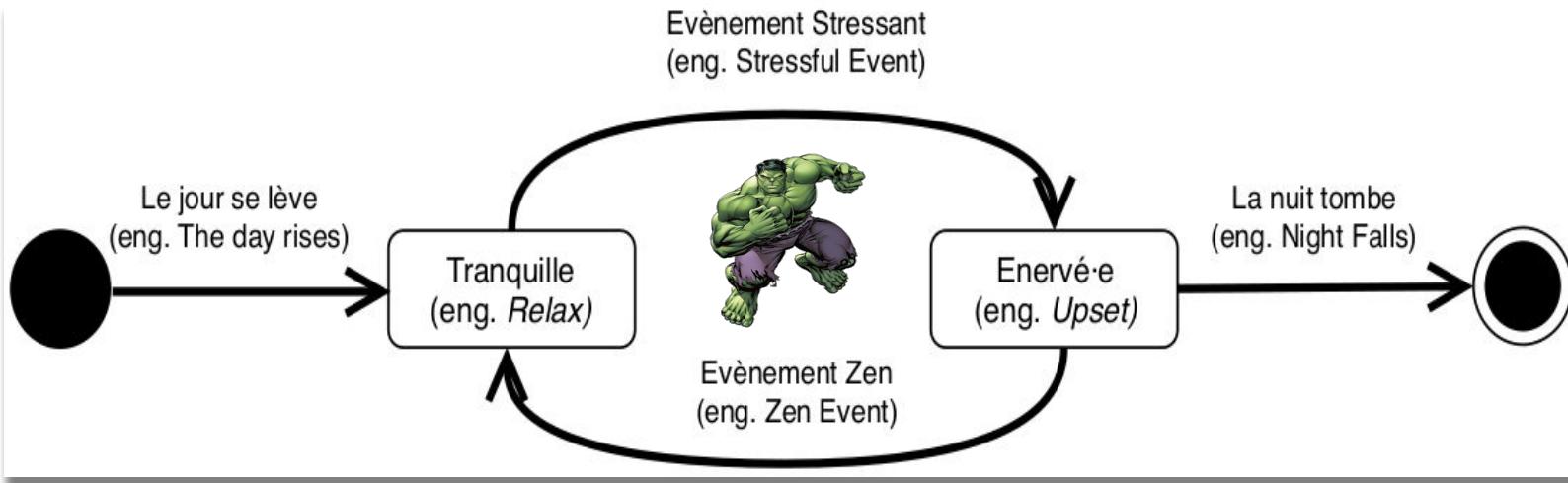


Object Diagram

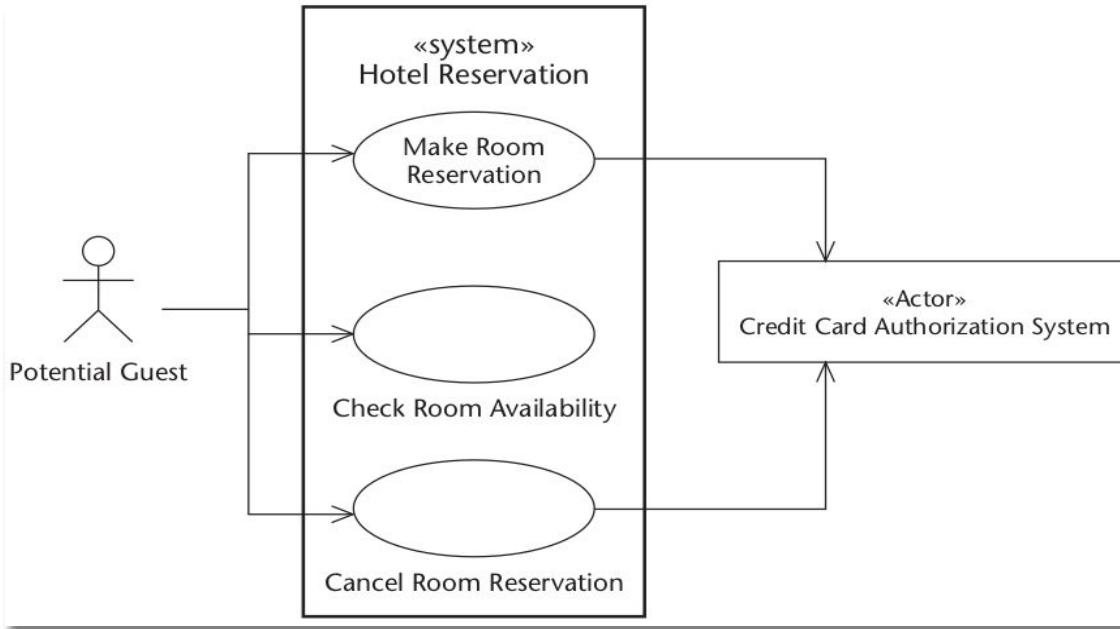


Quick overview of the 14 UML diagrams

State Machine Diagram

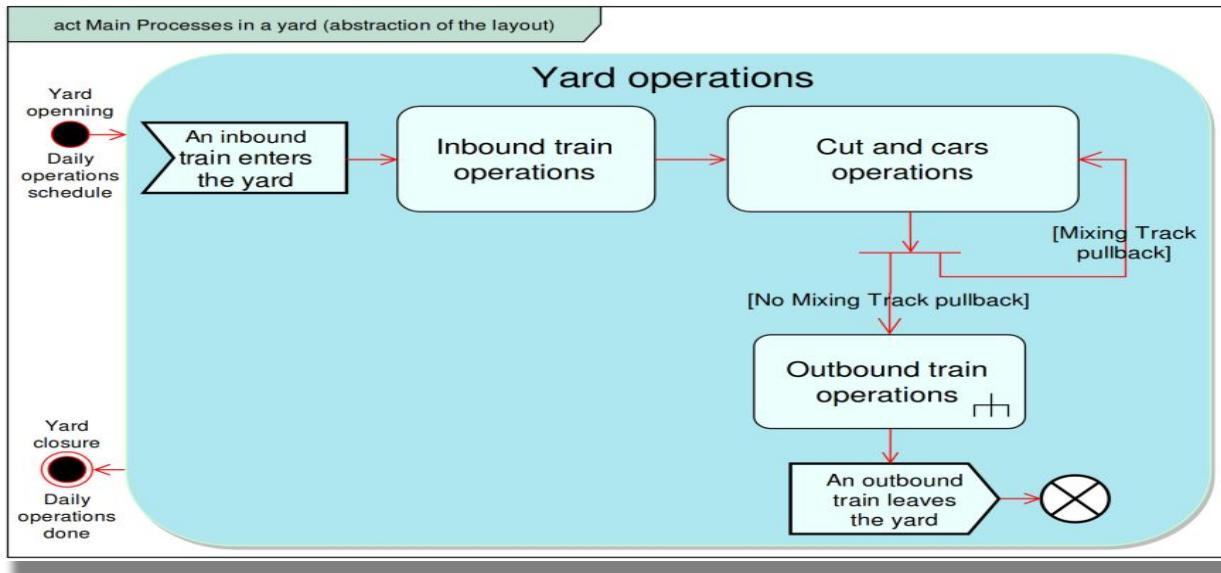


Use case Diagram

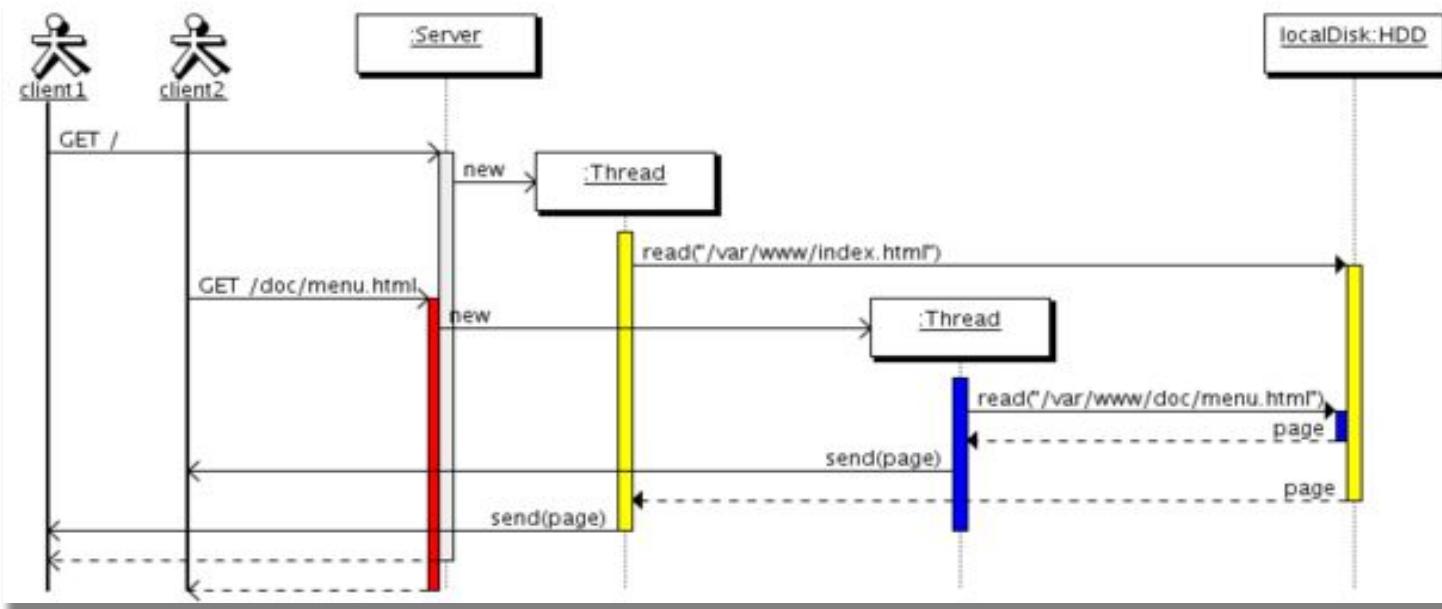


Quick overview of the 14 UML diagrams

Activity Diagram



Sequence Diagram

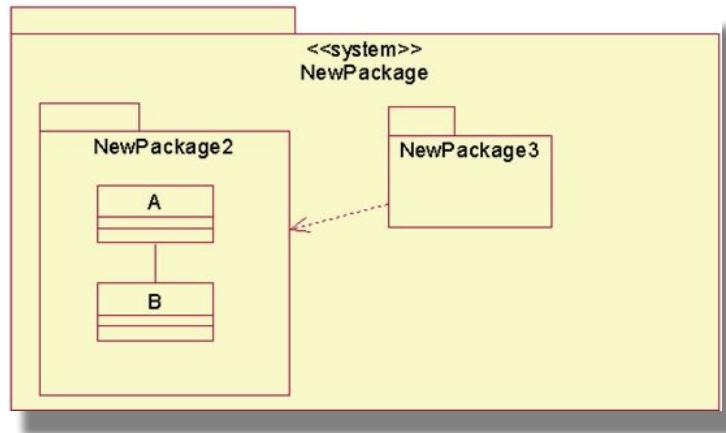




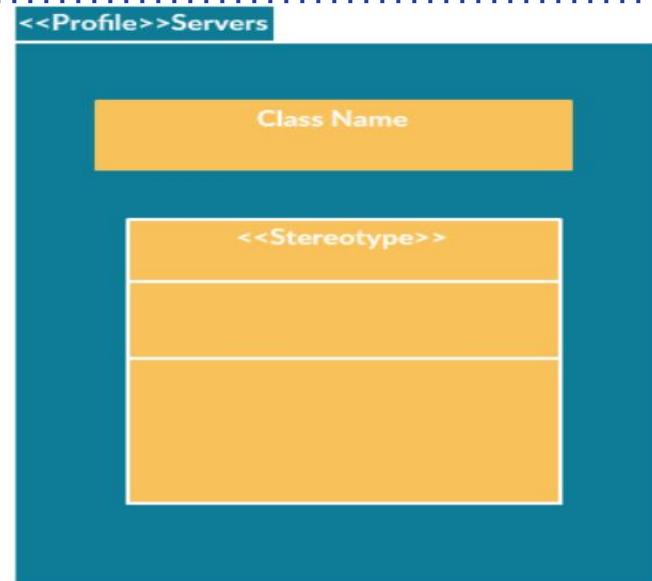
UML NOT in this course

Very very very short overview
of the rest of the diagrams.

Quick overview of the 14 UML diagrams



Package
Diagram

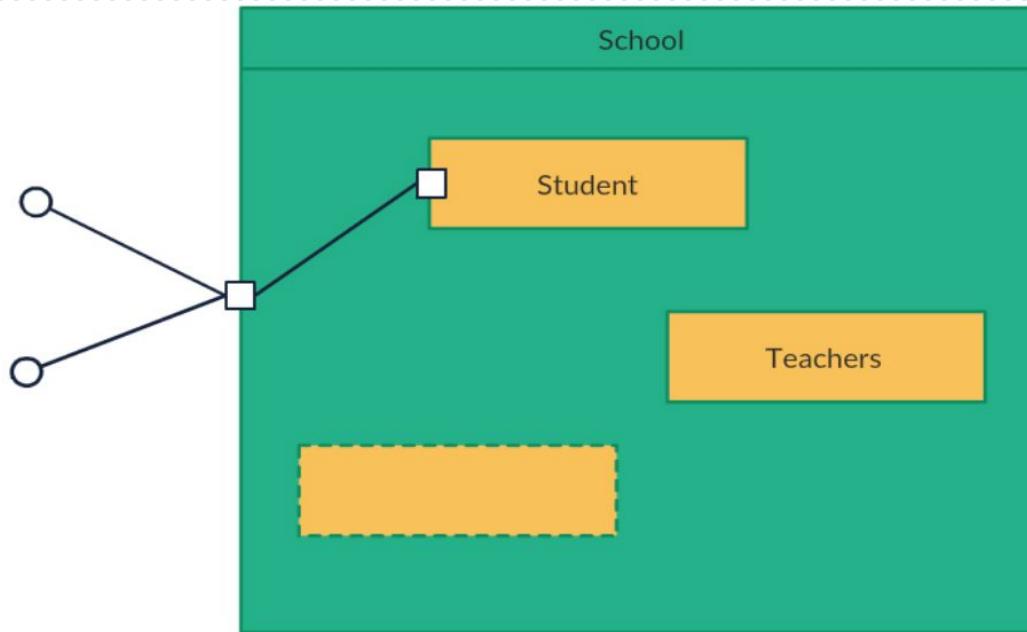


Profile
Diagram

Profile diagram, a kind of structural diagram in the Unified Modeling Language (UML), **provides a generic extension mechanism for customizing UML models for particular domains and platforms.**

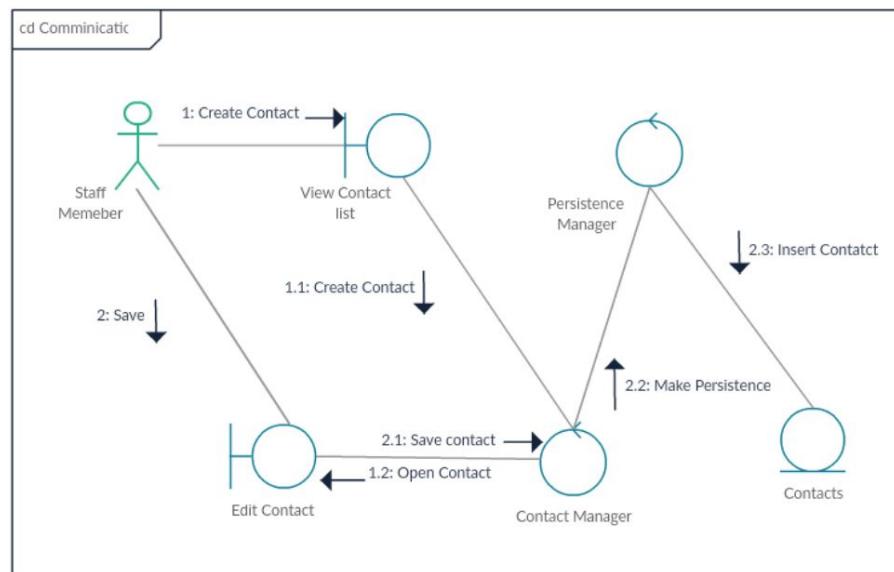
Quick overview of the 14 UML diagrams

A composite structure diagram is a UML structural diagram that contains classes, interfaces, packages, and their relationships, and that provides a logical view of all, or part of a software system. **It shows the internal structure** (including parts and connectors) **of a structured classifier or collaboration**.



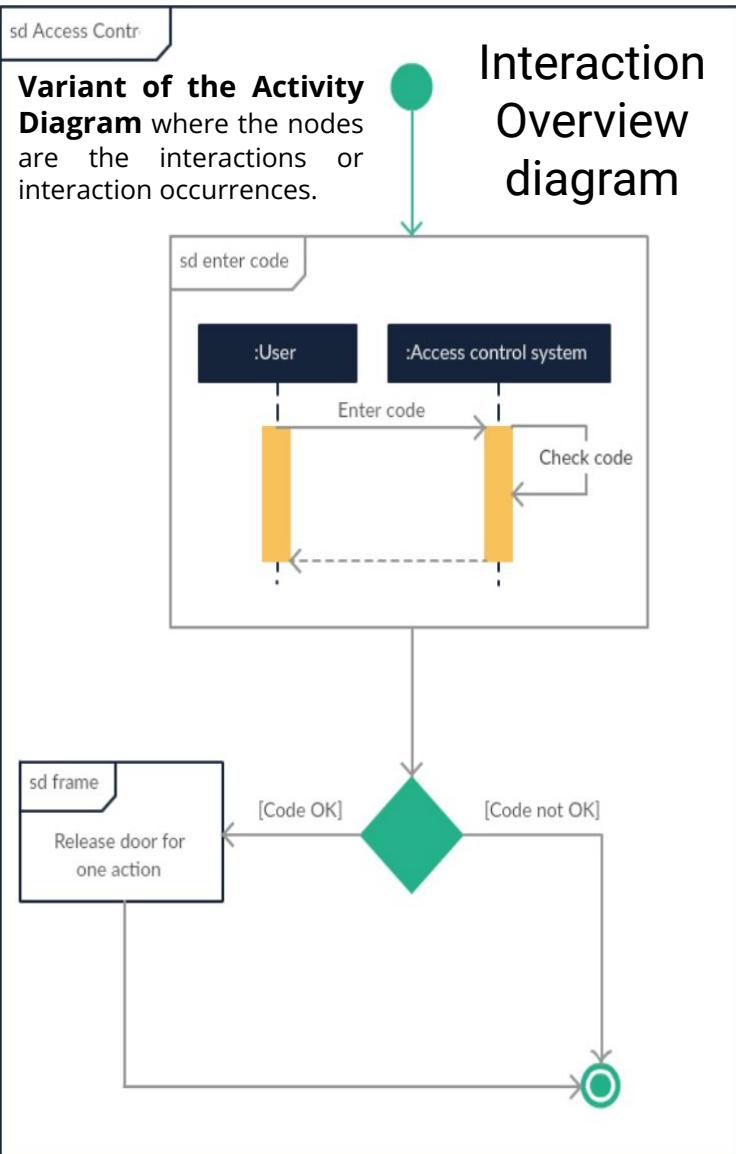
Composite
Structure
Diagram

UML communication diagrams, like the sequence diagrams - a kind of interaction diagram, **shows how objects interact**. A communication diagram is an extension of object diagram that shows the objects along with the messages that travel from one to another.



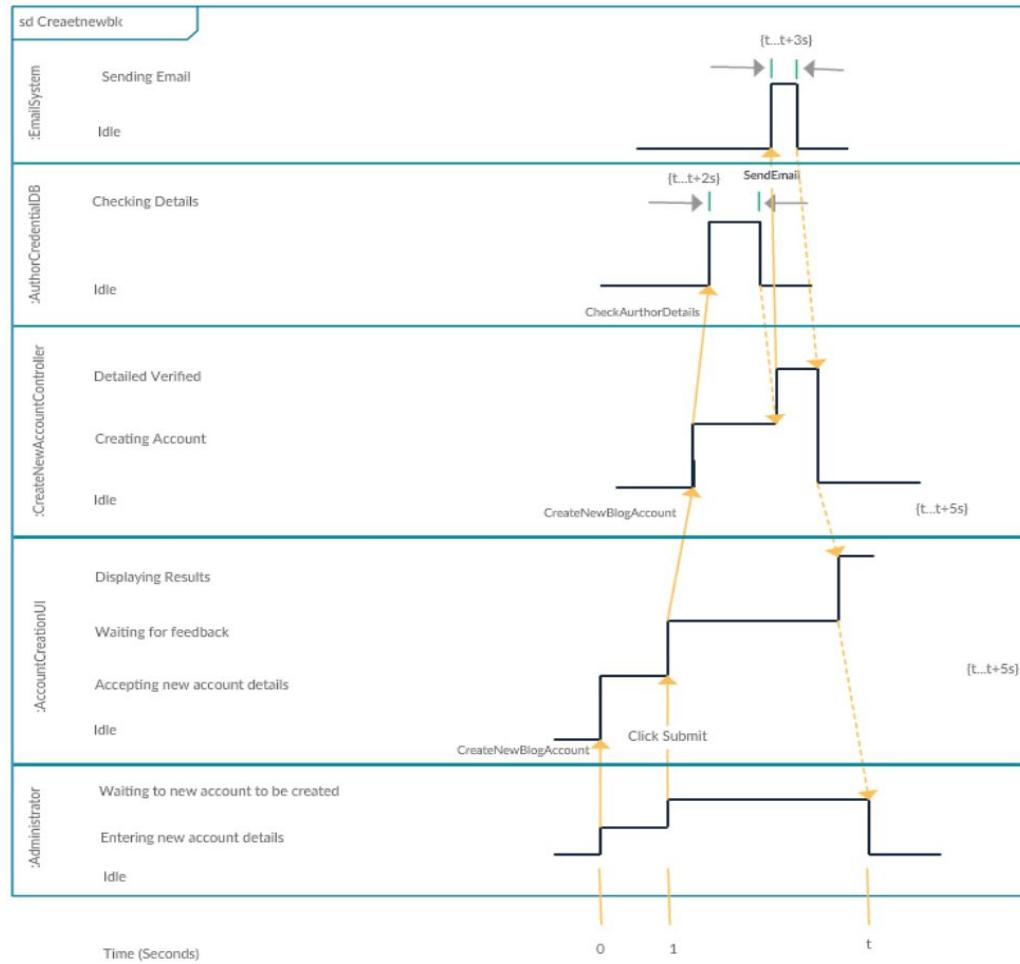
Communication
Diagram

Quick overview of the 14 UML diagrams



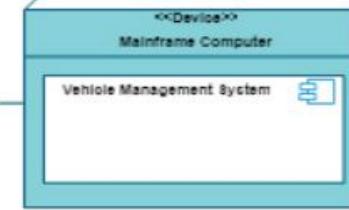
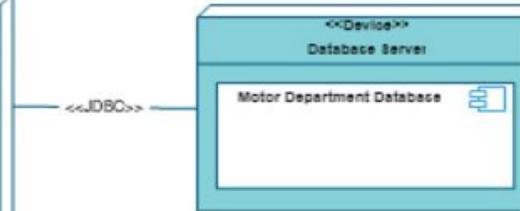
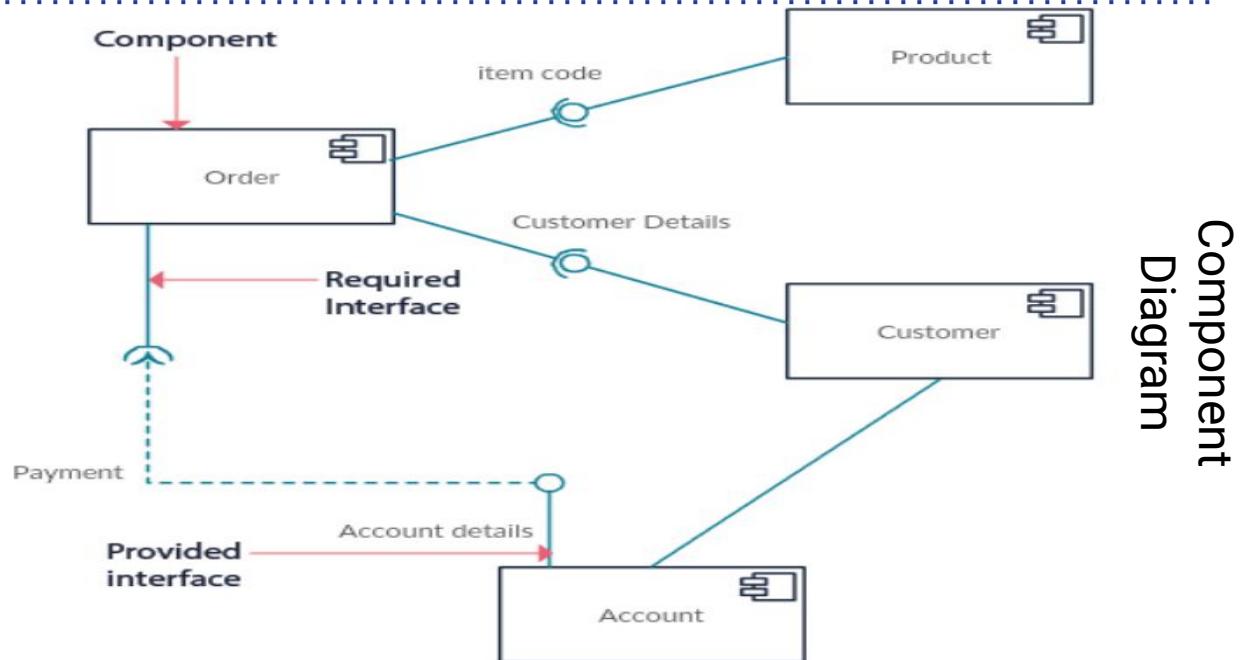
Timing Diagram

Timing diagrams are UML interaction diagrams used to **show interactions** when a primary purpose of the diagram is to reason **about time**.



Quick overview of the 14 UML diagrams

Component diagrams are used in **modeling the physical aspects of object-oriented systems** that are used for visualizing, specifying, and documenting component-based systems and also for constructing executable systems through forward and reverse engineering.



A UML deployment diagram is a diagram that shows the configuration of run time processing nodes and the components that live on them. Deployment diagrams is a kind of structure diagram used in modeling the physical aspects of an object-oriented system. They are **often be used to model the static deployment view** of a system (topology of the hardware).

Component Diagram
Deployment Diagram

Concept Object & IT Project.

UML & IT Projects

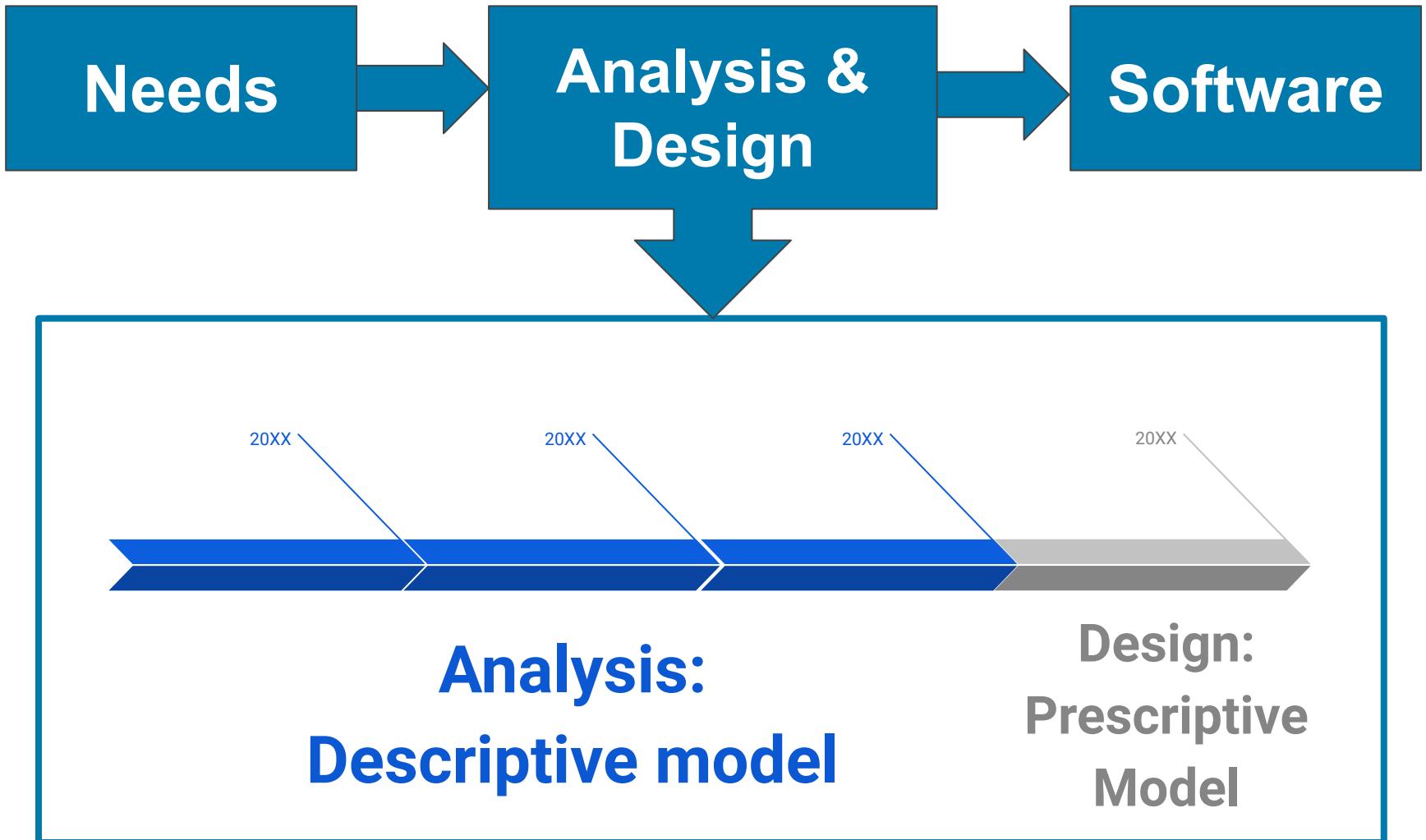


2 methods :

- The procedural approach,
- The object approach.

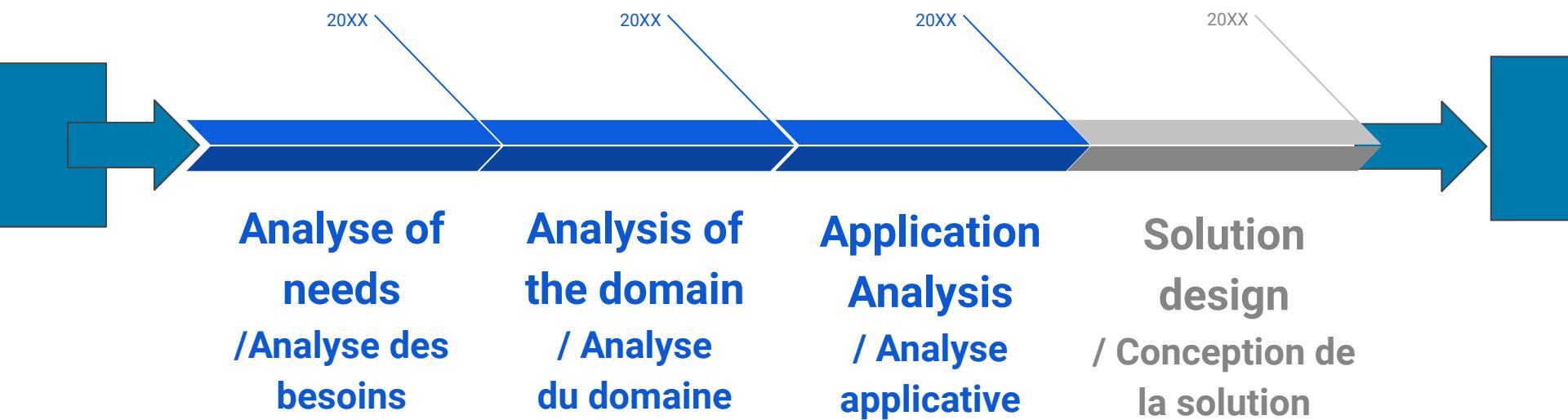
Concept Object & IT Project.

UML & IT Projects



Concept Object & IT Project.

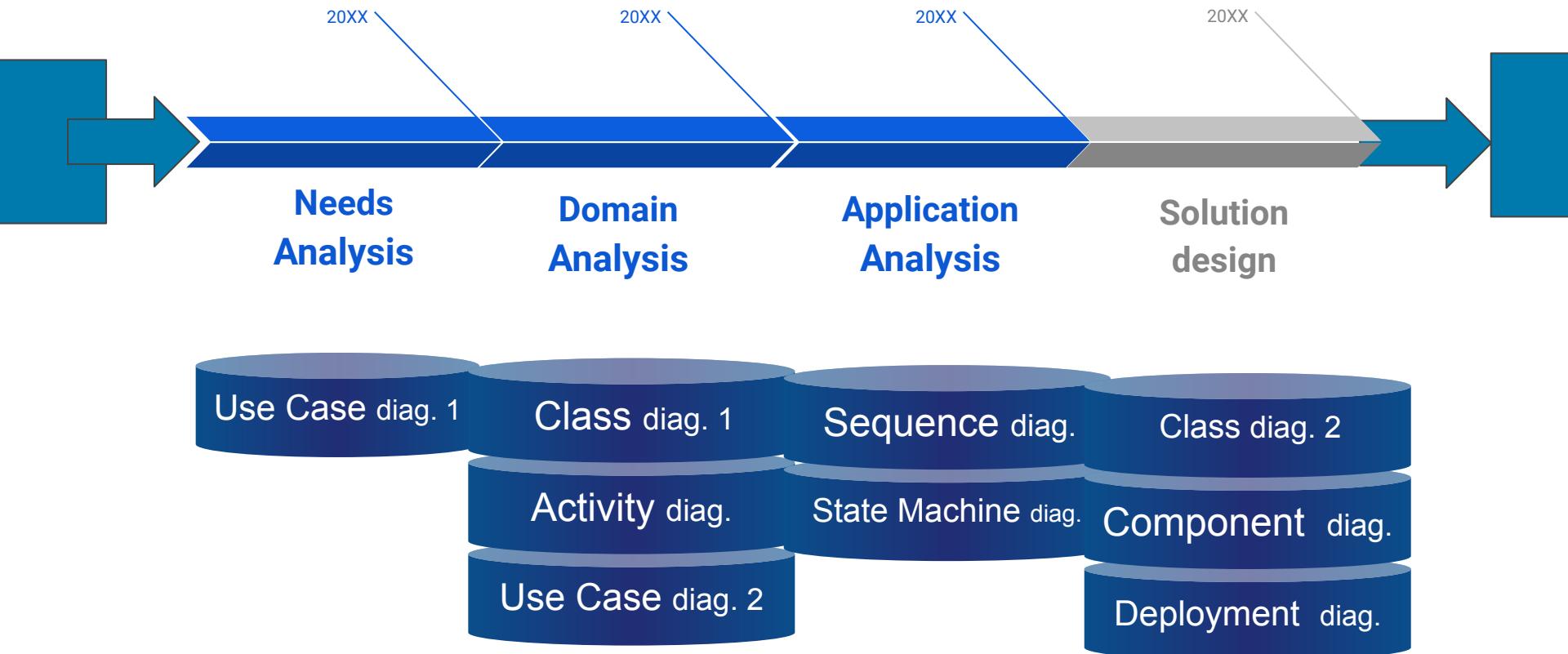
UML & IT Projects



=> We will see UML diagrams for each of these steps according to the object paradigm.

Concept Object & IT Project.

UML & IT Projects



Example: Stages & Diagrams of an IT project

3.

Unified Modeling Language (UML)

3.2.

Class Diagram

(/ Diagramme De Classe)

Class diagram

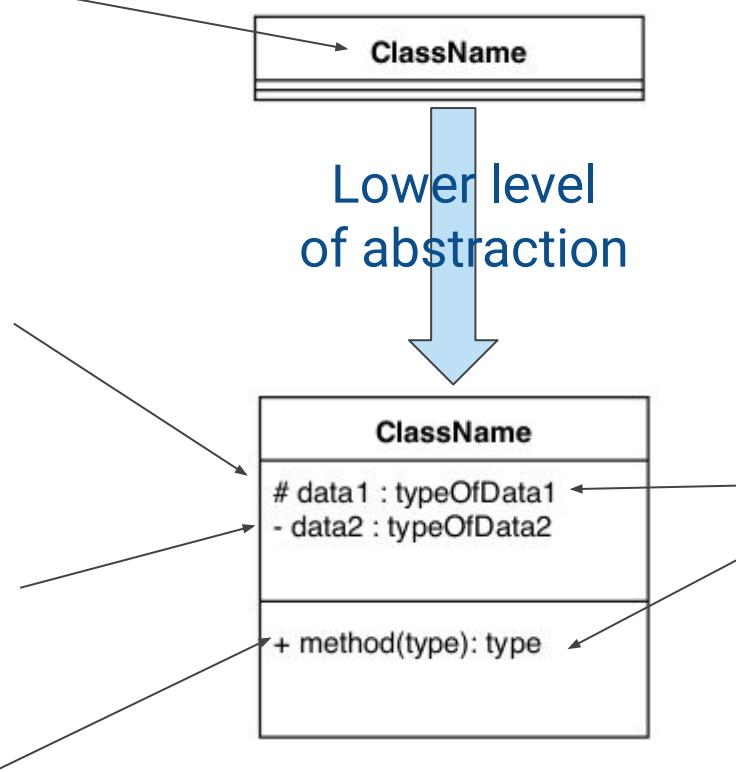
- **Central point of object-oriented development,**
- **Static modeling,**
- Level of abstraction varying according to the stage of the project:
 - Description of all classes (always),
 - Attributes and methods (depending on the stage),
 - Links between classes (depending on the stage),
- Brings up the 3 Basic Object Concepts:
 - **Encapsulation,**
 - **Inheritance** (/ héritage),
 - **Polymorphism.**

Class diagram

Modeling a class

Class name
starting with 1
uppercase

- # “Protected” data
- “Private” data
- + “Public” Method



The data type of the attribute and the data type returned by a method.

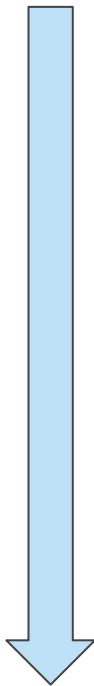
Note: attributes and methods are generally written in camelCase. (e.g., word1Mot2Mot3Mot4Mot5Mot6Mot7Etc)



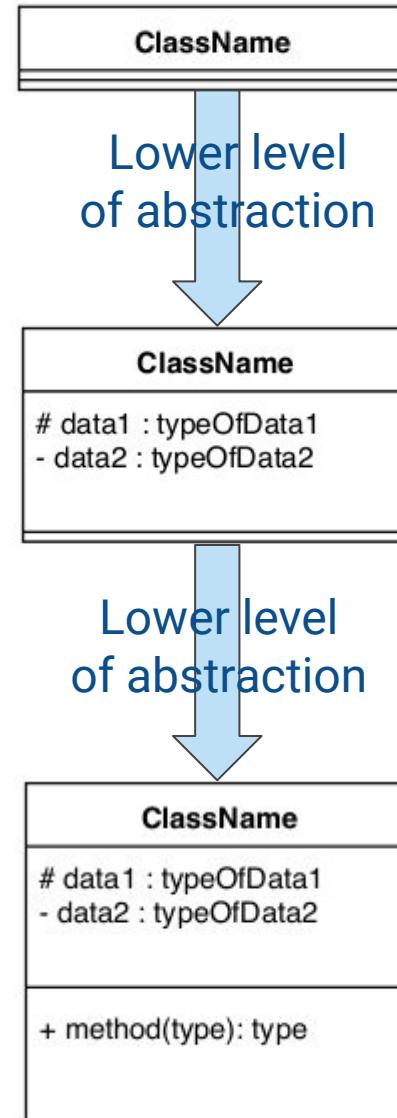
Class diagram

Modeling a class

Analysis
level

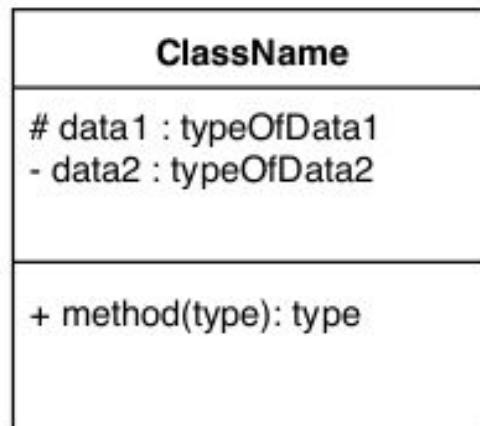
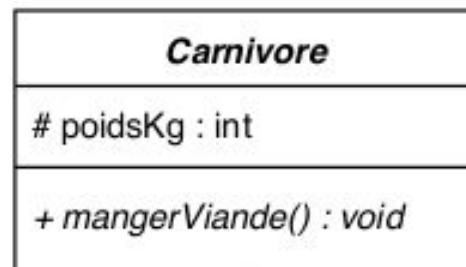


Technical
level
(software
design)



Class diagram

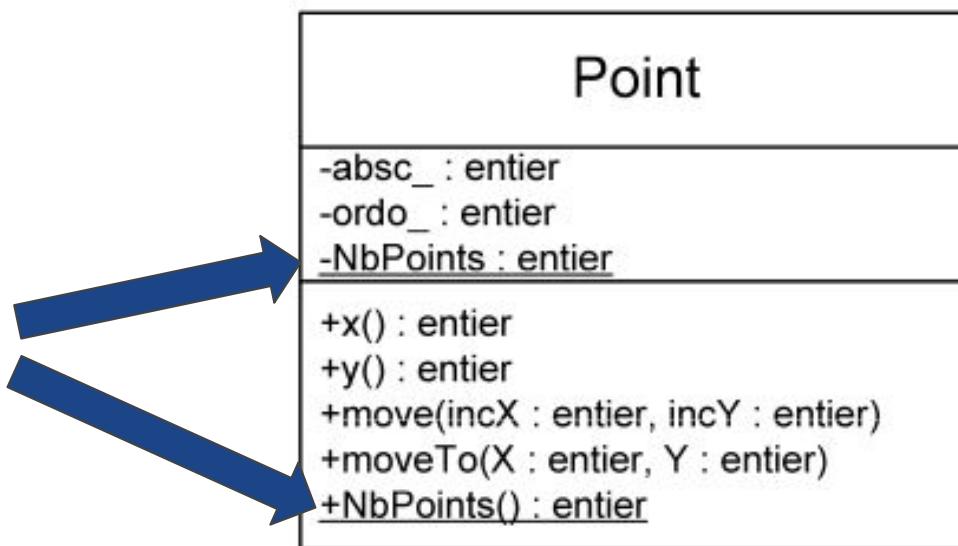
Class examples



Class diagram

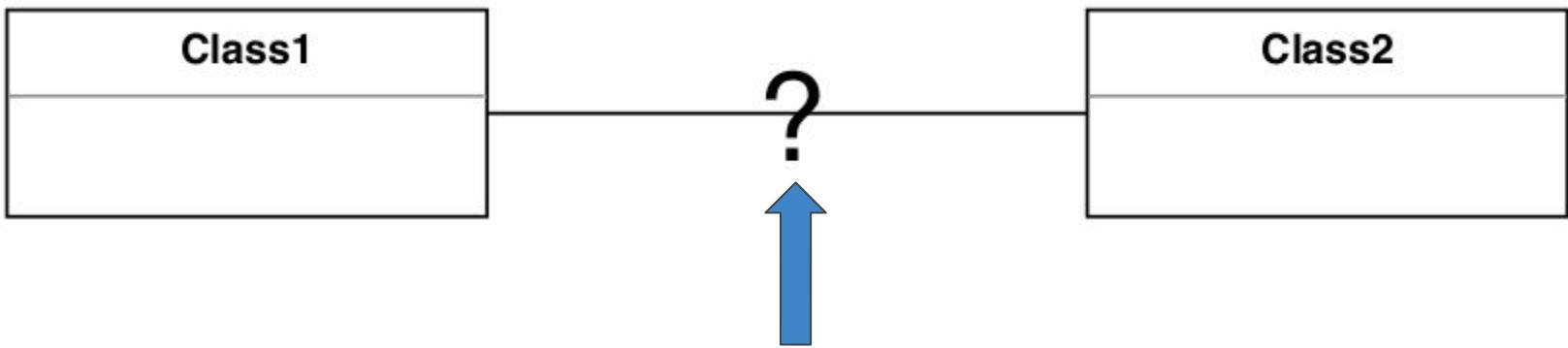
Example with **static** data and **static** methods

We underline
the “static”
attribute or
method



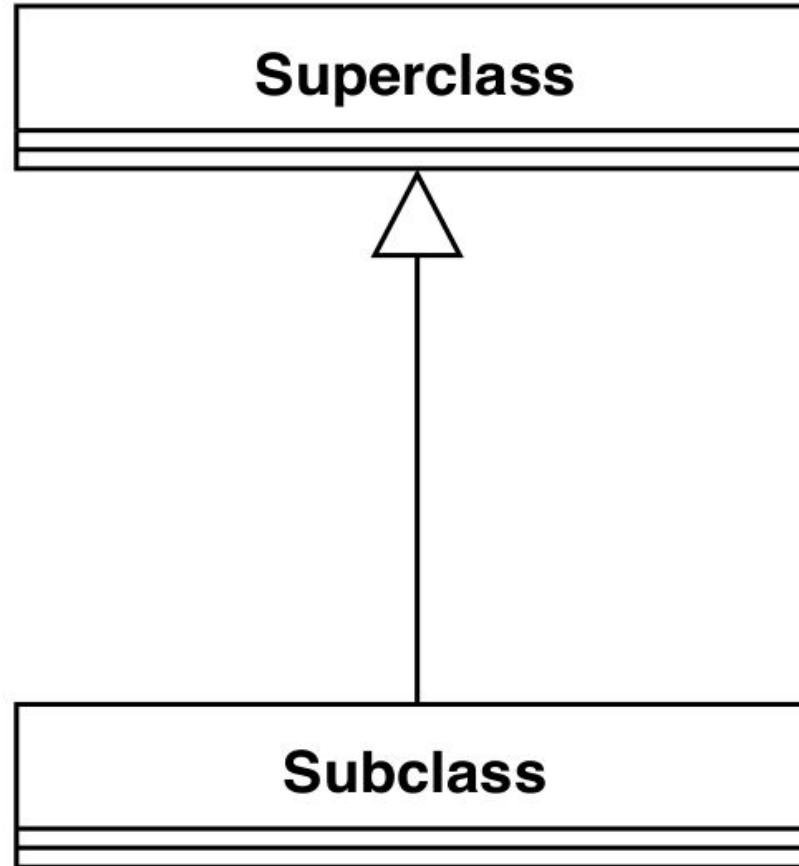
Class diagram

Links between two classes



What kind of link have
we already seen in this
course?

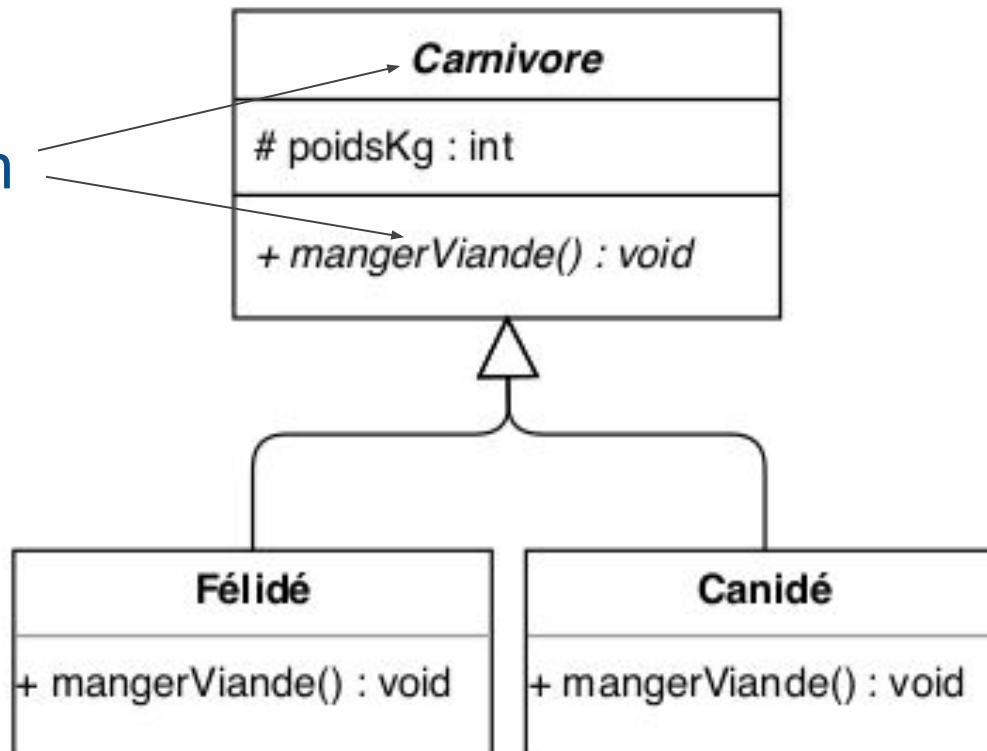
Inheritance



Reminder: **Inheritance** should be seen as a specialization of a superclass: an instance of subclass will retrieve everything from superclass in addition to its own elements.

Abstract class and abstract method

Abstract
classes and
methods are in
italics.





Example

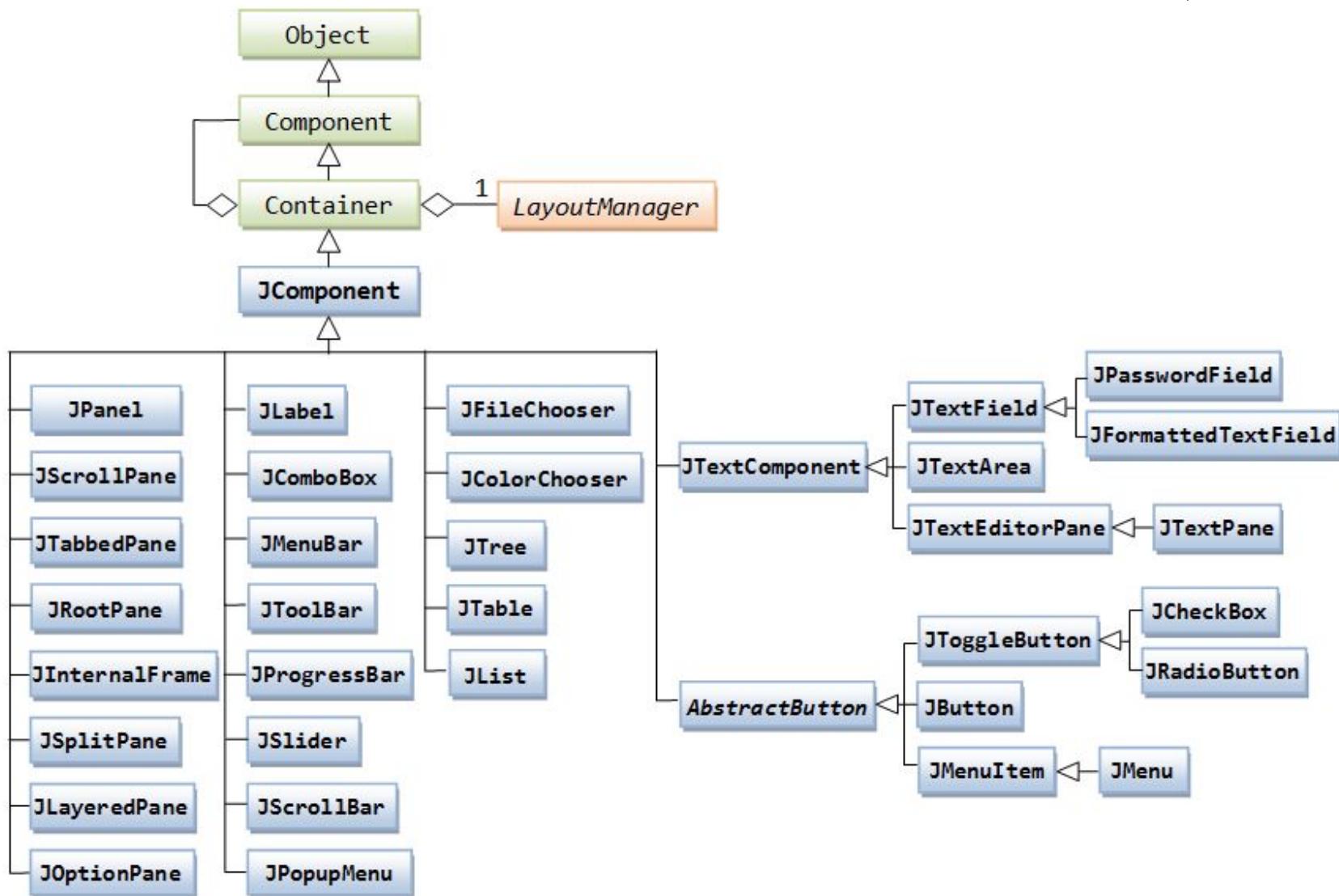
-

Inheritance in the graphics libraries

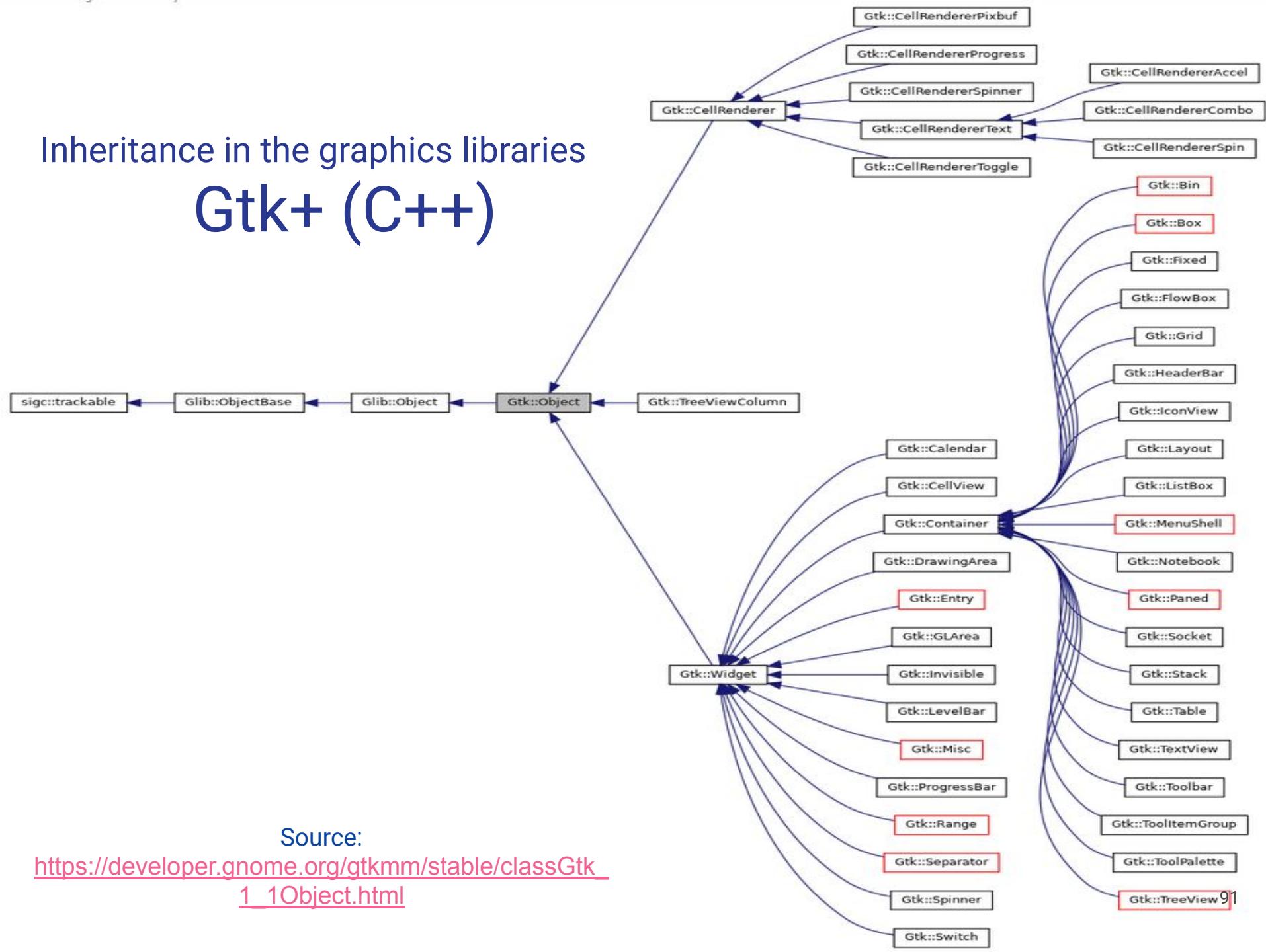


Inheritance in the graphics libraries

Swing Component (Java)



Inheritance in the graphics libraries Gtk+ (C++)

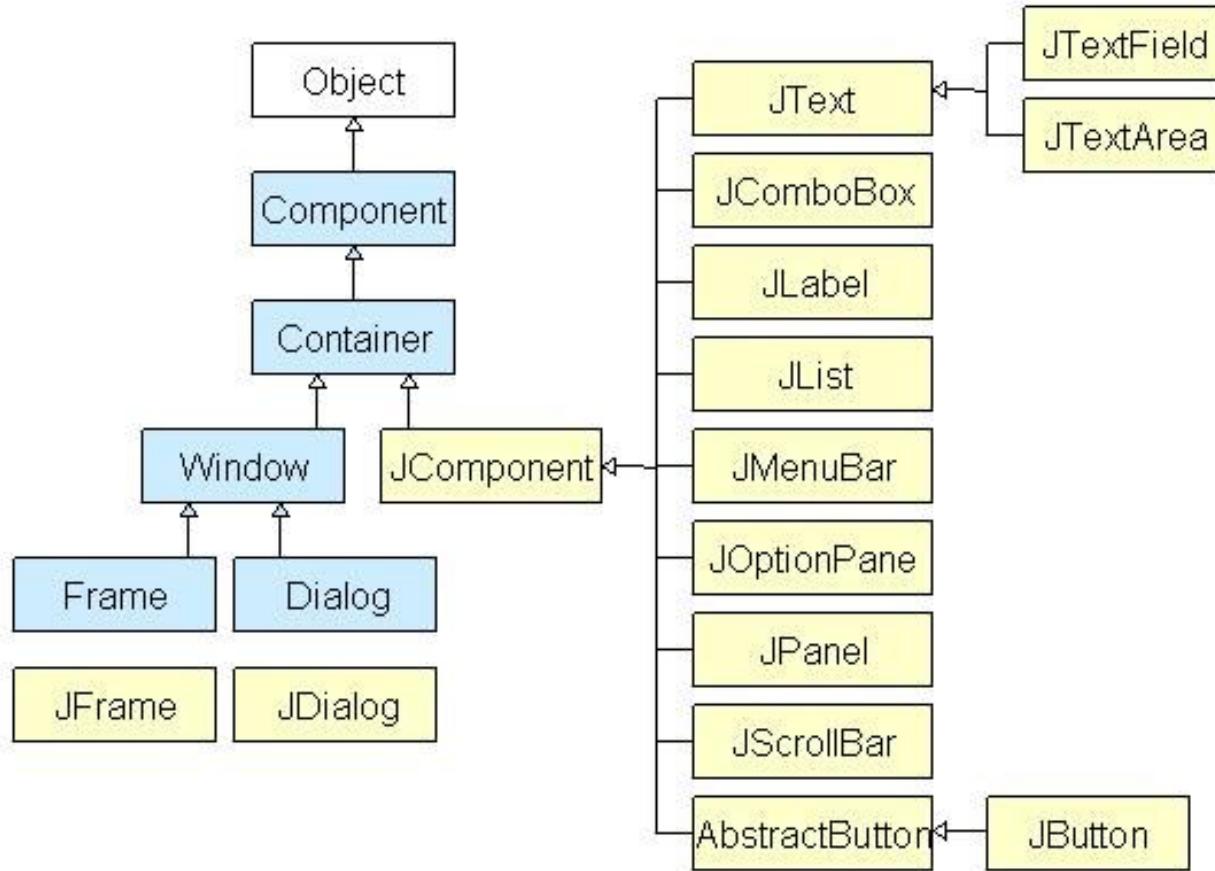


Source:

https://developer.gnome.org/gtkmm/stable/classGtk_1_1Object.html

Inheritance in the graphics libraries

Qt (C++)





Class diagram: Inheritance

- EXERCISE: Tolkien -



Among Tolkien's Mid-Earth LivingBeings, the good guys and the bad guys each have two types:

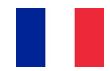
- Orcs and Trolls on the BadSide,
- Elves and Humans on the GoodSide.



Objective: Draw a coherent UML class diagram with the underlined words as entities.

Parmi les Êtres vivants de la terre du milieu de Tolkien, les gentils et les méchants ont chacun deux types :

- Les Orcs et les Trolls sont Méchants,
- Les Elfes et les Humains sont Gentils.



Objectif : Dessiner un diagramme de classe UML cohérent en vous servant des mots soulignés comme entités.



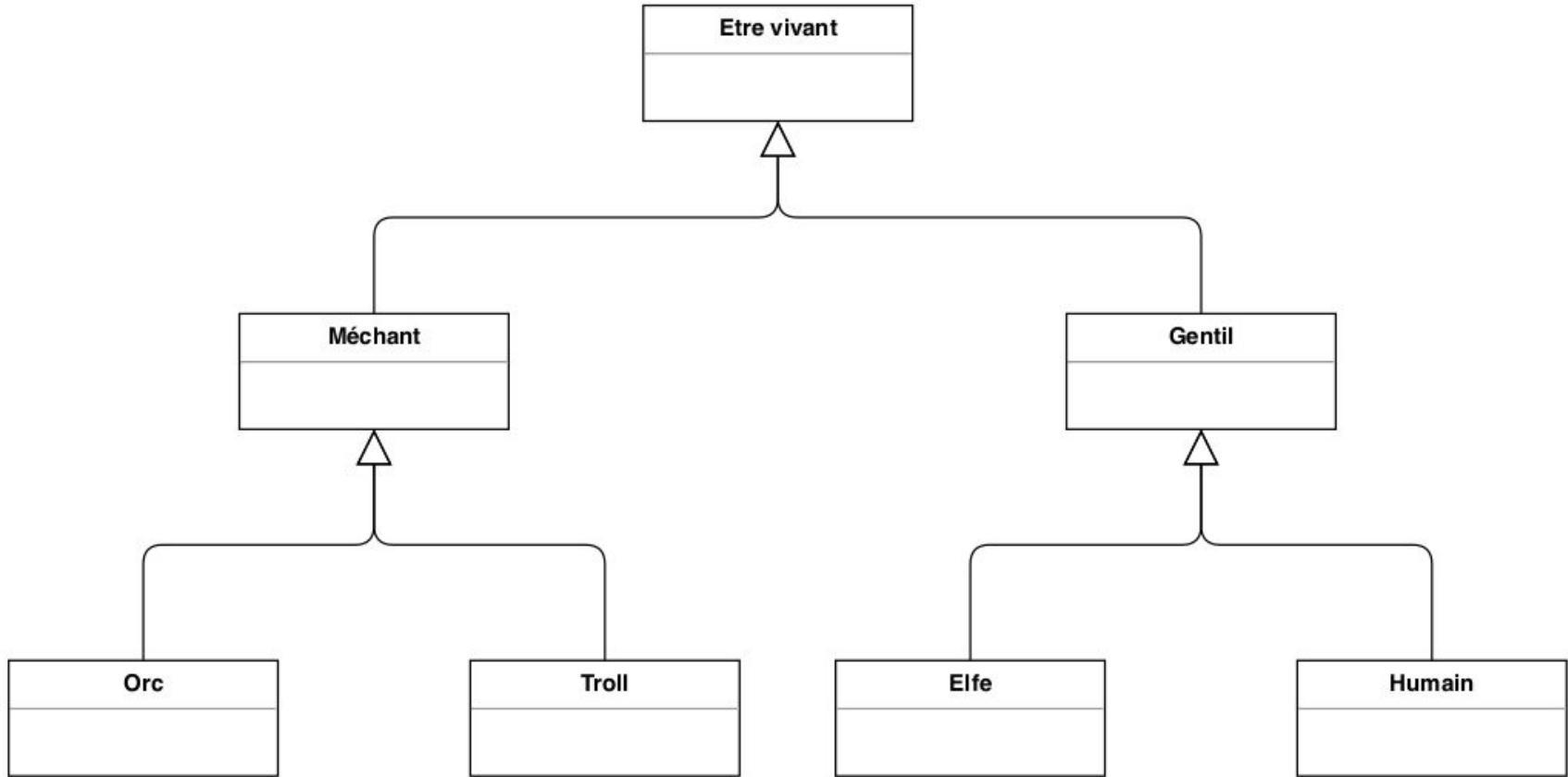
PS: Sorry for the aficionados of Tolkien's worlds for this light flight over middle earth, the hobbits will find themselves in this message.





Class diagram : Inheritance

- CORRECTION Tolkien EXERCICE -



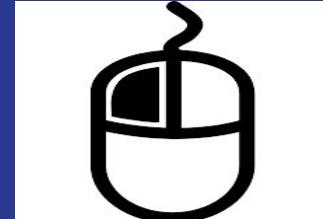
Question: Considering the unlikely union between a Troll and an Elf, what kind of inheritance would we have to do with?



Practical Work

/ Travaux Pratiques

Practical
Work
3



```

MainClass.java
1 public class MainClass {
2     public static void main(String[] args) {
3         System.out.println("Hello Abstract.");
4     }
5 }
6
7 }

Vehicle.java
1 abstract class Vehicle {
2     /*
3      * Default Constructor
4      */
5     public Vehicle()
6     {
7         km = 0;
8     }
9     /*
10    * Methods
11   */
12     abstract void move();
13     /*
14    * Attributes
15   */
16     protected int km;
17 }
18

Car.java
1 public class Car extends Vehicle {
2     /*
3      * Default Constructor
4      */
5     public Car()
6     {
7         super();
8     }
9     /*
10    * Methods
11   */
12     public void move()
13     {
14         super.km += 10;
15     }
16 }
17

Bike.java
1 public class Bike extends Vehicle {
2     public Bike()
3     {
4         super();
5     }
6     /*
7      * Methods
8      */
9     public void move()
10    {
11        super.km++;
12    }
13 }
14

```

Inheritance / Héritage

P3. 1. Take the example of vehicles (\approx slide 60) and complete its hierarchy with the following classes:
 - ***MotorCycle, Plane, Truck and Bus.***



Each object must move at different speed according to the class of object (i.e., rewrite the move() method).
 Make the **class diagram** then the **Java code (C++)**.

Inheritance / Héritage

P3

1. Complete with *MotorCycle, Plane, Truck and Bus.*

CORRECTION



Inheritance / Héritage

P3

.2. While keeping a coherent tree structure, complete with these new classes:

- *AirVehicle*,
- *RoadVehicle*,
- *WaterVehicle*,
- *Boat*,
- *Helicopter*,
- *Submarine*,
- *TwoWheels*,
-*Object*.... ?

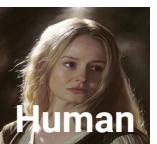
Tip: have the reflex to go through a drawing first.

Inheritance



.2. AirVehicle, RoadVehicle, WaterVehicle, Boat, Helicopter, Submarine,
TwoWheels, Object....

CORRECTION



.3.

*Take the code from **TP2***

- a. Add a superclass **LivingBeings** to the project.
- b. Make the other 3 classes inherit (Human, Elf & Orc) from **LivingBeings**.
- c. Make the superclass abstract.
- d. Imagine a method implemented (therefore not abstract) in the superclass which displays “I am a living being”. Call it from each instance of the subclass from the communication scheme of TP2. In other words, each time an instance “speaks” (in talkToBlabla), we also call the method of the superclass.



Human



Elf



Orc

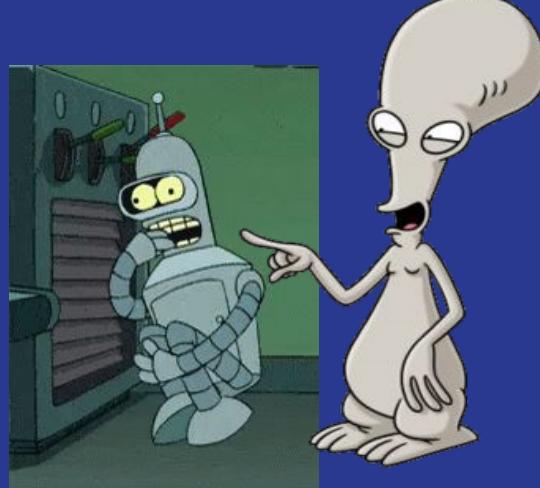


.3.

Reprendre le code du **TP2**

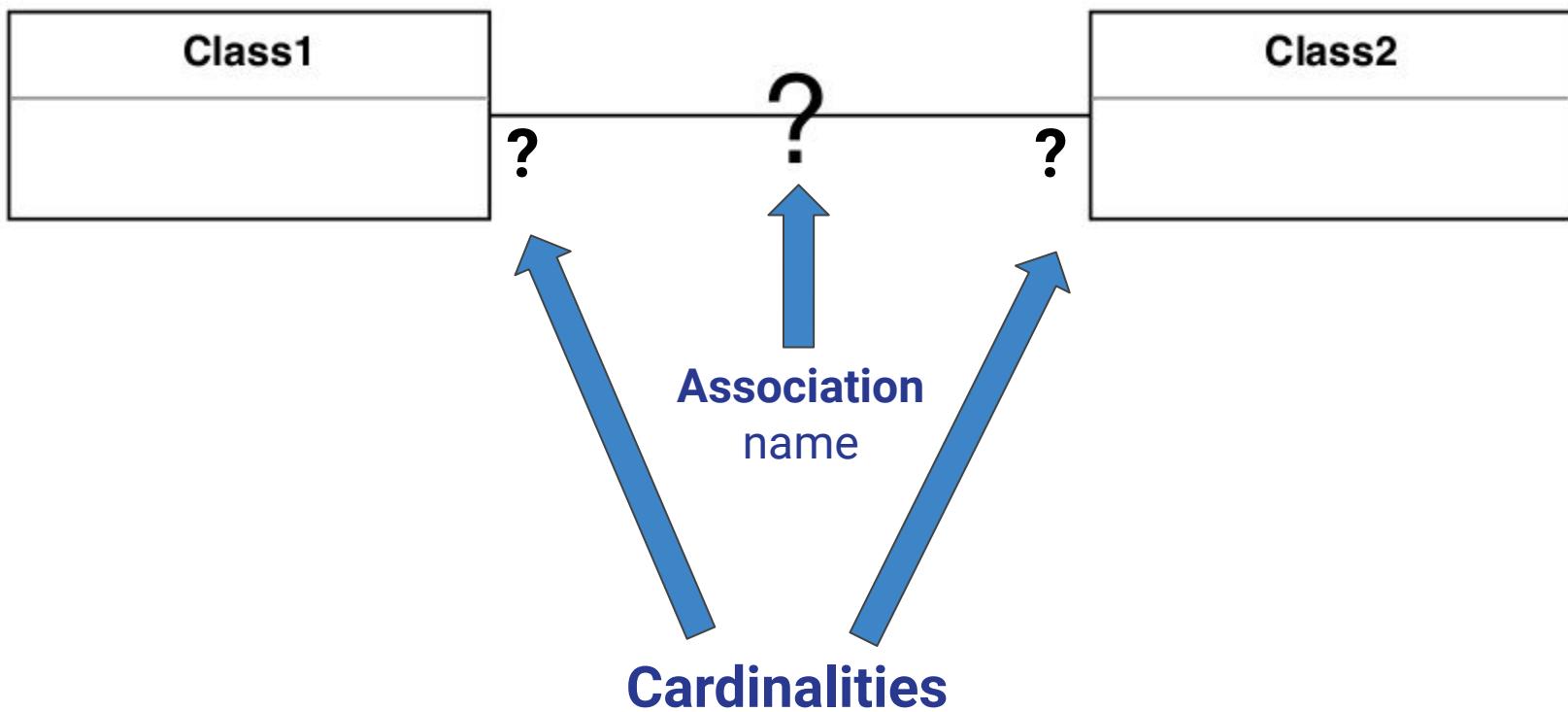
- a. Ajouter une classe mère **EtreVivant** au projet.
- b. Faire hériter les 3 autres classes (Humain, Elf et Orc) de la nouvelle classe **EtreVivant**.
- c. Rendre la class mère abstraite.
- d. Imaginer une méthode implémentée (donc non abstraite) dans la classe mère qui affiche “Je suis un être vivant”. Appeler la à partir de chaque instance de la classe fille selon le schéma de communication du TP2. Autrement dit, à chaque fois qu'une instance “parle” (dans *talkToBlabla*), on fait aussi appel à la méthode de la classe mère.

Back to the lecture / Retour au cours



Associations & Cardinalities

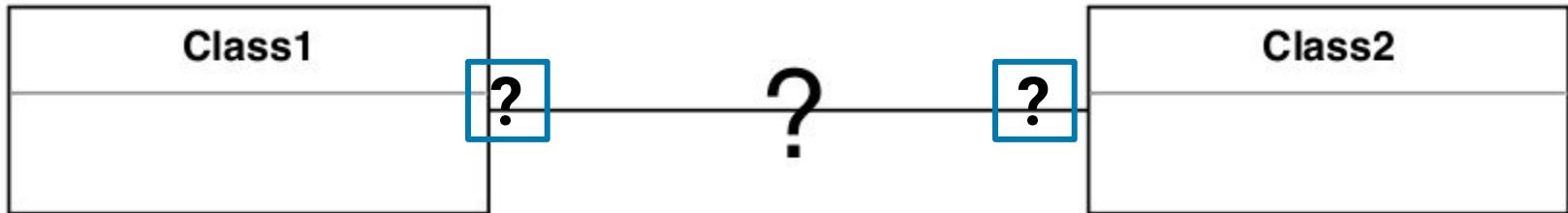
An association specifies the links between 2 classes.



Remarks:

- (1) **There is no cardinality for inheritance relationships** (not considered as "association").
- (2) Cardinalities are also called multiplicities.

UML Class diagram
Associations & Cardinalities



Cardinality	Alternative	Meaning
1	1..1	Exactly 1
*	0..*	0 or 1 or more
n	n..n	Exactly n
	1..*	1 or more
	0..1	0 or 1 (equiv. “max 1”)
	1..n	1 or more

Difference between **n** et ***** :
 - **n** is known,
 - ***** is unknown.

Association Simple

2 simple associations to describe that a bank account ("CompteBancaire") belongs either to a natural person ("PersonnePhysique") or to a legal person ("PersonneMorale").

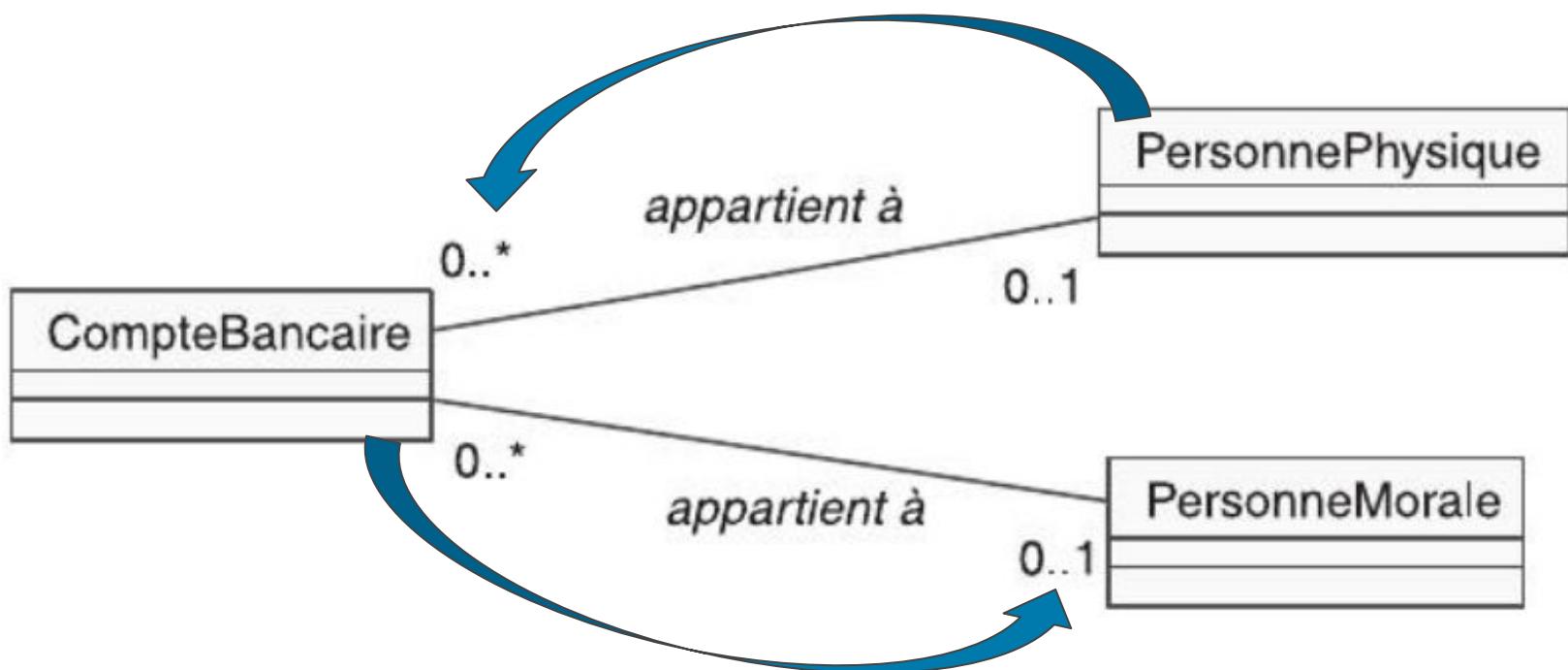
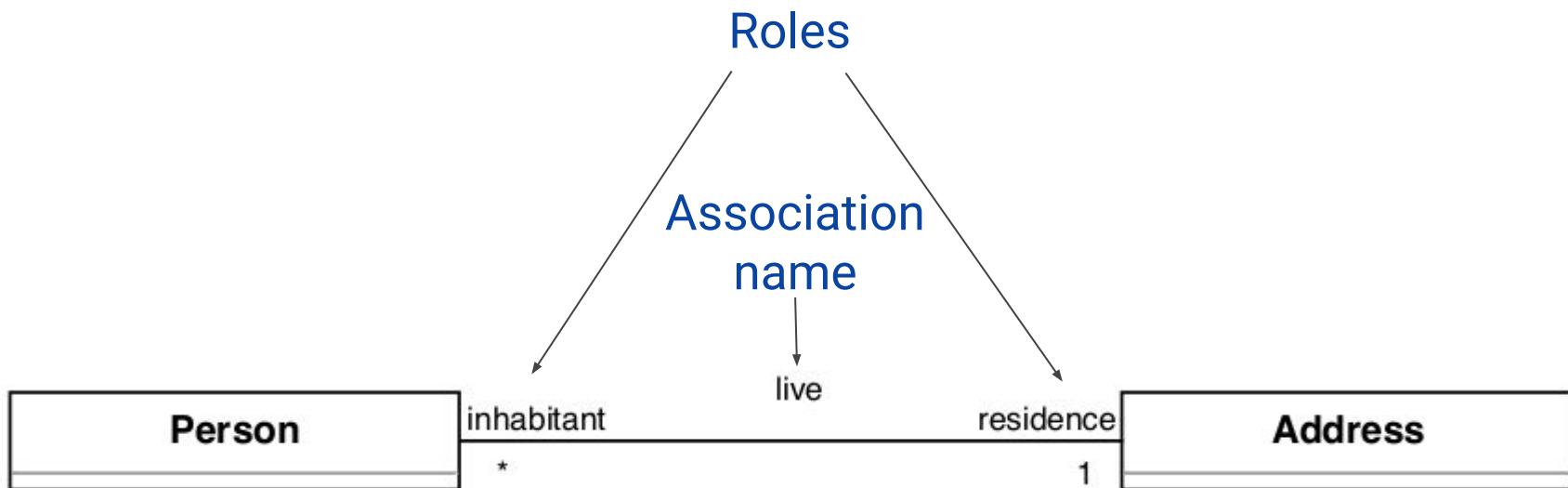


Diagramme de classe

Association

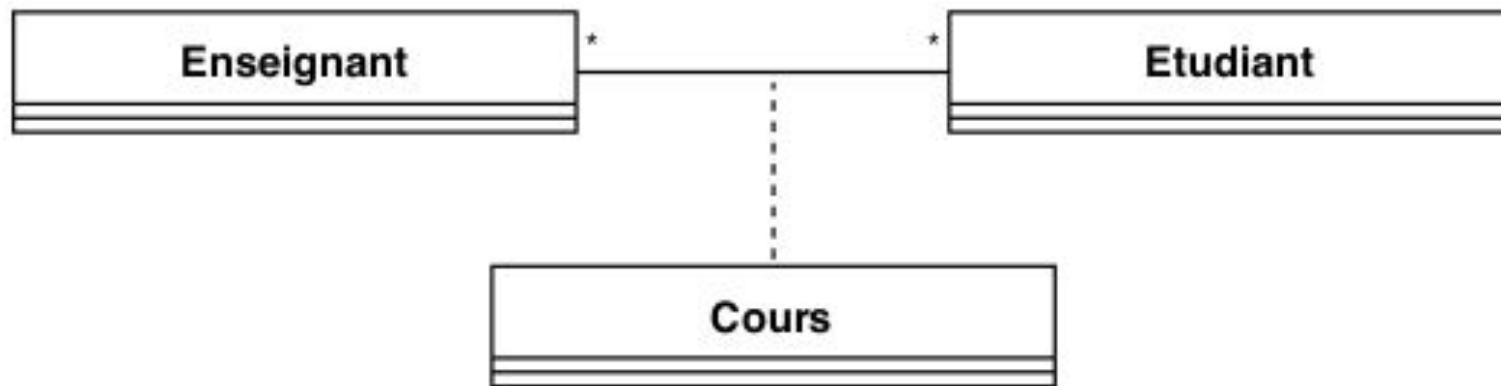
We can specify:

- The name of the association,
- The role of each class.



Classe d'Association

An association class is inseparable from the association between two classes. It allows a strong link to be modeled without going into the details of attributes and methods. It is used in the early stages of a project.



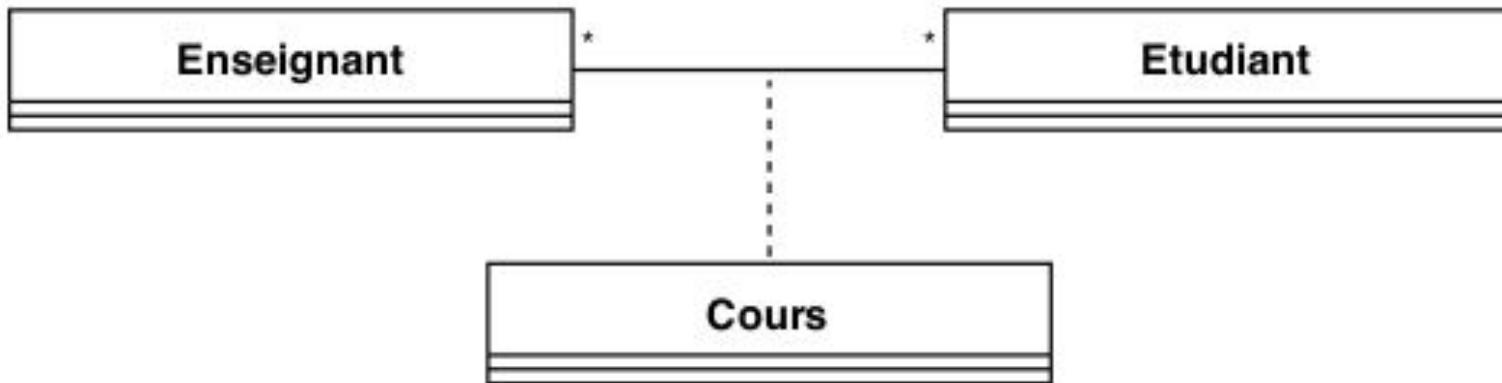
Une classe d'association est indissociable de l'association entre deux classes. Elle permet de modéliser un lien fort sans rentrer dans le détail des attributs et méthodes. Elle est utilisée dans les premières étapes d'un projet.



Classe d'Association

Mnemonic aid: if the link is broken between the two main classes, the third is no longer necessary.

> On the example: If the link is broken between teachers and students, the course no longer exists.

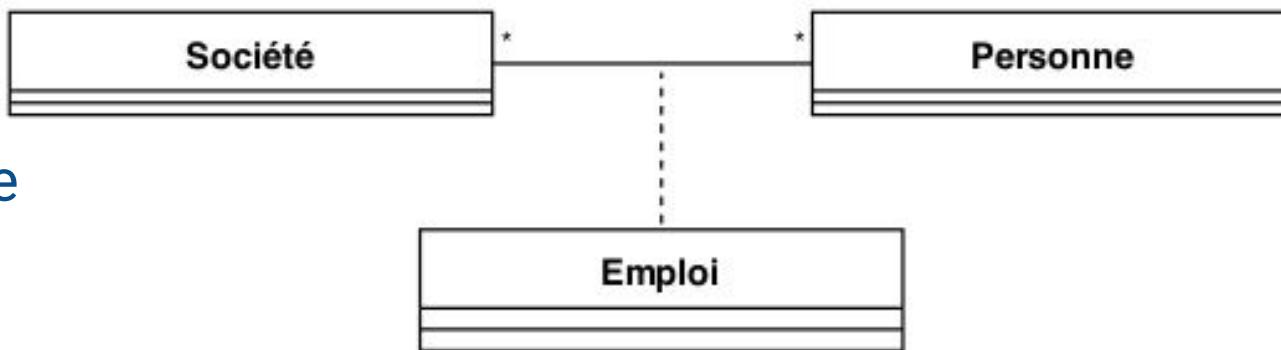


Aide mnémotechnique : si le lien est rompu entre les deux classes principales, la troisième n'a plus lieu d'être.

> Sur l'exemple : Si le lien est rompu entre enseignants et étudiants, le cours n'existe plus.



Diagramme de classe
Classe d'Association



Later stage
Of the
project

Lower
level of
abstraction

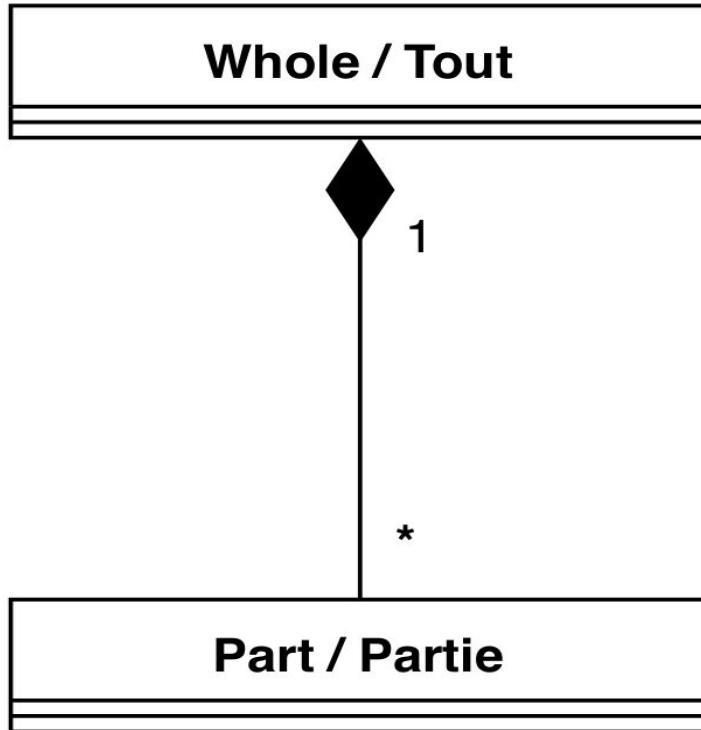
Société
sociétéId : int # emplois : Emploi[0..*]
+ get[...] : int + set[...] : void

Emploi
emploid : int # personnid : int # sociétéId : int
+ get[...] : int + set[...] : void

Personne
personnid : int # emplois : Emploi[0..*]
+ get[...] : int + set[...] : void

> Example: If the link is broken between Société and Personne, there is no more Emploi.
/ > Exemple : Si le lien est rompu entre Société et Personne, il n'y a plus d'Emploi.

Class Diagram - Association - Composition



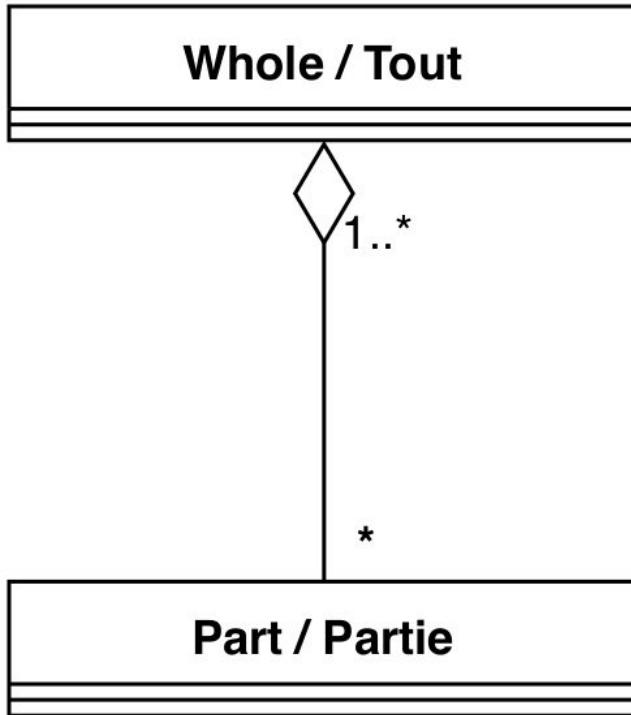
If we destroy an instance of a **Whole** class, the object of the **Part** class which is linked to the first one will disappear.



Si on détruit une instance de la classe **Tout**, l'objet de la classe **Partie** qui est liée à la première, va disparaître.



Aggregation / Agrégation

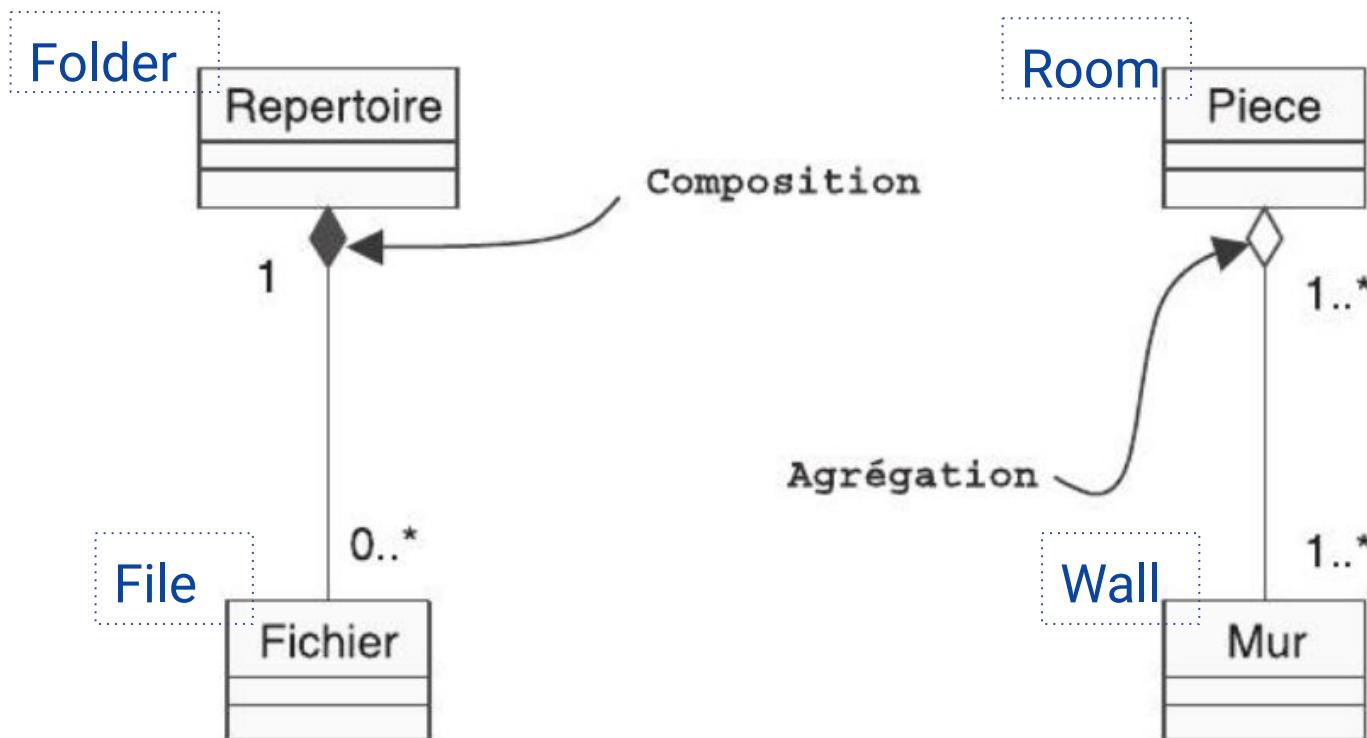


If we destroy an instance of the **Whole** class, the instance of the **Part** class which is linked to the first one will not disappear.

Si on détruit une instance de la classe **Tout**, l'instance de la classe **Partie** qui est liée à la première, ne va pas disparaître.

types of Association between classes

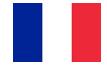
Example : Aggregation & Composition



Class diagram : Aggregation & Composition

- Exercise: JPod -

- > A friend only understands UML and its diagrams. You try to explain to him why he didn't lose all of his songs when he mistakenly deleted all the playlists in his jPod. 
- > Draw a UML diag. with the **Song**, **Album**, and **Playlist** classes and their relationships.

- > Un·e ami·e ne comprend que le langage UML. Vous essayez de lui expliquer pourquoi il n'a pas perdu toutes ses chansons lorsqu'il a supprimé par erreur toutes les *playlists* de son jPod. 
- > Dessiner un diag. UML avec les classes **Morceau** (titre), **Album** et **Playlist** ainsi qu'avec leurs relations.

Class diagram : Aggregation & Composition

- EXERCISE: FS -

- Represent associations between **File**, **Shortcut** and **Folder** into a UML class diagram as they exist in a operating system. 
 - Think about the **Folder** class, and how to represent the fact that a folder contains "*" subfolders and that a subfolder is contained in only one directory.
-
- Représentez les associations entre **Fichier**, **Raccourci** et **Dossier** dans un diagramme de classes telles qu'elles existent dans un système d'exploitation. 
 - Réfléchir également à la classe **Répertoire**, et à comment représenter le fait qu'un répertoire contient '*' sous-répertoires et qu'un sous-répertoire n'est contenu que dans un seul répertoire.

Class diagram : Aggregation & Composition

- EXERCISE GEOGRAPHY -

- Link the **Country**, **City** and **Capital** classes together.
Think about everything that can happen (war etc.).
- name, language, currency, nbOfInhabitants, Area are the attributes that can appear in at least one class.



- Lier les classes **Pays**, **Ville** et **Capitale** entre elles.
Pensez à tout ce qui peut se passer (guerre etc.).
- nom, langue, monnaie, nbHabitants, superficie sont autant d'attributs qui peuvent apparaître dans au moins une classe.



Class diagram : Aggregation & Composition

- EXERCISE : Chess 1 -

By taking all the chess pieces, and differentiating them by name (e.g., the **King** class), model the links between a **Player** and his pieces during a game with a UML class diagram. Reminder: all **Pawns** can be changed into **Queen**, **Knight** etc.



En reprenant toutes les pièces d'un jeu d'échecs, et en les différenciant par leur nom (e.g., la class **Roi**), modéliser les liens entre un **Joueur** et ses pièces durant une partie grâce à un diagramme de classe UML. Rappel : tous les **Pions** peuvent se changer en **Dame**, **Cavalier** etc.



Class diagram: Inheritance, Aggregation & Composition

- EXERCICE : Chess 2 -

Add the **Chessboard**, **Board** and **Piece** classes. Represent the correct relationship between:

- **Piece** and the different forms it can take (**Pawn**, **King** etc.),
- **Board & Chess**, **Piece & Chess**, **Board & Piece**,
- Place the attributes and methods in the appropriate classes while determining the data types and their abstraction:
 - color, moved, Check,
 - move(), take(), getOutOfTheBoard(), castling(), promote().



Ajouter les classes **Echiquier**, **Case** et **Pièce**. Représenter la bonne relation entre :

- **Pièce** et les différentes formes qu'elle peut prendre (**Tour**, **Pion**, etc.).
- **Case & Echiquier**, **Pièce & Echiquier**, **Case & Pièce**,
- Placer les attributs et méthodes dans les classes qui conviennent tout en déterminant les types de données et leur abstraction :
 - couleur, aBougé, estEnEchec,
 - seDéplacer(), prendre(), sortirDeLEchiquier(), roquer(), promouvoir().



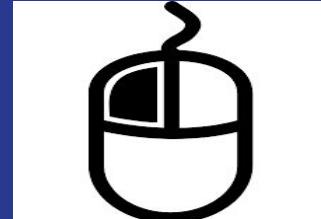
Practical Work

/ Travaux Pratiques

Practical
Work
4

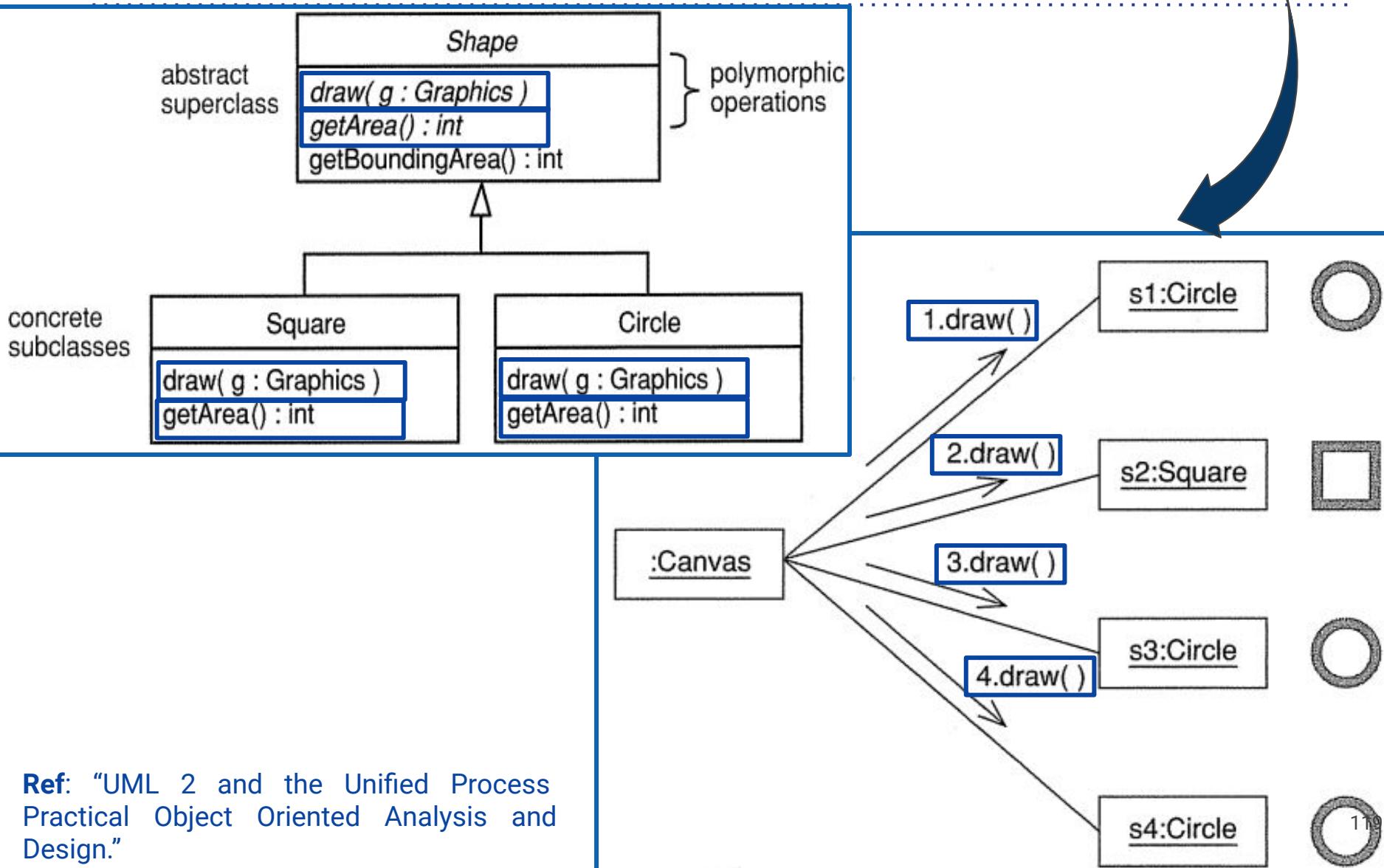
&

Practical
Work
5



Reminder on 2 fundamental concepts

Inheritance & Polymorphism

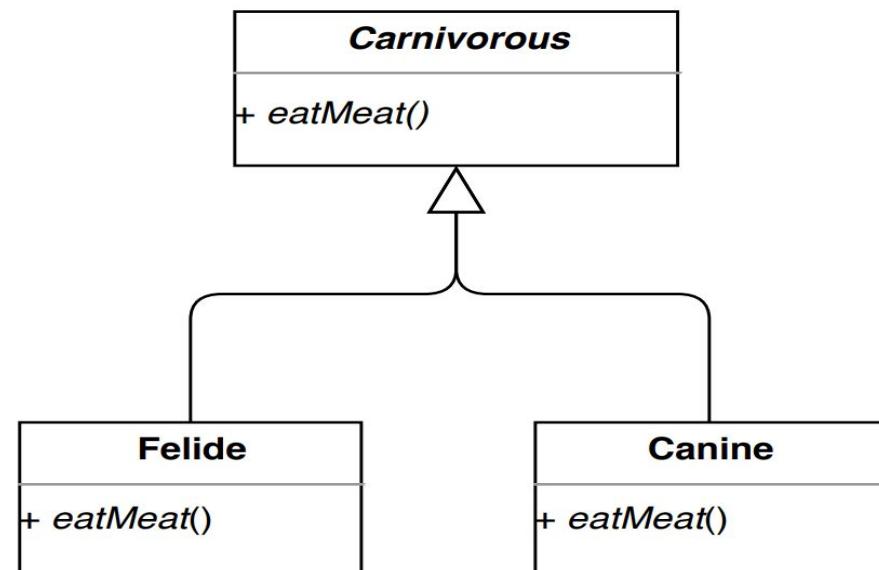
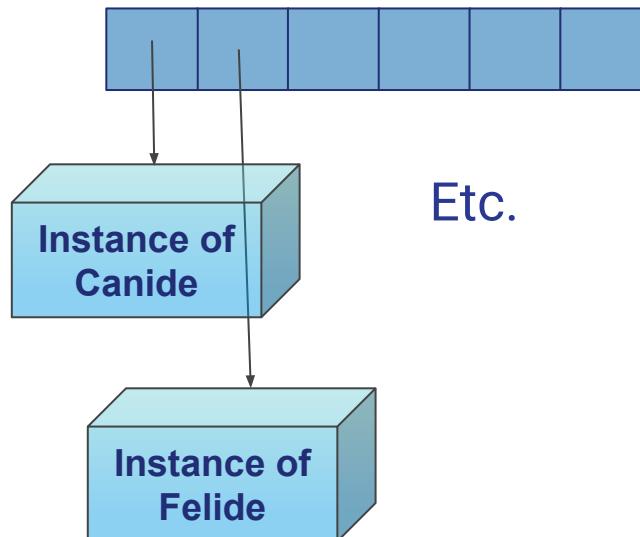




Polymorphism

Implement the example lesson with an **array¹** of **Carnivorous**, which are actually either **Felide** or **Canide**. The 3 classes each have the **eatMeat()** method but the **Carnivorous** class only defines its signature (the method is therefore **abstract**, and the class then also becomes abstract):

- Implement a different behavior for the single method of the subclasses,
- Try to use an abstract class. For example: instantiating a **Carnivorous**, call its undefined method, etc.
- + question in the next slide.



¹ the JAVA correction will use an ArrayList.



Polymorphism

Practical Work 4

...moreover...:

Give more sense to the exercise with:

- An **attribute** `weightInKg` (real type) of the **Carnivorous** class initialized in its constructor,
- A call to the **Carnivorous** of the superclass from subclasses (assuming that the starting weight is always the same, regardless of the type of class),
- Generate the **Getters & Setters*** of `weightInKg` in the superclass,
- Add a little weight when `eatMeat()` is called (integer value to add, bounded but random),
- Reuse the `ArrayList` and the `for` loop to grow the animals,
- Reuse the `ArrayList` before and after lunch, then, do the same for dinner. Show the weight changes with a `displayWeight()` method implemented in the superclass.

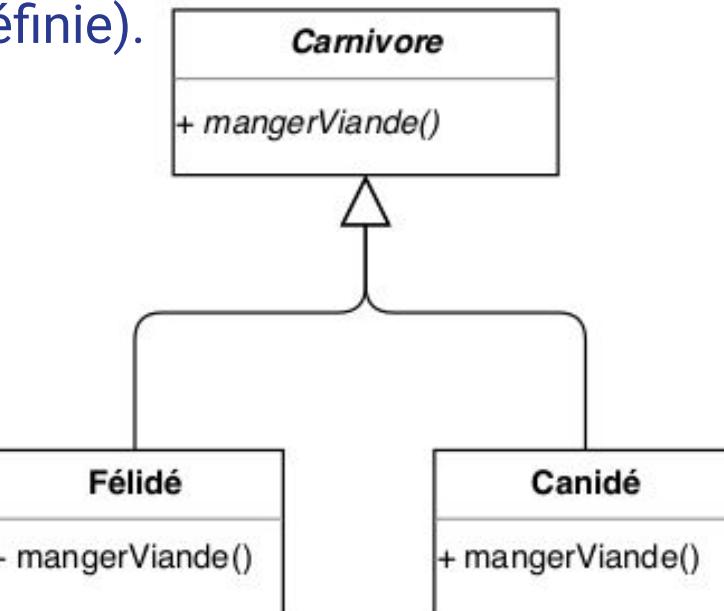
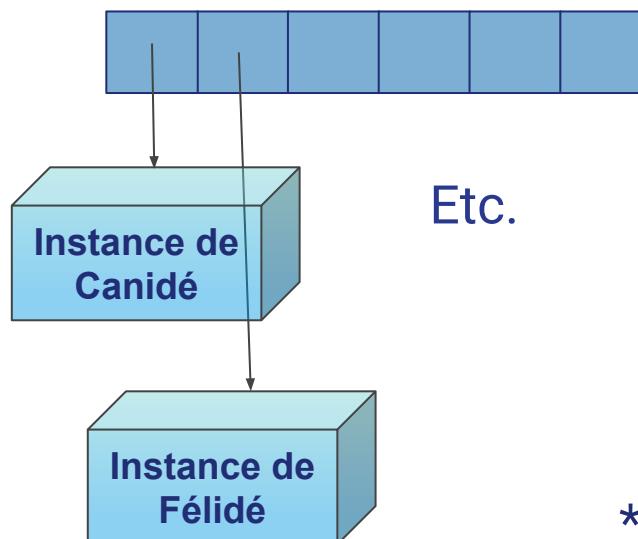
BONUS : Display the evolution of the weight of each of the animals over a month using a method of the `TimeUnit.SECONDS.sleep(1)` type for rendering;

* cf. Following slides with examples in Eclipse and Visual Studio Code.

Polymorphisme

Implémenter l'exemple de cours avec un **tableau*** de Carnivores, qui sont en réalité soit des **Félidés** soit des **Canidés**. Les 3 classes ont chacune la méthode **mangerViande()** mais la classe **Carnivore** ne définit que sa signature (la méthode est donc **abstraite**, et la classe devient alors également abstraite) :

- Implémenter un comportement différent pour l'unique méthode des classes filles,
- Essayer de manipuler une classe abstraite : instantiation d'un carnivore et appel de sa méthode (non définie).
-la suite slide en suivante....



* la correction JAVA utilisera un ArrayList



Polymorphisme

TP4 ...suite :

Donner un peu de sens à l'exercice avec :

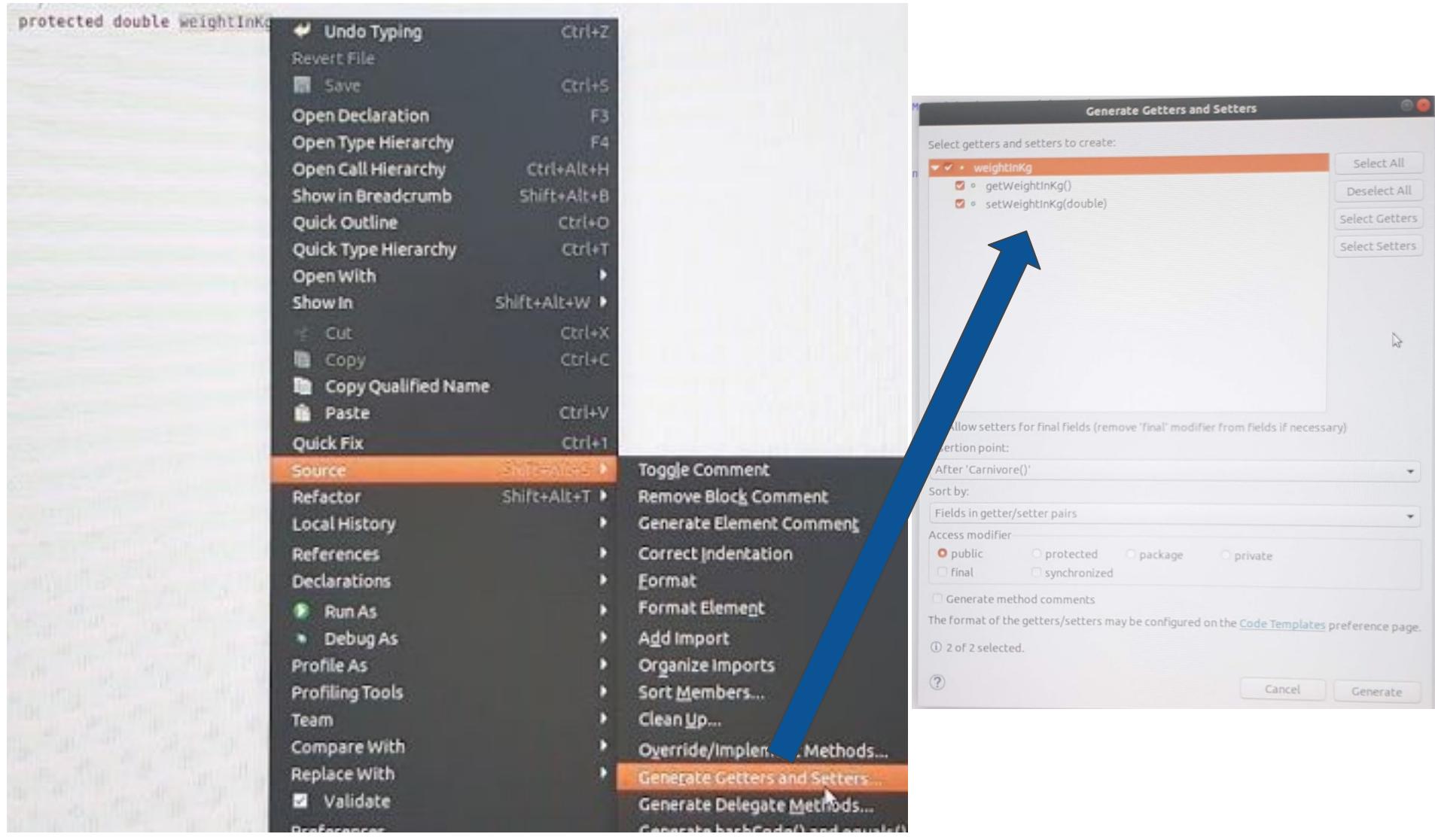
- Un **attribut** de type réel *weightInKg* de la classe Carnivore initialisé dans son constructeur,
- Un **appel au constructeur de la classe mère à partir des classes filles** (supposant que le poids de départ est toujours le même, qu'importe le type de classe),
- Générer les **Getters & Setters*** de *weightInKg* dans la classe mère,
- Ajouter un peu de poids à chaque appel de *mangerViande()* (valeur entière à ajouter, bornée mais aléatoire),
- Réutiliser l'*ArrayList* et la boucle *for* pour faire grossir les animaux,
- Réutiliser l'*ArrayList* avant et après le *déjeuner*, puis la même chose pour le *dîner*. Afficher l'évolution du poids grâce à une méthode *displayWeight()* implémentée dans la classe mère.

BONUS: Afficher l'évolution du poids de chacun des animaux sur un mois en utilisant une méthode du type *TimeUnit.SECONDS.sleep(1)* pour le rendu ;

* cf. Slides suivantes avec exemples sous Eclipse et Visual Studio Code.

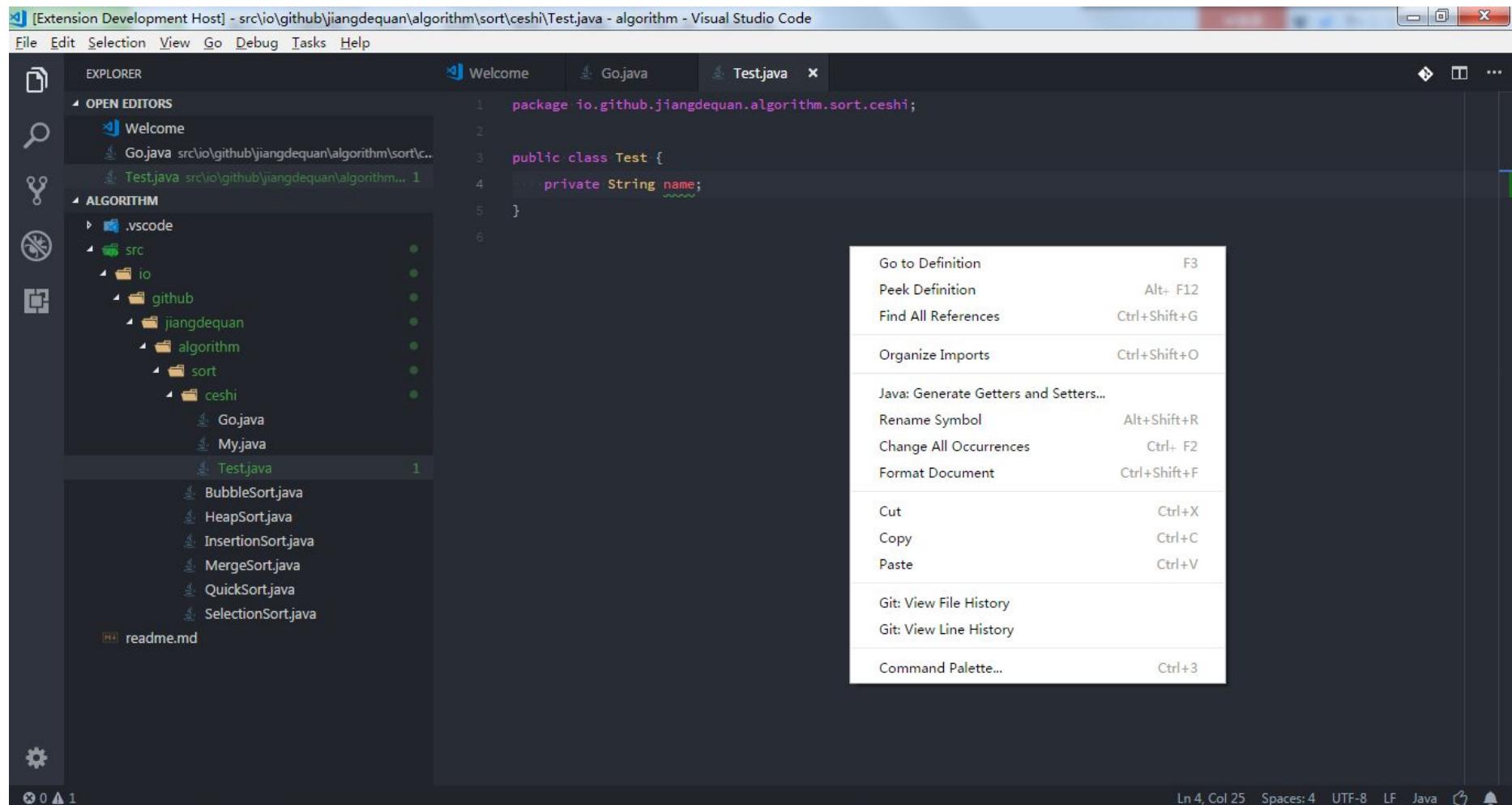
Generate Getters & Setters in an IDE

ECLIPSE example (Java or CDT / C++)



Generate Getters & Setters in an IDE

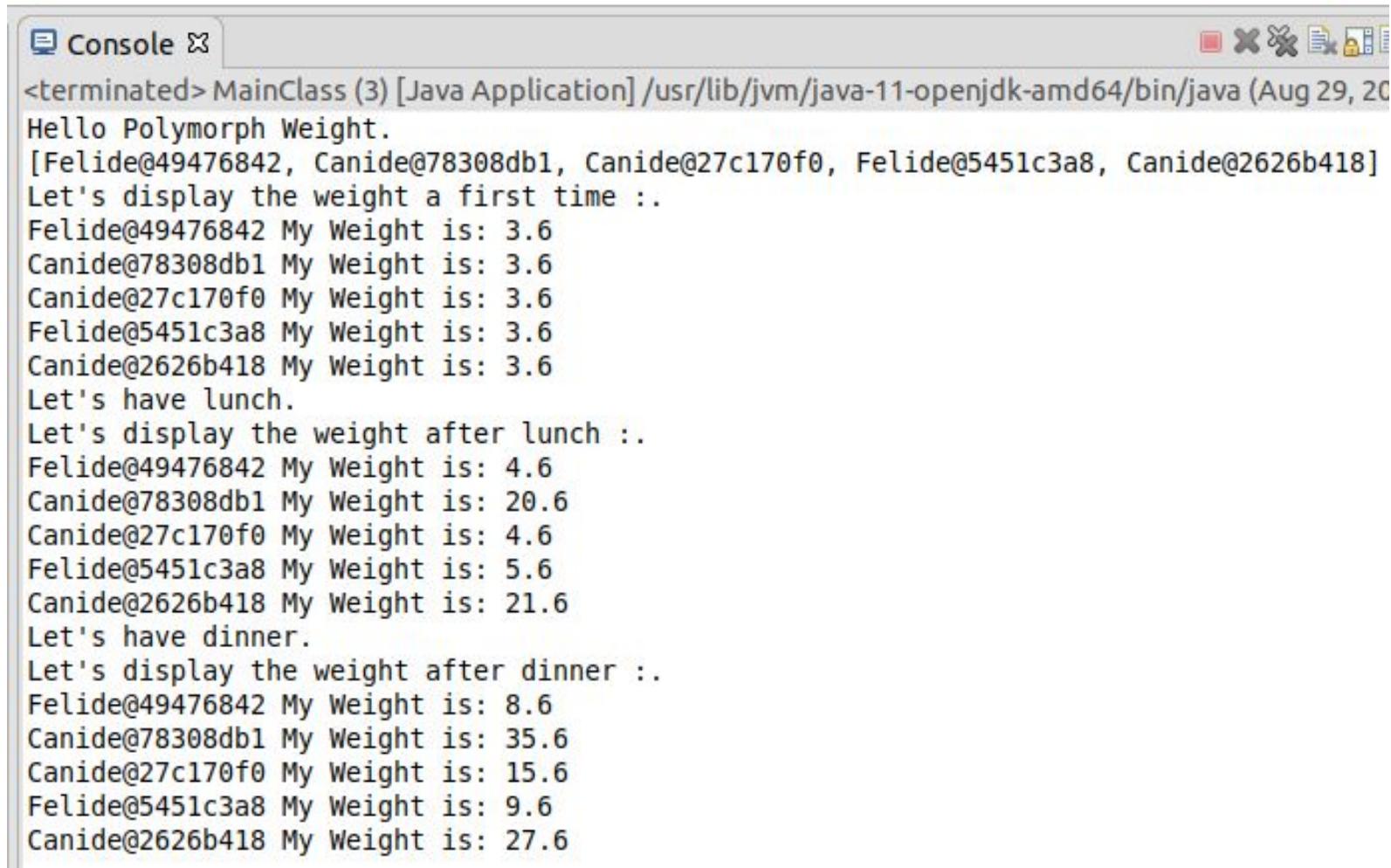
Visual Studio Code Example



Polymorphism

CORRECTION

You should get something like this:



The screenshot shows a Java application window titled "Console". The output in the console is as follows:

```
<terminated> MainClass (3) [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (Aug 29, 2023)
Hello Polymorph Weight.
[Felide@49476842, Canide@78308db1, Canide@27c170f0, Felide@5451c3a8, Canide@2626b418]
Let's display the weight a first time :.
Felide@49476842 My Weight is: 3.6
Canide@78308db1 My Weight is: 3.6
Canide@27c170f0 My Weight is: 3.6
Felide@5451c3a8 My Weight is: 3.6
Canide@2626b418 My Weight is: 3.6
Let's have lunch.
Let's display the weight after lunch :.
Felide@49476842 My Weight is: 4.6
Canide@78308db1 My Weight is: 20.6
Canide@27c170f0 My Weight is: 4.6
Felide@5451c3a8 My Weight is: 5.6
Canide@2626b418 My Weight is: 21.6
Let's have dinner.
Let's display the weight after dinner :.
Felide@49476842 My Weight is: 8.6
Canide@78308db1 My Weight is: 35.6
Canide@27c170f0 My Weight is: 15.6
Felide@5451c3a8 My Weight is: 9.6
Canide@2626b418 My Weight is: 27.6
```

Polymorphism

CORRECTION



Practical Work 5

RACING!



(Polymorphism and pseudo-random number generation)

(read all 3 slides of TP5 before starting)

- Take over the hierarchy of TP3 and its 6 subclasses.
- The goal of this lab is to “simulate” a race with at least <<2>> instances of each subclass. Each type of vehicle uses its own **move()** method through polymorphism and inheritance, at every step.
- A step therefore corresponds to the call to **move()** of each object and the number of steps may vary depending on the object, it will be determined once the object has finished the race.
- All **Vehicles** are stored in the same array.
- All vehicles have a Boolean disadvantage variable initialized randomly in the (good) constructor (random function, cf. TP4). At the end of each **move()** method, the disadvantage value is updated for the next step, and this time randomly (it's up to you to adjust) the following probabilities of being TRUE (i.e., 1):
 - 95% for **Plane**, 60% for **Car**, 40% for **MotorCycle**, 35% for **Truck**, 20% for **Bus**, 5% for **Bike**.



- ***disadvantage*** corresponds to different penalties depending on the type of vehicle (you have to adjust): e.g., Plane: cannot fly, Car: slows down to 10% of its speed, [...], Bike goes to 90% of its speed.
- If ***disadvantage*** is FALSE (i.e. 0), each call to ***move()*** advances a certain number of kilometers depending on the type of vehicle (it's up to you to adjust). For example, the car can progress 10km if ***disadvantage*** is FALSE and 1km if ***disadvantage*** is TRUE.
- Add a polymorphic but never abstract ***display()*** method. It will first display its own object (“pointer this”), then, secondly, the number of km, this time using the method of the same name of the superclass.
- The race ends when all the vehicles have traveled ***nbKmToArrive*** as you determine. Please note: when a vehicle has reached its goal, it will no longer move forward.





- Find the right mechanism that allows all vehicles to finish the race, being able to display the overall classification, the classification by vehicle category and the number of stages required for each object.
- BONUS 1: Make “<<2>>” variable so you can easily expand or shrink the number of instantiated Vehicles of each class.
- BONUS 2: Create a simple console output that will display the progress of the race in a terminal using a method such as: TimeUnit.SECONDS.sleep(1); by displaying the progression after each call to **move()** of all instances.

Example :

Step n°6, race status:

----->
----->
----->

| Car.instance2
| Bike.instance2
| Plane.instance1
| **nbKmToArrive**





TP5 RACING!



(Polymorphisme et génération de nombres pseudo-aléatoires)

(lire toutes les 3 slides du TP5 avant de commencer)

- Reprendre la hiérarchie du TP3 et ses 6 classes filles,
- Le but de ce TP est de “simuler” une course avec au moins <<2>> instances de chaque classe fille. Chaque type de véhicule utilise sa propre méthode **move()** grâce au polymorphisme et à l’héritage, et ceci à chaque étape.
- Une étape correspond donc à l’appel de **move()** de chaque objet et le nombre d’étapes peut varier selon l’objet, il sera déterminé une fois que l’objet aura terminé la course.
- L’ensemble des **Vehicles** est stocké dans un même tableau.
- Tous les véhicules ont une variable booléenne **disadvantage** initialisée aléatoirement dans le (bon) constructeur (*fonction random, cf. TP4*). A la fin de chaque méthode **move()**, la valeur de **disadvantage** est mise à jour pour la prochaine étape, et ceci aléatoirement avec cette fois (à vous d’ajuster) les probabilités d’être TRUE (i.e., 1) suivantes :
 - 95% pour **Plane**, 60% pour **Car**, 40% pour **MotorCycle**, 35% pour **Truck**, 20% pour **Bus**, 5% pour **Bike**.

- **disadvantage** correspond à différentes pénalités selon le type de véhicule (à vous d'ajuster) : e.g., **Plane**: ne peut voler, **Car**: ralentit à 10% de sa vitesse, [...], **Bike** va à 90% de sa vitesse.
- Si **disadvantage** est FAUX (i.e. 0), chaque appel de **move()** fait progresser d'un certain nombre de kilomètre selon le type de véhicule (à vous d'ajuster). Par exemple, la voiture pourra progresser de 10km si **disadvantage** est FAUX et de 1km si **disadvantage** est TRUE.
- Ajouter une méthode **display()** polymorphe mais jamais abstraite. Elle va d'abord afficher son propre objet (**this**), puis, dans un second temps, le nombre de km, cette fois-ci grâce à la méthode du même nom de la classe mère.
- La course se termine lorsque tous les véhicules ont parcouru **nbKmPrArriver** que vous déterminerez. Attention : lorsqu'un véhicule a atteint son but, il n'avance plus.



- Trouver le bon mécanisme qui permet à tous les véhicules de finir la course, d'être capable d'afficher le classement général, le classement par catégorie de véhicule et le nombre d'étape nécessaire à chaque objet.
- BONUS 1 : Rendre “<<2>>” variable afin de pouvoir facilement élargir ou rétrécir le nombre de **Vehicles** instanciés de chaque classe.
- BONUS 2 : Réaliser une sortie console simple qui va afficher l'évolution de la course dans un terminal grâce à une méthode du type : `TimeUnit.SECONDS.sleep(1);` en affichant ce qui a été parcouru à chaque appel de `move()` de l'ensemble des instances. *Exemple :*

Etape n°6, état de la course :

```

-----> | Car.instance2
-----> | Bike.instance2
-----> | Plane.instance1
| nbKmPrArriver

```



Polymorphisme

Practical
Work 5

RACING!

NO....

CORRECTION



Back to the lecture / Retour au cours



Aggregation & Composition

- JAVA -

/ Question: How to distinguish the case of aggregation and composition in JAVA?

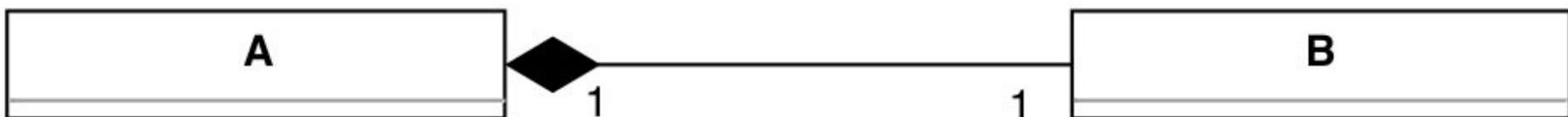


/ Question : comment distinguer le cas de l'agrégation et de la composition en JAVA ?



Composition

- JAVA -



The instance of class **B** which makes up an instance of A **is created when A is constructed**.

The link between these two classes from the point of view of B, **is set by the Setter setA**.

The screenshot shows two Java code editors side-by-side. The left editor contains the code for class A:

```
1 public class A {  
2     protected B b;  
3  
4     public A() {  
5         this.b = new B();  
6         this.b.setA(this);  
7     }  
8 }  
9  
10  
11  
12
```

The right editor contains the code for class B:

```
1 public class B {  
2     protected A a;  
3  
4     public A getA() {  
5         return a;  
6     }  
7  
8     public void setA(A a) {  
9         this.a = a;  
10    }  
11  
12  
13  
14 }
```

In both code snippets, the protected fields 'b' and 'a' are highlighted with a blue box.

Aggregation JAVA



The link between the two classes from the point of view of B as from the point of view of A, is informed by the Setters.

The difference lies in the fact that **it is the setB of A which will make the two links** "It is A who has the monopoly of the links".

The screenshot shows two Java code editors side-by-side. The left editor is for class A.java and the right for class B.java.

A.java:

```
1 public class A {  
2     protected B b;  
3     public A() {}  
4     public void setB(B b)  
5     {  
6         this.b = b;  
7         b.setA(this);  
8     }  
9 }  
10  
11  
12  
13 }
```

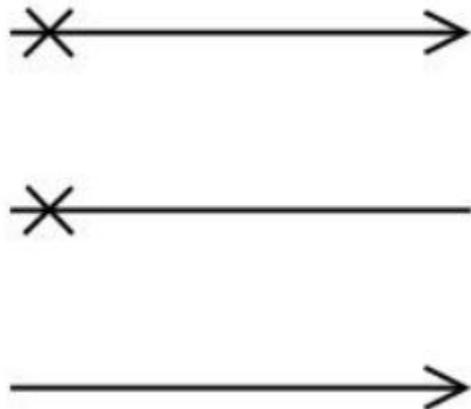
B.java:

```
1 public class B {  
2     protected A a;  
3     public A getA()  
4     {  
5         return a;  
6     }  
7     public void setA(A a)  
8     {  
9         this.a = a;  
10    }  
11  
12  
13  
14 }
```

Annotations in blue boxes highlight specific parts of the code:

- In A.java: The field `protected B b;` and the call `b.setA(this);` in the `setB` method.
- In B.java: The field `protected A a;` and the assignment `this.a = a;` in the `setA` method.

Associations with the navigation-arrow symbol / Navigualités restreintes des associations

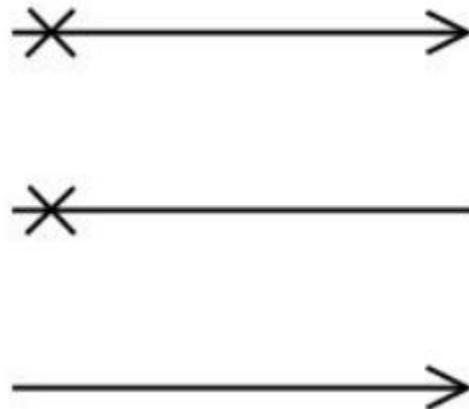


- These 3 arrows model **the same thing**,
- They limit an association to one direction.

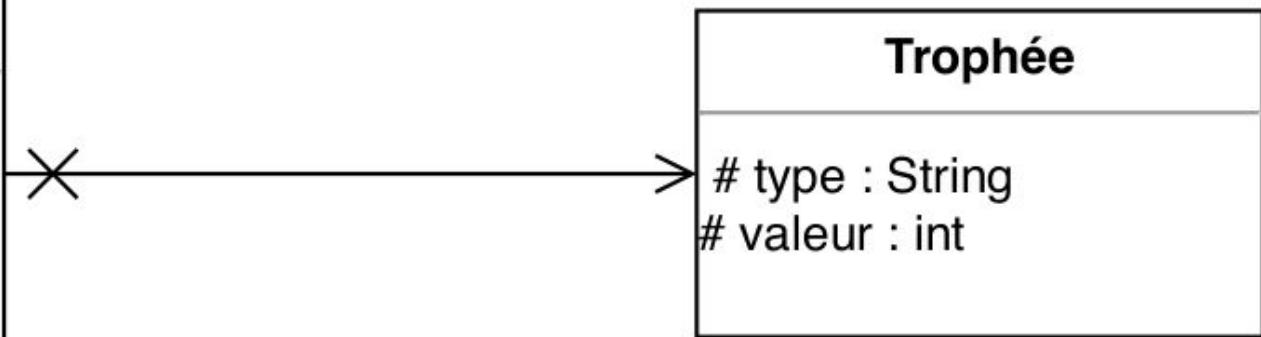


Associations with the navigation-arrow symbol / Navigualités restreintes des associations

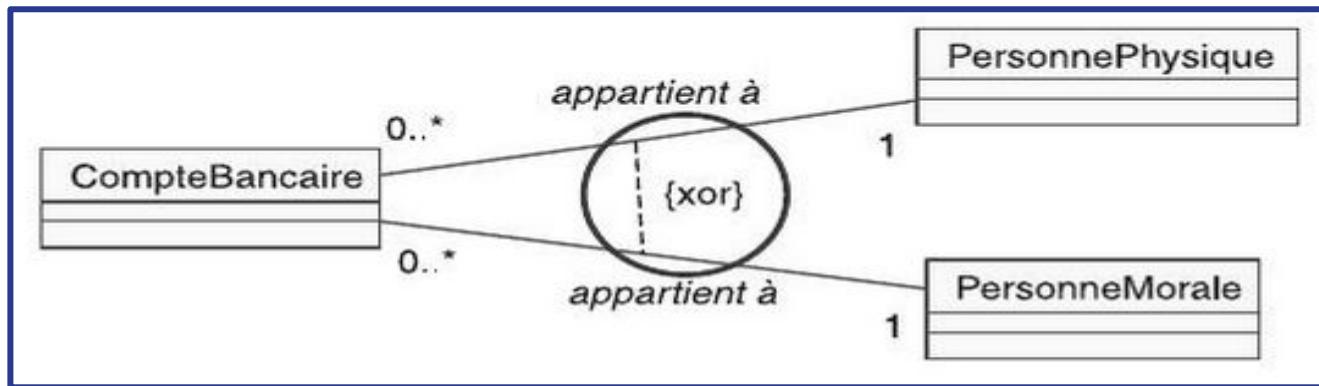
Example



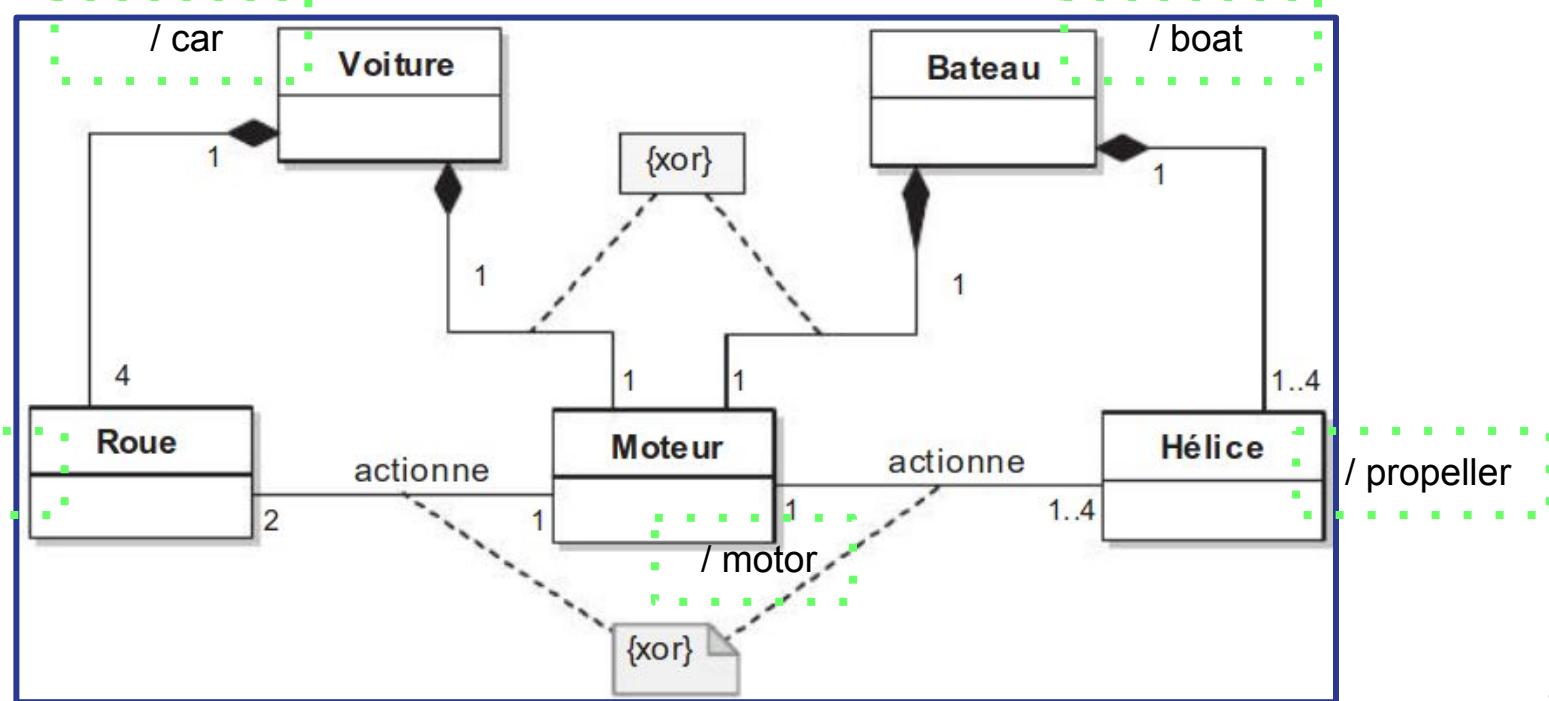
- These 3 arrows model **the same thing**,
- They limit an association to one direction.



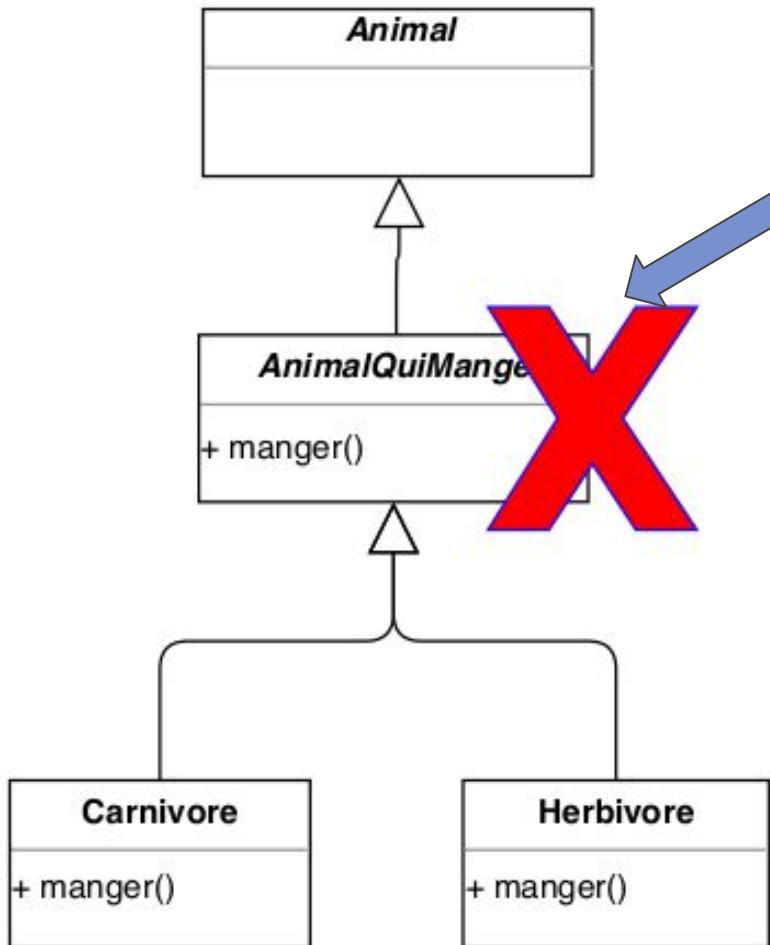
Special Associations - 2 examples of {xor}



{xor} = {eXclusive OR}



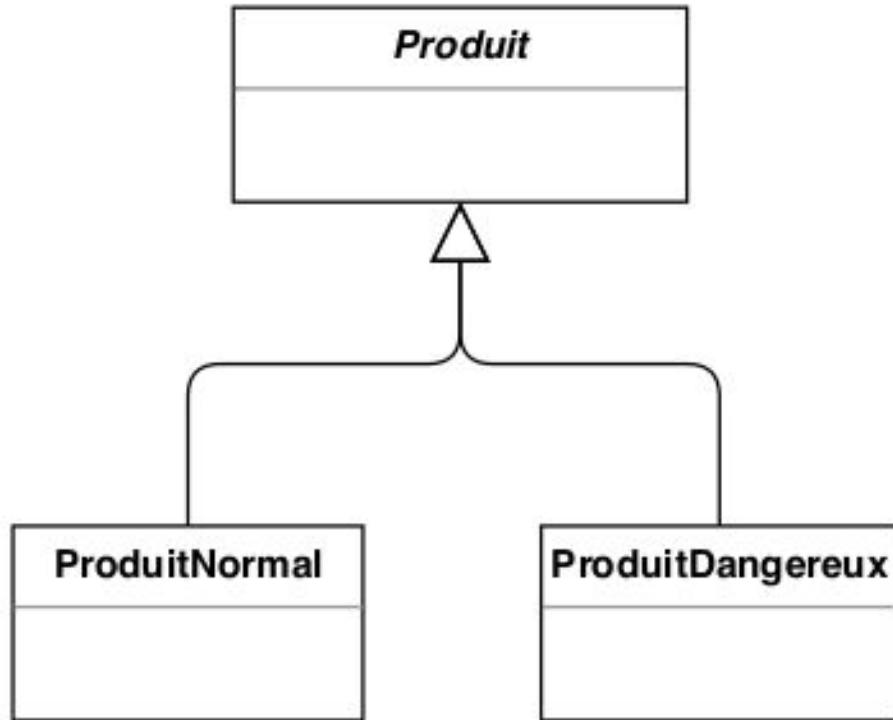
Some details on inheritance and associations



- **Avoid unnecessary tree structure:**
 - An inheritance is only justified if the subclass has at least one specific attribute, method or association;
- All associations of a superclass apply to subclasses.

Some details on inheritance and associations

Tiny Exercise “Product” / “Produit”



- Is this inheritance justified?
- If the answer is no (...), it needs to be improved!



- Est-ce que cet héritage est justifié ?
- Si la réponse est non (...), il faut l'améliorer !

Class Diagram

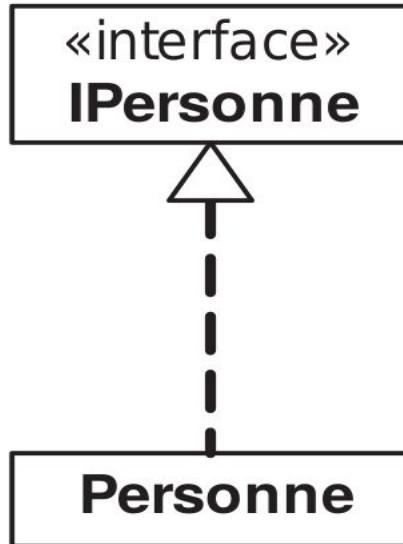
Interface concept

- Brings together a coherent set of properties and operations,
 - Must be implemented by at least 1 class,
 - **A class can implement several interfaces,**
 - **An interface can be implemented by several classes.**
-
- Remarks:
 - An interface **defines** a kind of **contract** that must be respected by the classes which implement the interface.
 - **Solves problems related to multiple inheritances.**
 - **We can compare it to an abstract class having only constants and abstract methods (Java ≥ 8).**

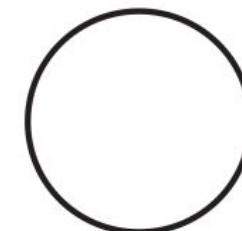
Class Diagram

Interfaces in UML

- Represented as a class,
- NO *italic*
- Indicated by `<< interface >>` above the class name.



Another (but rare) way to represent an interface.

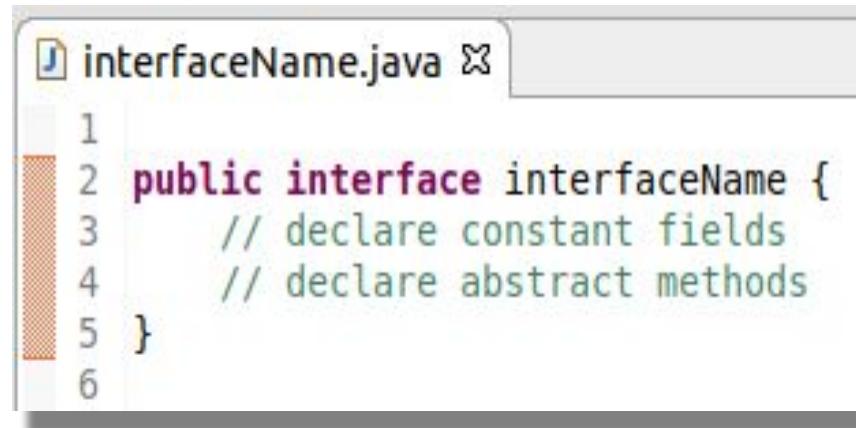


IPersonne

Class Diagram

Interface & JAVA

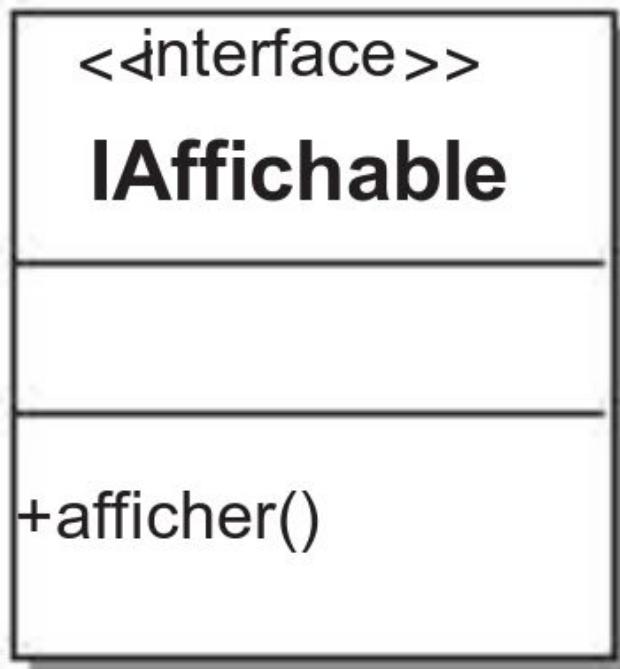
In Java: unlike UML and C++, **only simple inheritance is allowed**, but can be “replaced” by interfaces.



A screenshot of a Java code editor window titled "interfaceName.java". The code defines a public interface named "interfaceName" with two blank method declarations:

```
1
2 public interface interfaceName {
3     // declare constant fields
4     // declare abstract methods
5 }
6
```

Example : UML & JAVA



UML

(Class diagram)

```
public interface IAffichable {  
    public void afficher();  
}
```

JAVA

Class Diagram

Interface & JAVA

Properties of interfaces in Java 7 and earlier

- interfaces can be realized (i.e. **implemented**) by **several different classes**,
- **the classes can implement several different interfaces**,
- **All the methods of an interface are abstract**,
- All the methods of an interface must be implemented by the classes that implement it.

Since Java 8

- A **default implementation** can be provided (with the **default** keyword),
- Possibility of having static methods defined in the interfaces.

Example JAVA Version ≥ 8

```
2 public interface interfaceName {  
3     // declare constant fields  
4     public static final String exemple = "voilalexemple";  
5     // declare abstract methods  
6     public double otherMethod();  
7     // declare default methods  
8     default public double surface()  
9     {  
10         return - 1;  
11     }  
12     default public double perimetre()  
13     {  
14         return - 1;  
15     }  
16 }
```

Example JAVA

Implementation of an interface by a class

```
public interface MyInterface {  
    void add(int value);  
    void remove(int value);  
    void show();  
}
```

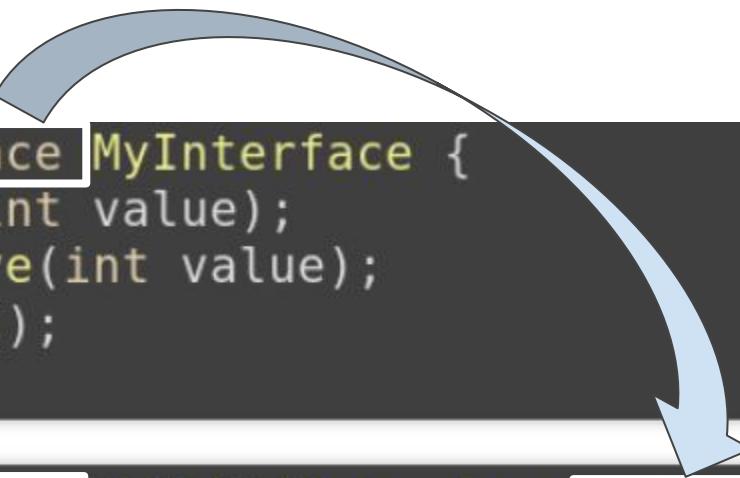
```
public class MyInterfaceRealization implements MyInterface {  
    private int x;  
    public void add(int value) {  
        x += value;  
    }  
    public void remove(int value) {  
        x -= value;  
    }  
    public void show() {  
        System.out.println(x);  
    }  
}
```

Inheritance between interfaces

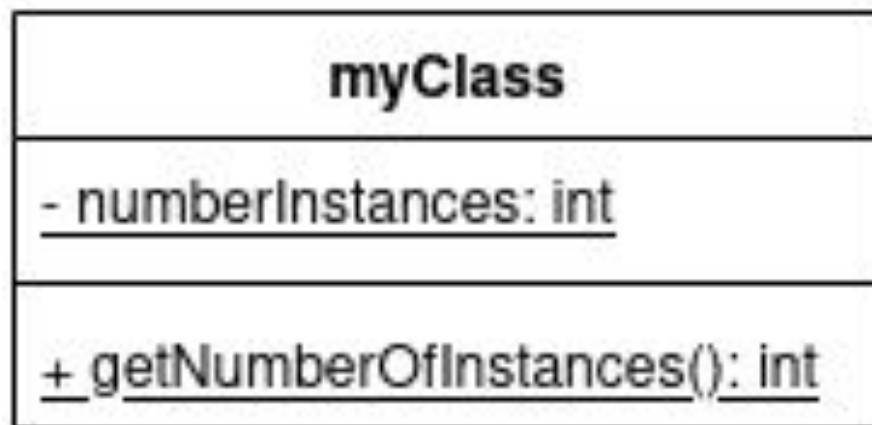
(/ Héritage entre interfaces)

```
public interface MyInterface {  
    void add(int value);  
    void remove(int value);  
    void show();  
}
```

```
public interface MyChildInterface extends MyInterface {  
    void multiply(int value);  
    void divide(int value);  
}
```



Static Data & ***Static*** methods in UML



Static Data & Static methods in java

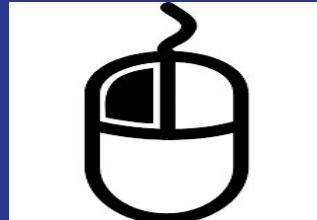
```
public class MyStaticMembClass {  
    public static final int INCREMENT = 2;  
    public static int incrementNumber(int number){  
        return number+INCREMENT;  
    }  
}
```

```
public class MyStaticImportExmp {  
    public static void main(String a[]){  
        System.out.println("Increment value: "+INCREMENT);  
        int count = 10;  
        System.out.println("Increment count: "+incrementNumber(count));  
    }  
}
```

Output:

Increment value: 2
Increment count: 12

Practical Work 6





Java & interfaces

Implement a JAVA program with:

- a **Polygon** interface with an abstract method **getArea()** and a default method **getPerimeter()** which displays the perimeter of a polygon whose sides are given as parameters;
 - example : ...**getPerimeter(3,8,9,10)** displays 30.
- A **Triangle** class implementing the **Polygon** interface with its 3 sides (integer) as private data. A constructor retrieves the values of the 3 sides. **getArea()** “Triangle version” will output the area of the triangle.
- A main function instantiating one or more Triangles and where all the methods of the program are called and displayed.
- “BONUS”: Try with other **Polygons** such as **Rectangle**, **Pentagon**, **Hexagon** etc.



Java & interfaces

Practical
Work 6

Implémenter un programme JAVA avec :

- une interface **Polygon** ayant une méthode abstraite **getArea()** et une méthode par défaut **getPerimeter()** qui affiche le périmètre d'un polygone dont les côtés sont donnés en paramètres ;
 - exemple : ...**getPerimeter(3,8,9,10)** affiche 30.
- Une classe **Triangle** implémentant l'interface **Polygon** avec comme données privées ses 3 côtés (entiers). Un constructeur récupère les valeurs des 3 côtés. **getArea()** "version **Triangle**" va afficher l'aire du triangle.
- Une fonction *main* instanciant un ou plusieurs **Triangles** et où toutes les méthodes du programme sont appelées.
- “BONUS”: Tenter l’expérience avec d’autres Polygones tels que le Rectangle, Pentagone, Hexagone etc.

Back to the lecture / Retour au cours



3.

Unified Modeling Language (UML)

3.3. Object Diagram */ Diagramme d'objets*

Object diagram

An object diagram shows:

- **Class instances,**
- Links between objects,
- The state of the system at a given time (i.e. attributes values).

Somehow:

- the class diagram models the rules,
- **the object diagram models the facts.**

Note: The class diagram is often sufficient. The object diagram can express a special case or facilitate the understanding of a complex class diagram.



Un diag. d'objets représente :

- des **instances de classes,**
- les **liens entre les objets,**
- l'état du système à un moment donné (valeurs des attributs),

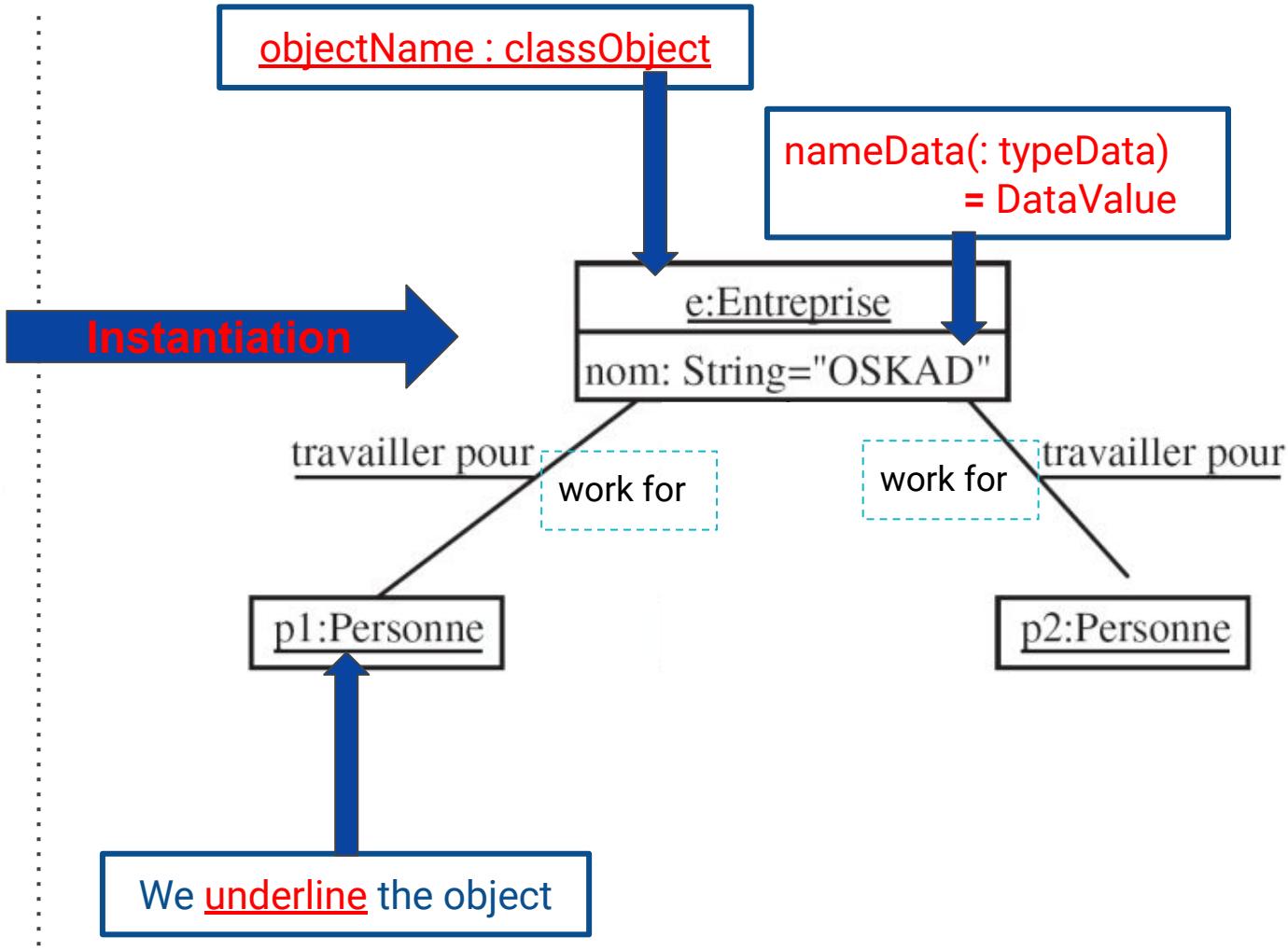
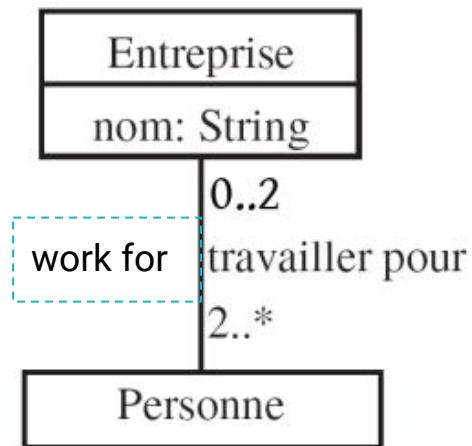
D'une certaine manière :

- le diagramme de classe modélise les règles,
- **le diagramme d'objet modélise les faits.**

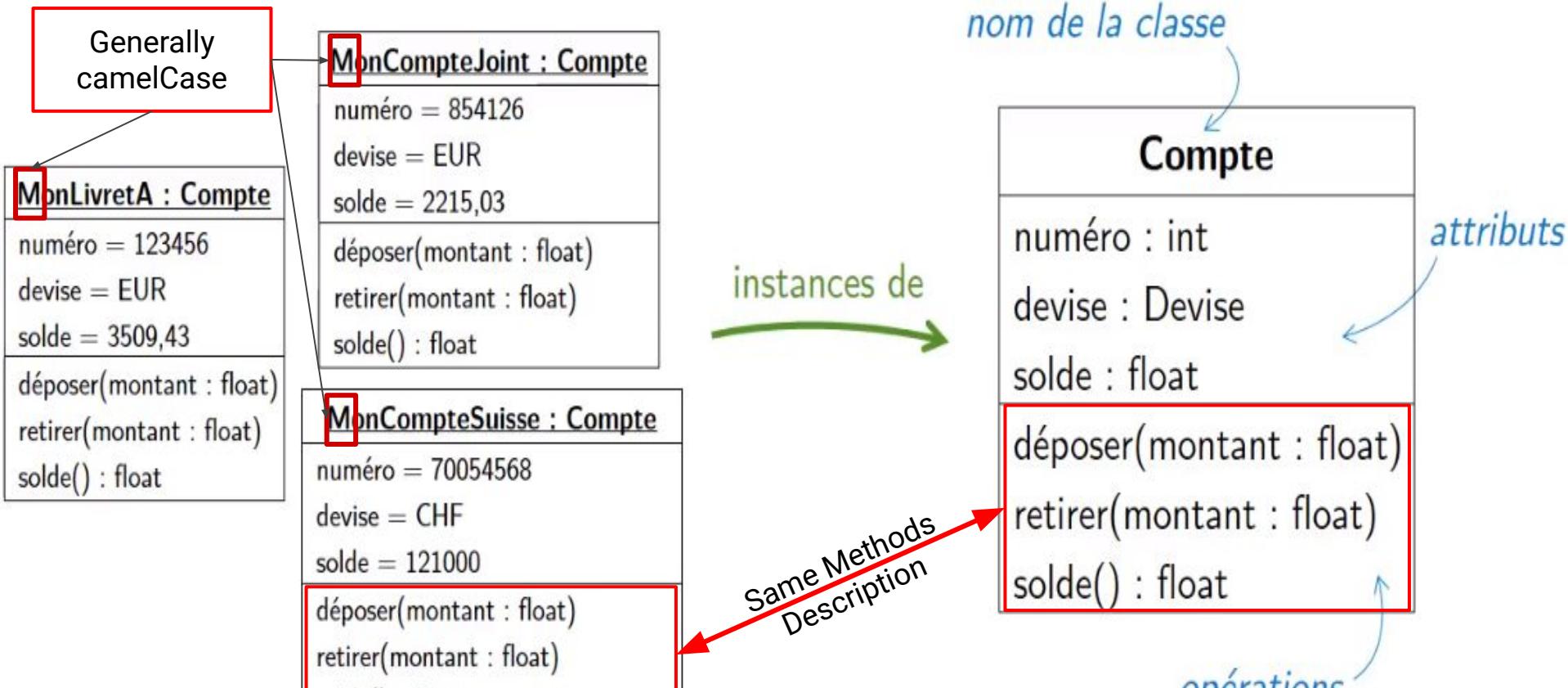
Note : le diag. de classe est souvent suffisant. Le diag. d'objets peut exprimer un cas particulier ou faciliter la compréhension d'un diagramme de classe complexe.



Object Diagram - Example 1



Object Diagram - Example 2



Object
Diagram

Class Diagram

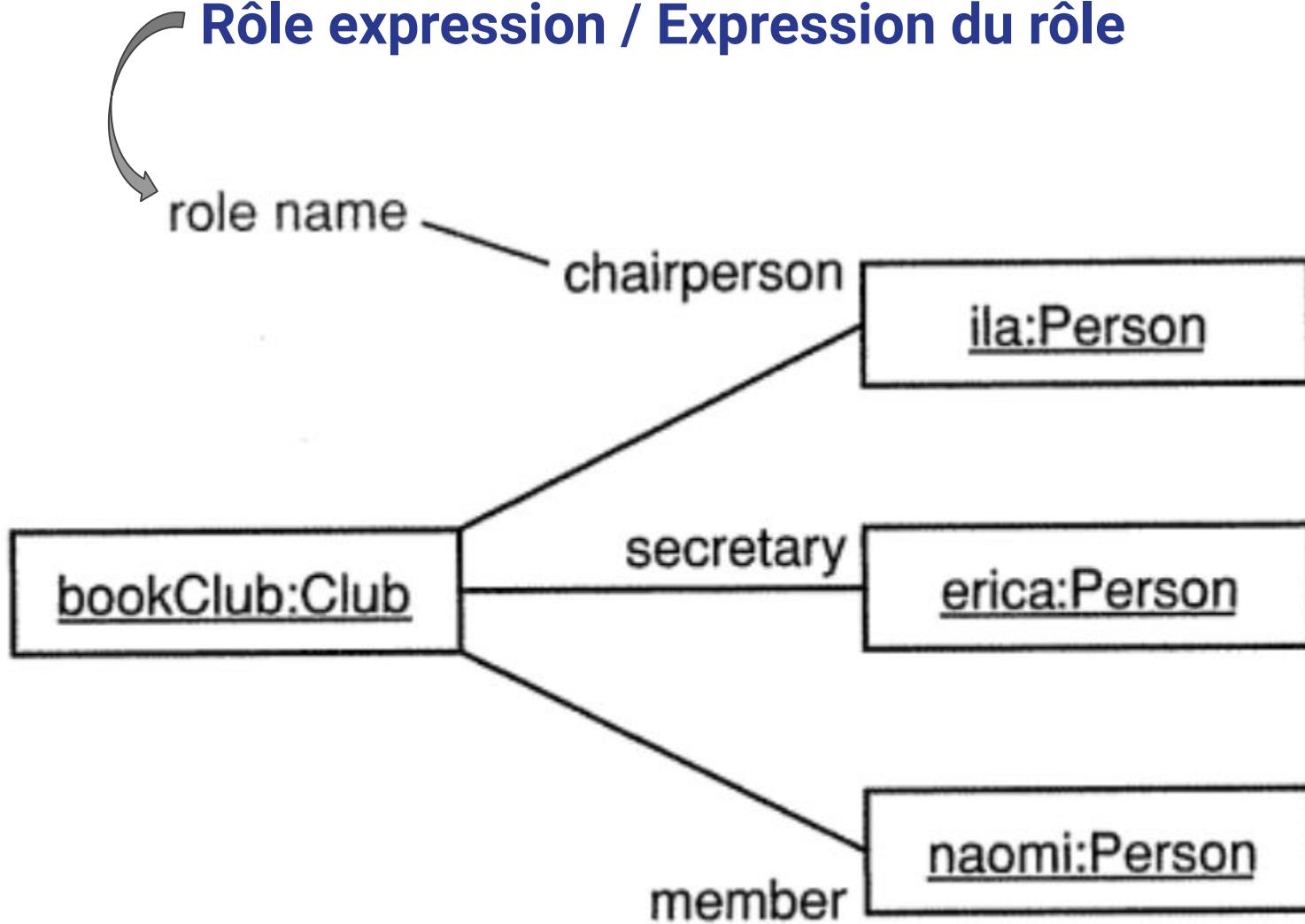
Ref: "Cours UML de L. Longuet"



Reminder: **Attributes** and **Methods** are more generally written in **camelCase**. (e.g., mot1Mot2Mot3Mot4Etc)

Object Diagram - Example 3

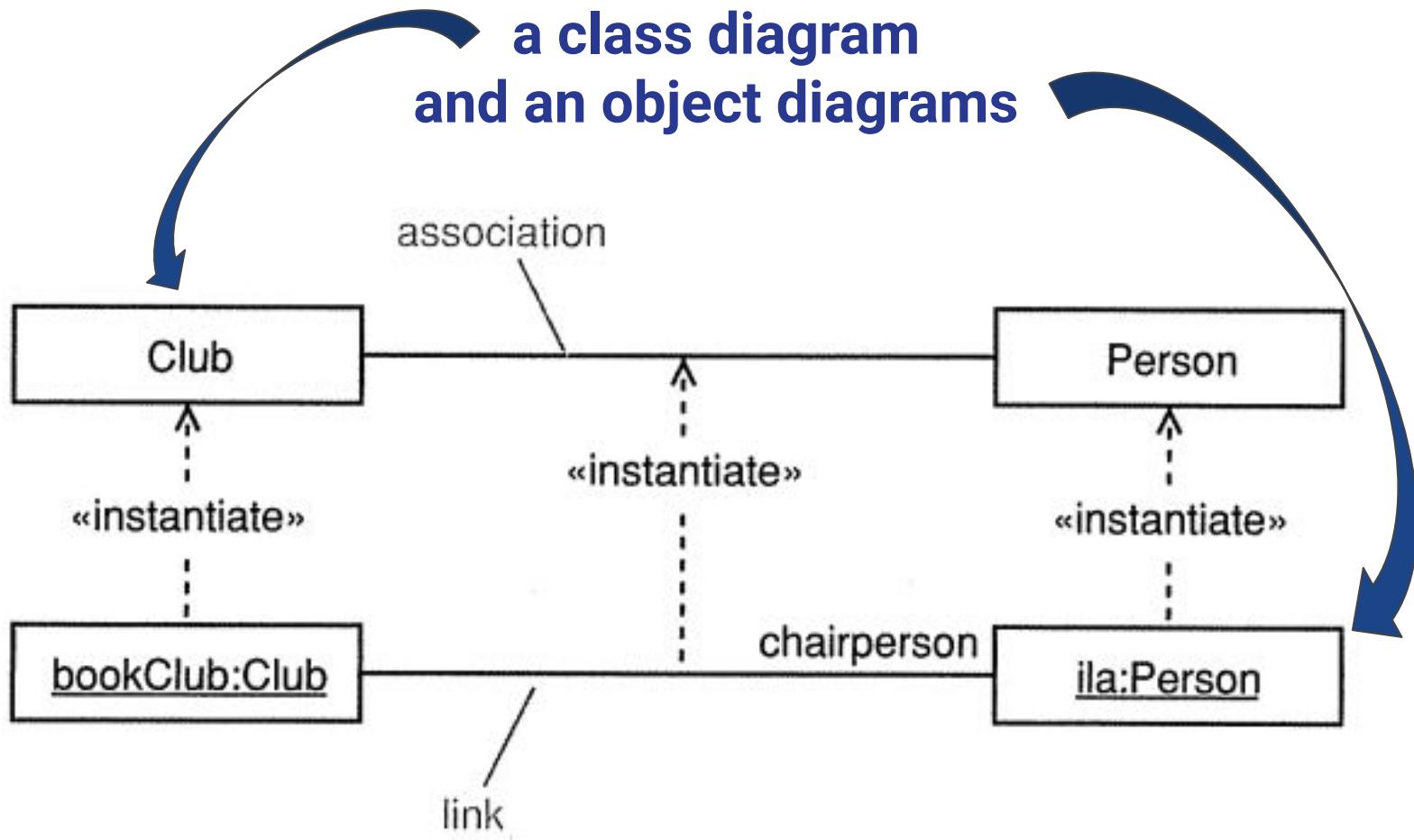
Rôle expression / Expression du rôle



Object Diagram - Example 4

If necessary: mixing both types of diagrams

a class diagram
and an object diagrams



Object diagram

Exercise

- Draw an object diagram based on two classes:
 - Polygon (no data and no attribute),
 - Point (with x and y real types attributes),
 - (no need to specify the type) ;
 - and with these instantiations:
 - 1 Polygon Object named ‘triangle’,
 - 3 Point objects all having a simple relation/link with ‘triangle’ named “PartOf” and with random (x,y) values.

“Break n°1 related
to the project”

SIMULATION

Once upon a time...

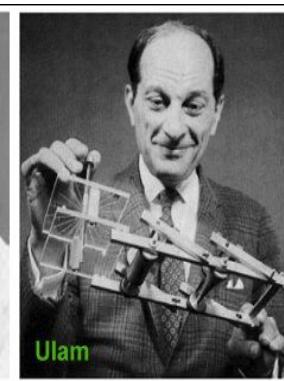
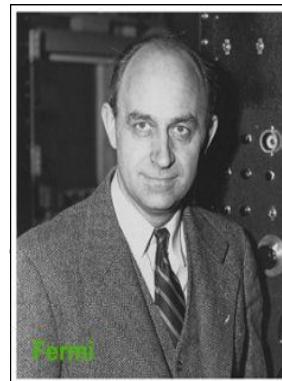
Simulation

SECRETS
d'HISTOIRE

- 2nd World War
 - Simulation (computer) within the framework of the **Manhattan project** in order to model the process of nuclear detonation,
 - Monte Carlo ;
- 1953
 - **Civil simulation** in theoretical physics,
 - 1st numerical simulation.

Manhattan Project

Fermi-Pasta-Ulam experiment



Example of SIMULATION

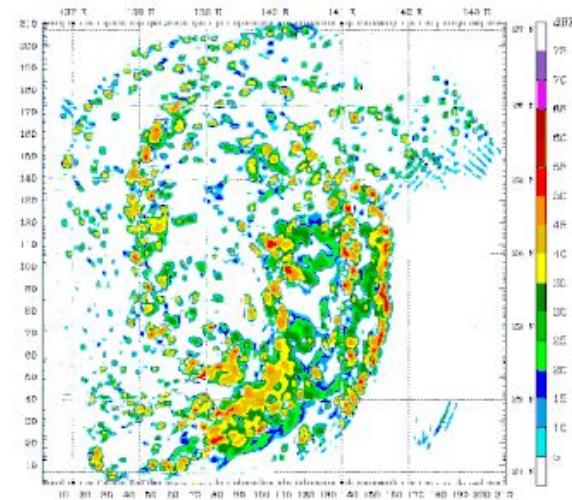
Climate & Weather

Simulation at the heart of forecasts (/Prévisions):

- Flood (/Inondation), Earthquake, etc.,
- (and even: simulation of economic wars!)

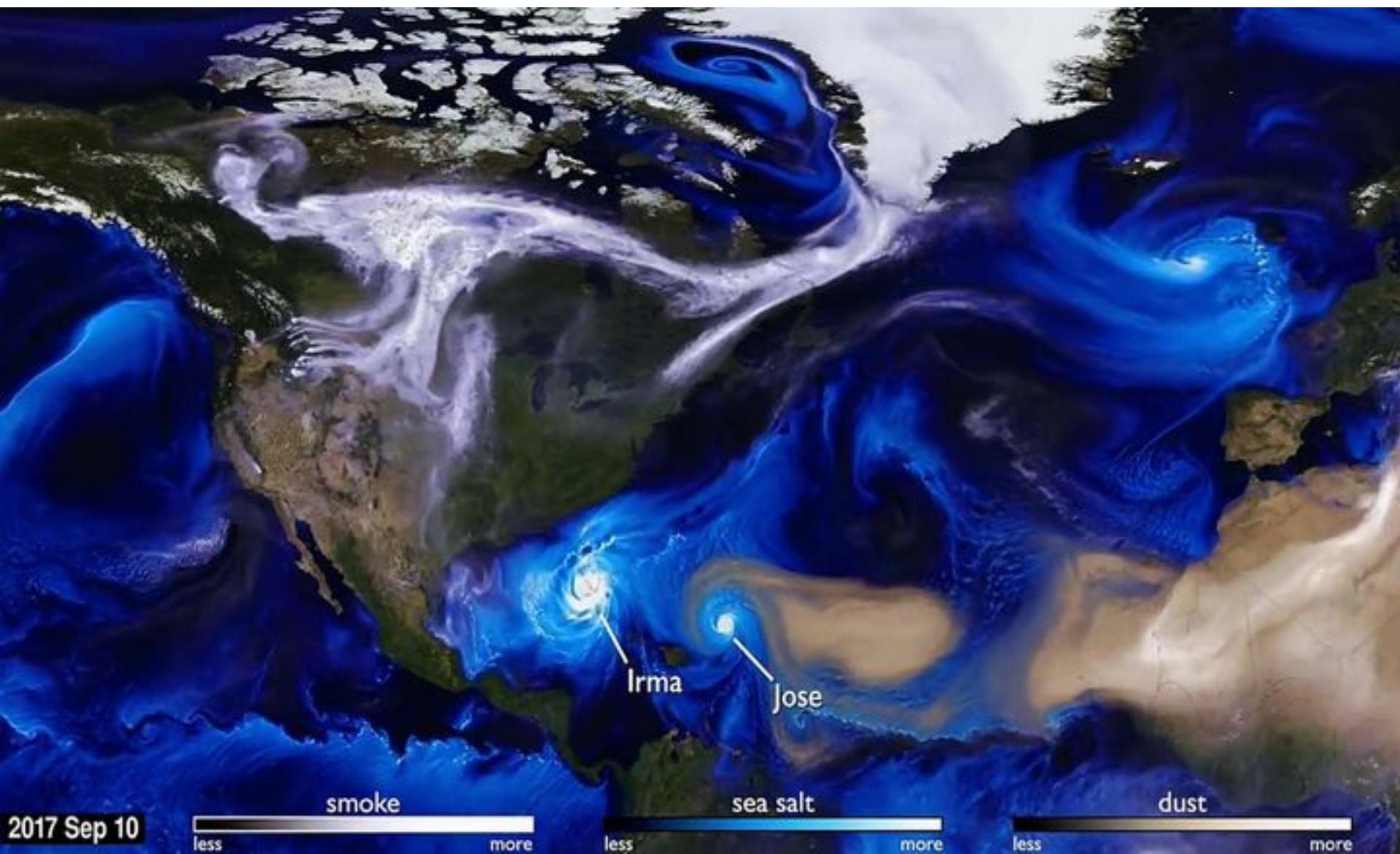
Scheme (a *simulation process life*):

- **Study of the System,**
- **Writing of a model** according to domain knowledge (functioning of the atmosphere, atmospheric phenomena, physical laws of gases, liquids and solids etc.),
- **Simulation development,**
- Target the **input data** of the simulation (e.g., current weather),
- **Analysis of outputs** (forecasts).



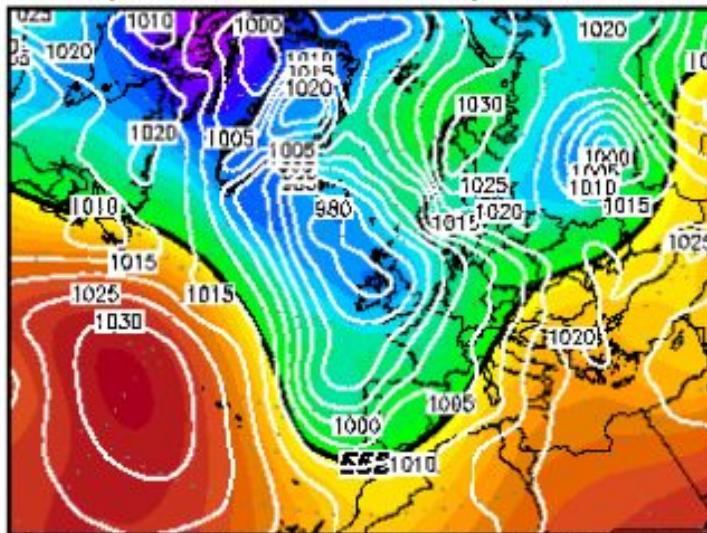
Example of SIMULATION

(Hurricanes/Ouragans - 2017)

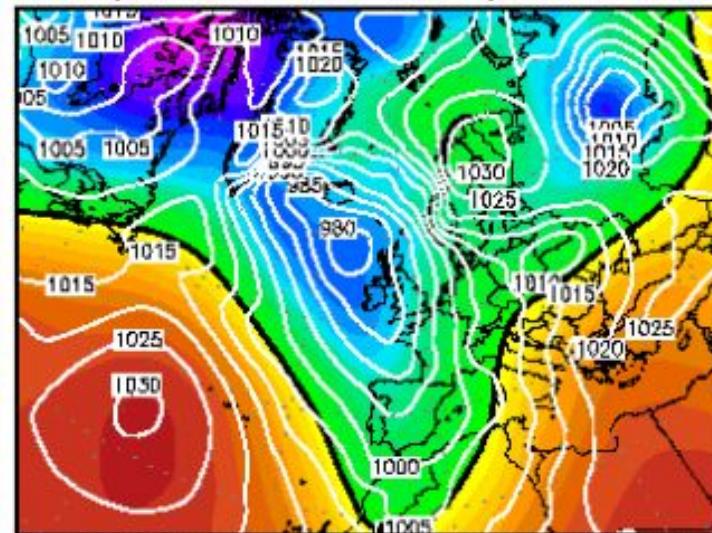


Example of SIMULATION (Weather)

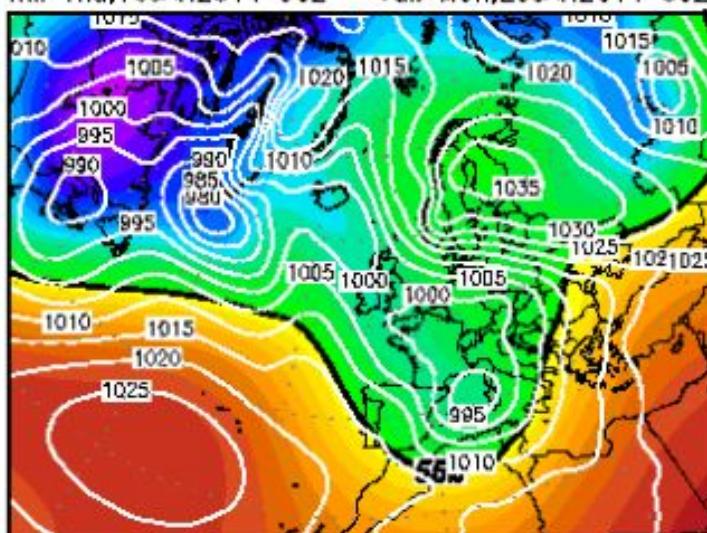
Ini: Thu,16JAN2014 06Z Val: Fri,17JAN2014 06Z



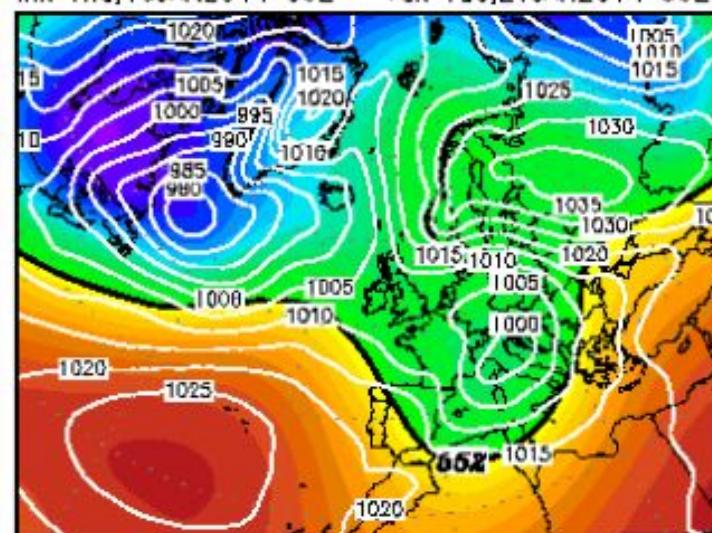
Ini: Thu,16JAN2014 06Z Val: Sat,18JAN2014 06Z



Ini: Thu,16JAN2014 06Z Val: Mon,20JAN2014 06Z



Ini: Thu,16JAN2014 06Z Val: Tue,21JAN2014 06Z



SIMULATION & Computing Power / Puissance de Calcul

- **Supercomputers are very often dedicated to simulation** in the following areas:
 - Weather and Climate,
 - Aerodynamics,
 - The resistance of the materials,
 - Acoustics (e.g., ISEN Labo Samba),
 - Nuclear (military),
 - Biology (evaluation of complex models, e.g., animals, COVID-19),
 - At the movies (e.g. ; War in the Lord of the Rings, GoT etc.),
 - Video games,
 - In IT (testing real-time algorithms),
 - In Optimization [etc.]
 - ... and even more and more in geopolitical!

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,299,072	415,530.0	513,854.7	28,335
2	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	Sierra - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	Tianhe-2A - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express- 2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482

Unité de performance des processeurs

Nom	FLOPS
yottaFLOPS	10^{24}
zettaFLOPS	10^{21}
exaFLOPS	10^{18}
pétaFLOPS	10^{15}
téraFLOPS	10^{12}
gigaFLOPS	10^9
mégaFLOPS	10^6
kiloFLOPS	10^3

1 FLOPS : 1 floating point operation per second.

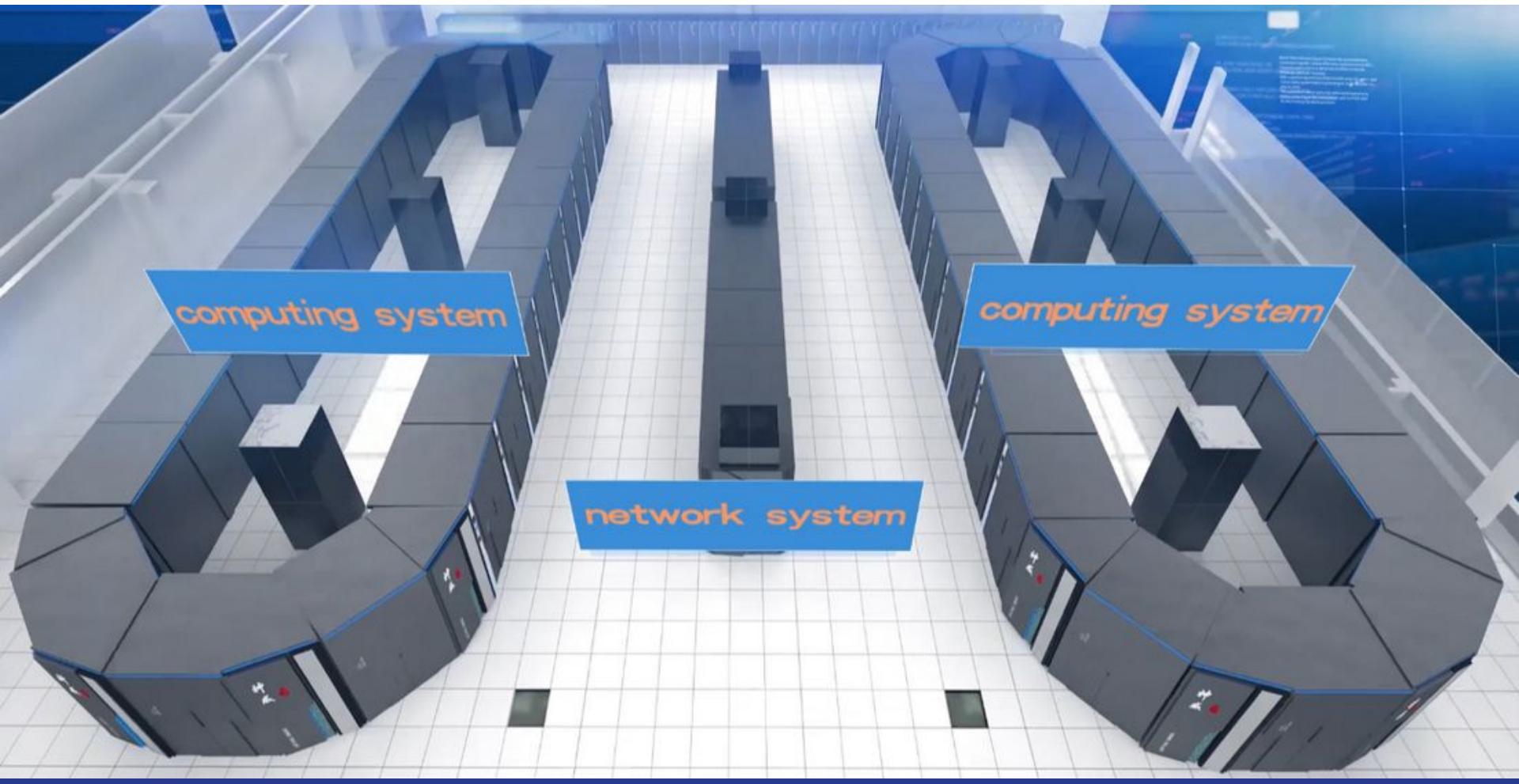
15	PANGEA III - IBM Power System AC922, IBM POWER9 18C 3.45GHz, Dual-rail Mellanox EDR Infiniband, NVIDIA Volta GV100, IBM Total Exploration Production France
----	--

TaihuLight

(n°5 TOP500 au 06/2020)

The first above 100 pFLOPS

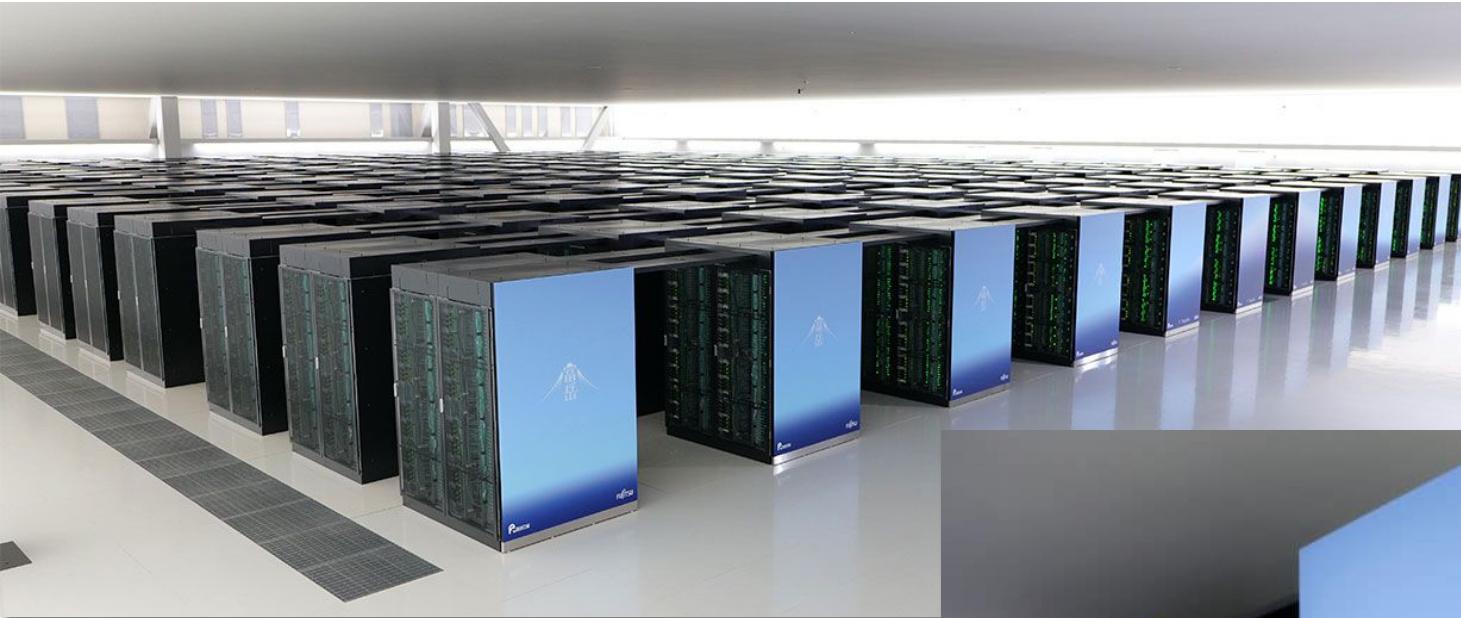
/ Le premier au dessus des 100 pFLOPS



Fugaku

(n°1 TOP500 au 06/2020)

More than 7 Millions of cores (ARM architecture)



3.

Unified Modeling Language (UML)

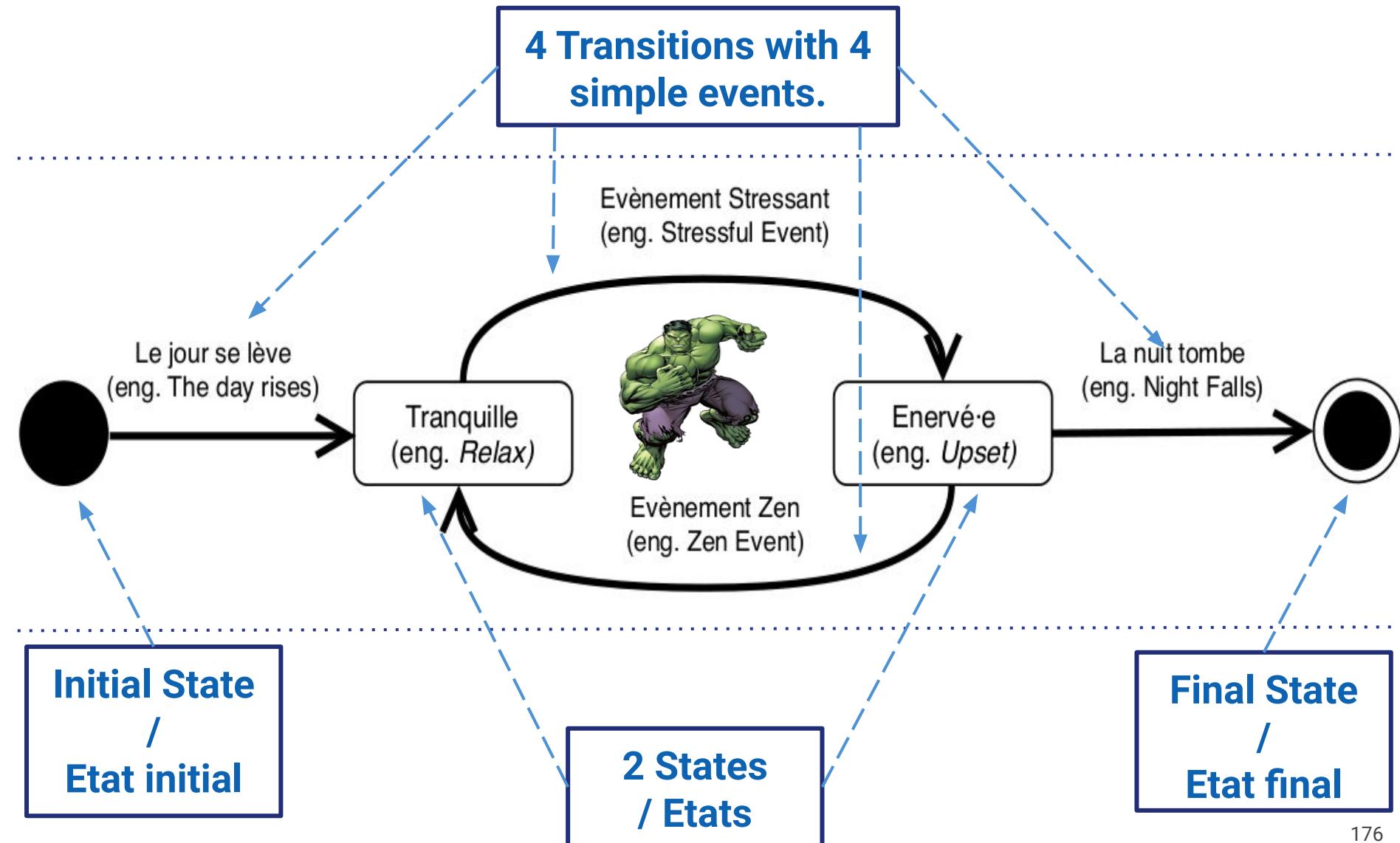
3.4. State Machine Diagram / Le diagramme d' Etats - Transitions

State Machine Diagram / Diagramme d'états - transitions

- **Objective:** Model the behavior of objects based on events;
- **States / Etats:** condition or situation of an object during which:
 - it satisfies a certain condition, or
 - it performs a certain activity, or
 - it is waiting for a certain event;
- **Transitions** (from one state to another):
 - **Event**
 - **[Condition(s)]**
 - **(Parameter)**
 - **/Actions**

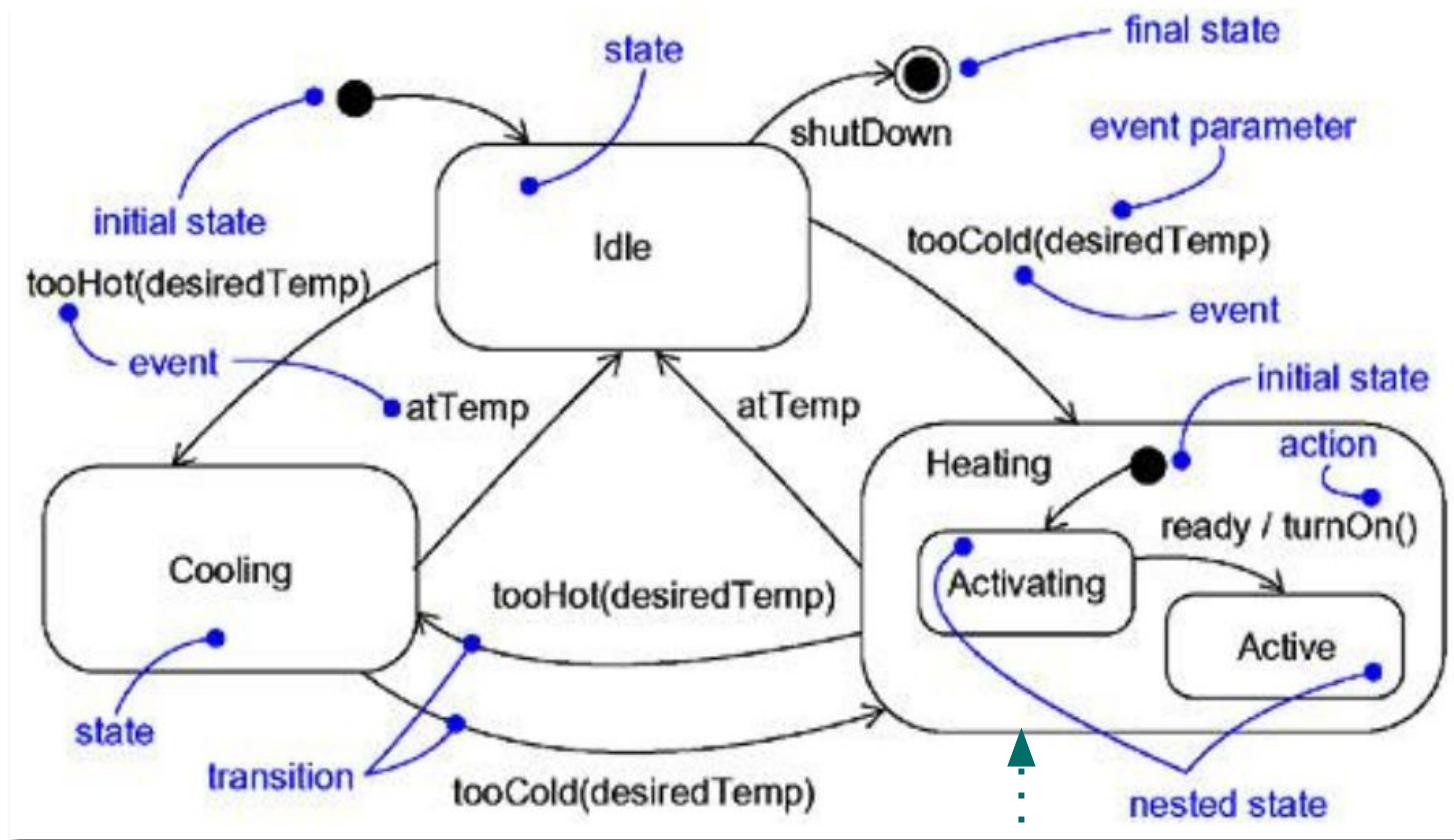
aClass
-aState: int
+ChangeState():void

Diagramme d'états - transitions (eng. state machine diagram)



State Machine Diagram

/ Diagramme d'états - transitions

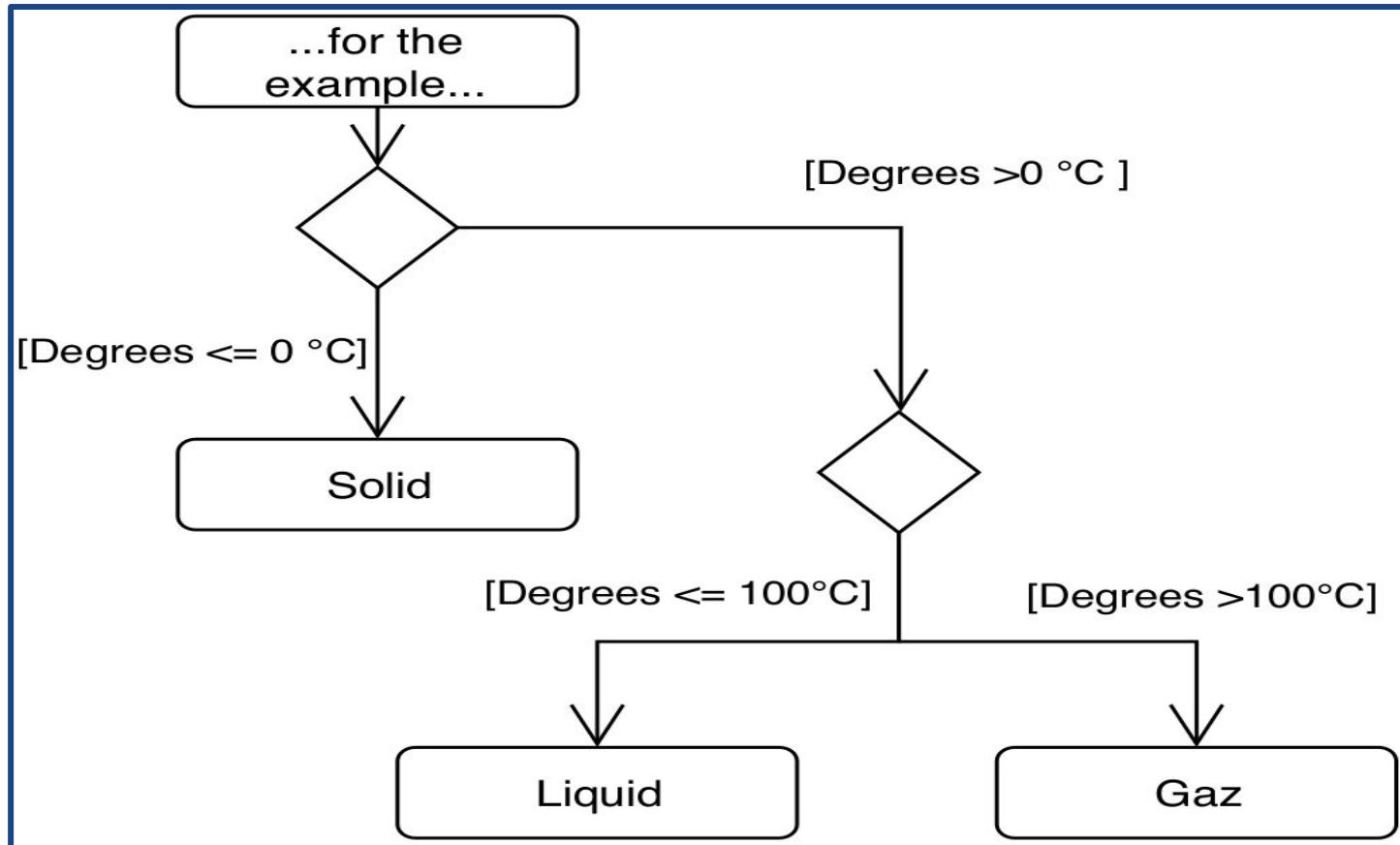


air conditioning example

We will see later how to model such factoring “composite state”.

(diamond-shaped) dynamic [condition]

water state example



How to write a transition

From one state to another



Event [condition] (parameter) / action



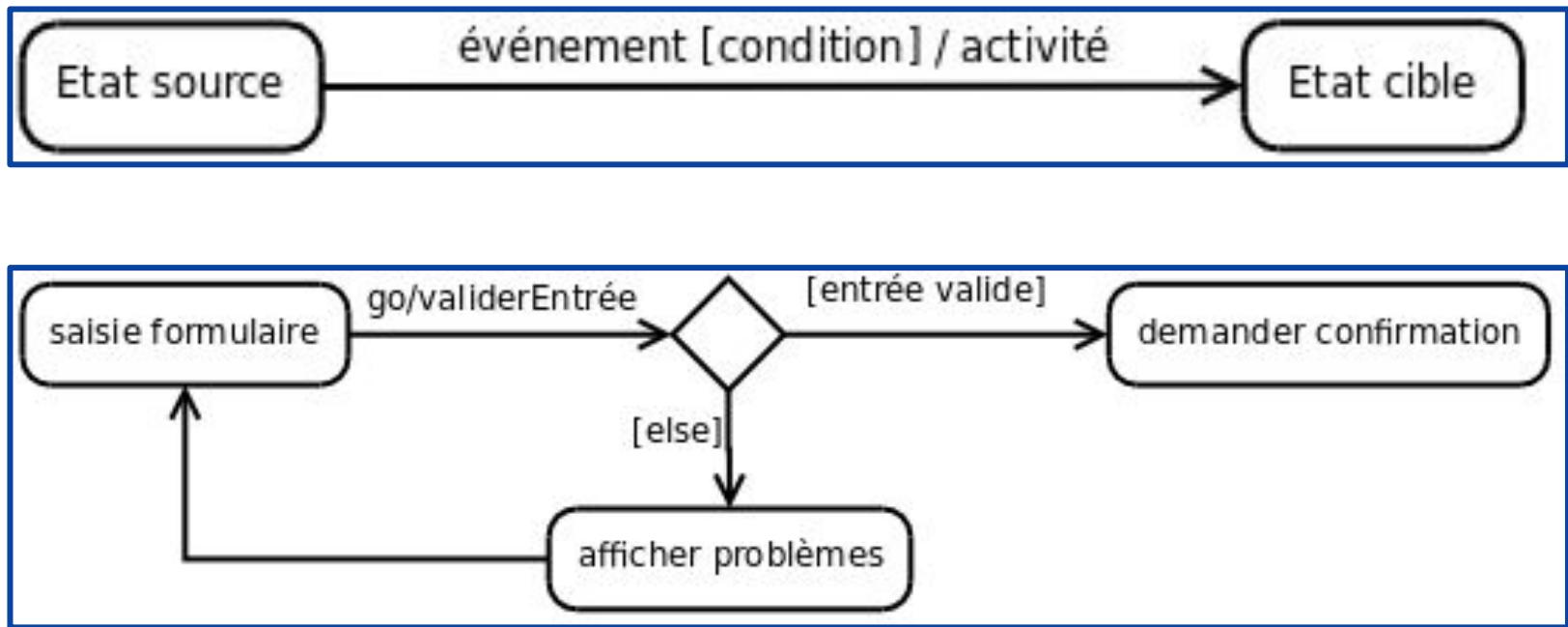
Événement [condition] (paramètres) / action

How to write a transition



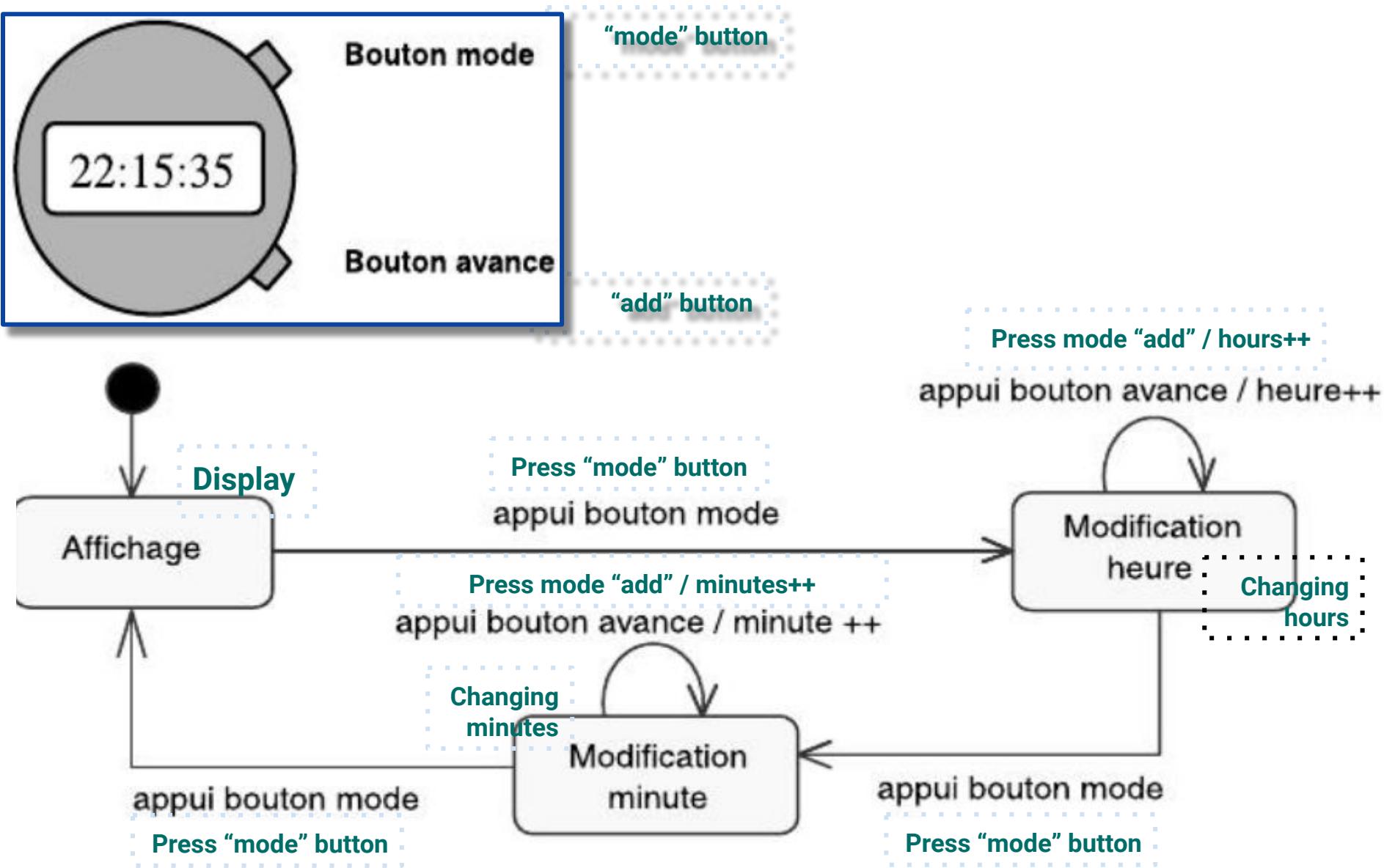
Event [condition] (parameter) / action

Examples



State Machine Diagram / Diagramme d'états - transitions

Example "Watch"



State Machine Diagram

Composite States (also called Super State)

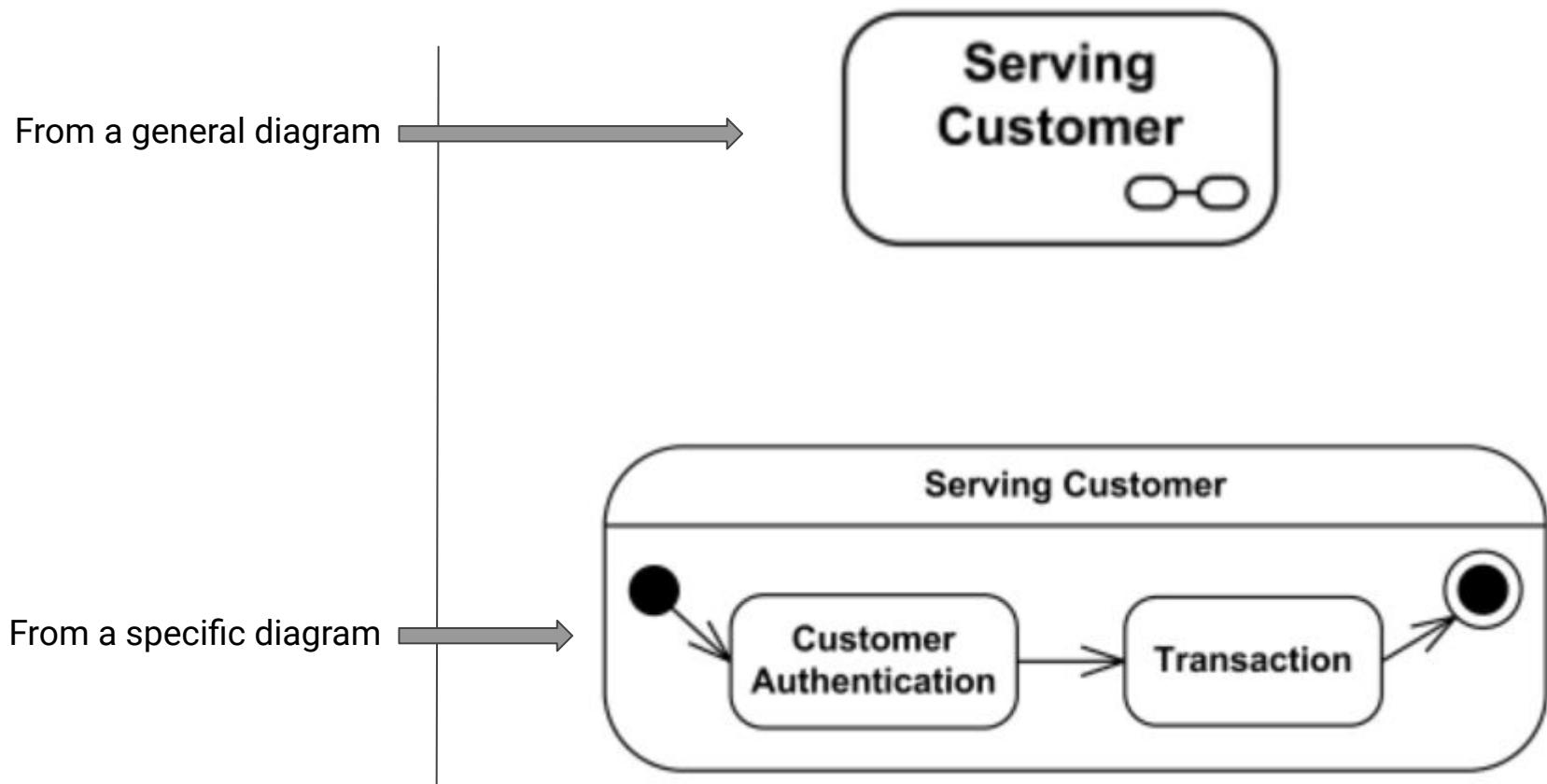
Etat Composite ou "Super-Etat"
(eng. Composite State)



State Machine Diagram

Composites States

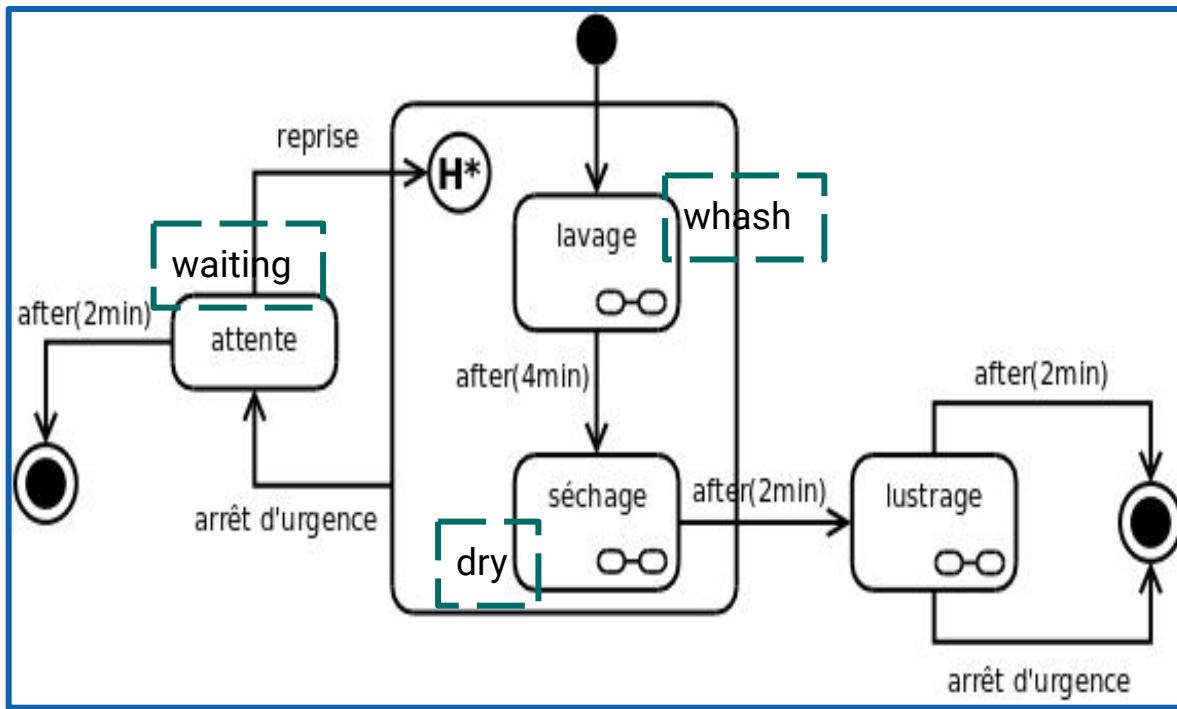
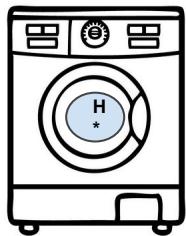
Examples



Advanced State Machine Diagram the History pseudostate

H*

From an external transition, it saves the current state to resume the process from where it stopped.

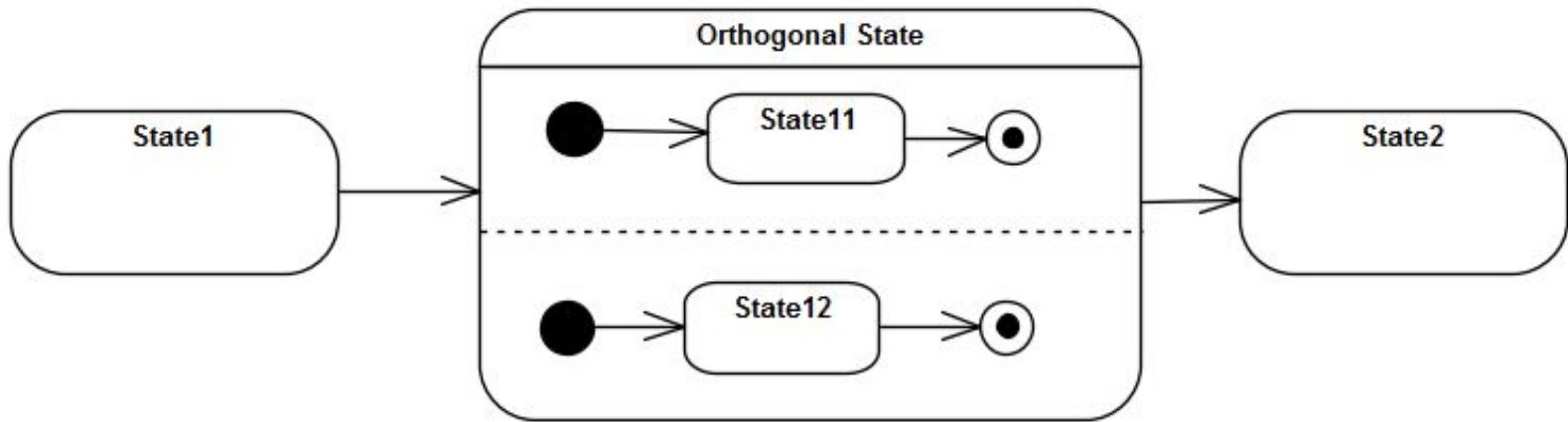


Pic ref: Cours Laurent Audibert

On the example above: allows you to resume washing (or drying) where it was before the interruption.

Advanced State Machine Diagram

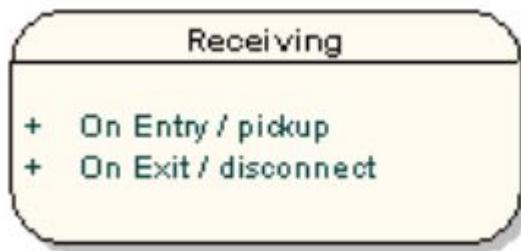
The Orthogonal State



For concurrent mechanisms. **They must all reach their final state for the composite state to be considered complete.**

Advanced State Machine Diagram

State Actions



- + On Entry (or directly "Entry")
- + On Exit (or directly "Exit")

You can also have "+ do" (or directly "do")

State Machine Diagram

- EXERCICE SHOP -



A simplified cash drawer has 3 modes: **Customer Waiting**, **Item Registration** and **Payment Waiting**. They can alternate following:

- the entry of an item by the seller (e.g., barcode scan),
- when the seller has registered all the items,
 - (there can indeed be several...);
- Entering payment.

Goal: Draw the related state machine diagram.



Diagramme d'états-transitions

- EXERCICE SHOP -



Un tiroir-caisse simplifié a 3 modes : Attente de client, Enregistrement des articles et Attente de paiement. Ils peuvent alterner suite à :

- la saisie d'un article par le vendeur (e.g., scan code-barres),
- le moment où le vendeur a enregistré tous les articles,
 - (il peut en effet y en avoir plusieurs...);
- la saisie du paiement.

But : Dessiner le diagramme d'états-transitions du tiroir-caisse.



State Machine diagram

- Library -



In order to manage a library, you start to model everything that happens in it, in particular, by the representation of a state machine diagram on the 3 states of a book (**Available**, **Borrowed** and **Non-Returned**) knowing that the events that concern him are as follows:

- **Restitution**,
- **Borrow** (for one month),
- **Robbery**,
- **OneMonthIsPassed**.

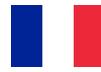
Events may be linked to more than a couple of states.



Source image : <https://www.planetminecraft.com/>

Diagramme d'états-transitions

- EXERCICE *BIBLIOTHÈQUE* -



Afin de gérer une bibliothèque, vous vous mettez à modéliser tout ce qui s'y passe, notamment, par la représentation d'un diagramme d'états-transitions sur les 3 états d'un livre (**Disponible**, **Emprunté** et **NonRendu**) sachant que les événements qui le concernent sont les suivants :

- **Restitution**,
- **Emprunt** (pour un mois),
- **Vol**,
- **UnMoisEstPassé**.

Il se peut que des événements se retrouvent liés à plus d'un couple d'états.



Source image : <https://www.planetminecraft.com/>

Diag Etats-Transitions - EXERCICE telephoneBooth -

Dessinez le diag. d'états-transitions d'une cabine téléphonique. Vous pouvez mutualiser le décrochage par un "super-état".

Les états du téléphone sont les suivants :

- Raccroché
- AttentePièce
- AttenteNuméro
- AttenteDécrochageAuBoutDeLaLigne
- EnCommunication
- FinDeCommunication

Rappel sur les transitions :



Créer les transitions en combinant les éléments suivants :

Evènements :

- raccrocherCombiné
- décrocherCombiné
- nouveauNuméro
- débuterCommunication
- pièceEnPlus
- vérifierArgentRestant

Conditions :

- crédits>0.2€ (Minimum attendu)
- crédits<0.21€
- numéroFaux - numéroValide
- créditSuffisant - créditInsuffisant

Activités :

- rendrePièces
- initCrédits=0
- taxer (débiter sur somme initiale)

Diag State Machine - EXERCICE telephoneBooth -

Draw the state machine diagram according to the states and transitions below. Try to factor through a superstate.

The states of the phone are the following:

- HangUp
- WaitingCoin
- WaitingNumber
- WaitingPhonePickUpAt TheEndOfTheLine
- InCommunication
- EndOfCommunication

Help on transitions :



Create the transitions by combining the following:

Events :

- hangUp
- pickUpPhone
- newNumber
- startCommunication
- someCoins
- checkRemainingMoney

Conditions:

- credits>0.2€ (minimum to phone)
- credits<0.21€
- falseNumber - validNumber
- sufficientCredits - insufficientCredits

Activities:

- givingChange
- initCrédits=0
- tax (taken at the start)



"Break n°2"

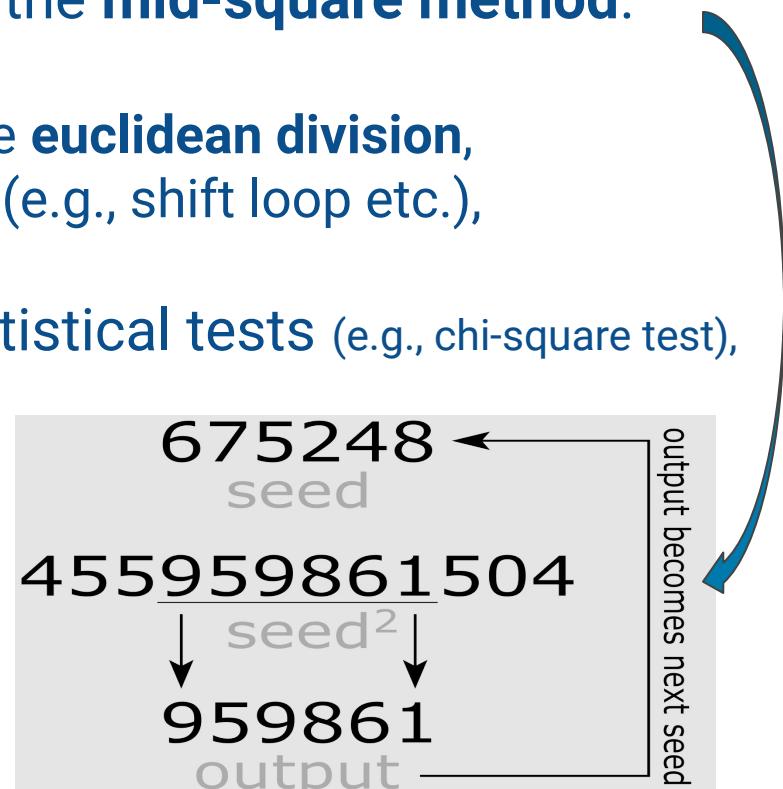
SIMULATION & Pseudo-random numbers



Pseudorandom number generators



- There is a **real competition** on who will have the best **pseudo-random number generator**.
- A good generator is **important for simulations** that require a lot of numbers.
- First pseudo-random number generation algorithm by **John Von Neumann (1940)**: **the mid-square method**.
- Later:
 - Methods based on the rest of the **euclidean division**,
 - Methods with **binary operations** (e.g., shift loop etc.),
 - **Merging of generators**;
- Their quality is determined by statistical tests (e.g., chi-square test),
- Example of a good generator:
 - **Mersenne Twister**.
 - **Future: quantum machines?**



Monte Carlo

Method and Simulation
(Static and Dynamic)

- Invented by **John Von Neumann** and **Stanislaw Ulman**,
- 1st simulation of the **name of the Casino**
- Focused on the generation of pseudo-random numbers:
 - **stochastic process.**

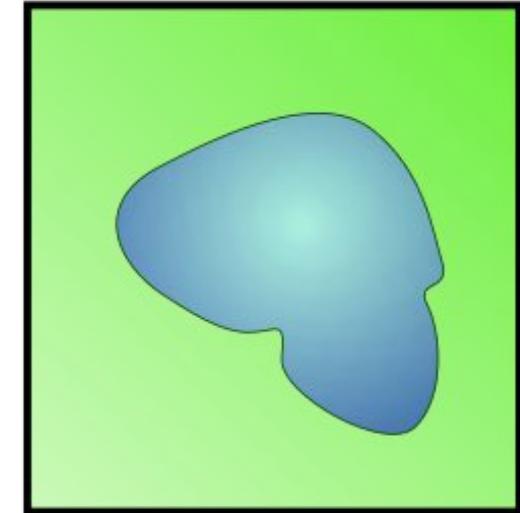
... examples in the practical work.



Lake approximation with the Monte Carlo method

To find the area of a lake, an army is asked to fire cannonballs (just that!). We have:

- X cannon shots fired **randomly** on the area,
- N is the number of balls that did **NOT** make a “**splash!** / **plouf!**”,
- $X - N$ is therefore the number of balls which made “**Splash!** **Plouf!**”.

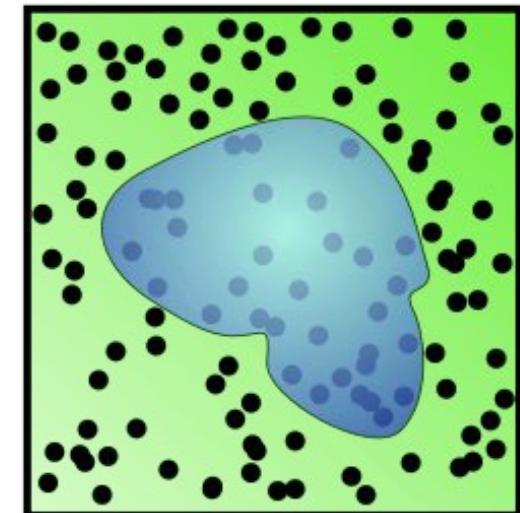


We can deduce an approximation of the area of the lake.

(area == superficie; lake = lac; terrain = (here) ‘land’: all the square area)

$$\frac{\text{superficie}_{\text{terrain}}}{\text{superficie}_{\text{lac}}} = \frac{X}{X - N}$$
$$\Rightarrow \text{superficie}_{\text{lac}} = \frac{(X - N)}{X} \times \text{superficie}_{\text{terrain}}$$

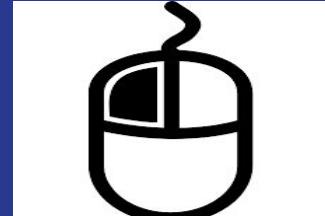
The land area (superficie terrain) is of course known here.



Practical Work

Practical
Work 7

Monte Carlo Method





π approximation with the Monte Carlo method

- Main idea: we generate points at random in a domain of known area. When the number of points placed tends to infinity, the proportion of points "fallen" in a subdomain gives its area. Here, we will take the opportunity to calculate π .
- Goal: Get a value close to one hundredth of π by developing the following algorithm with n very large:

```
quarterOfTheDiscArea = 0 ;
```

```
For i from 1 to n Do
```

```
    x <= randomly chosen in [0;1] ;
```

```
    y <= randomly chosen in [0;1] ;
```

```
    If  $x^2 + y^2 \leq 1$  Do
```

```
        quarterOfTheDiscArea
```

```
            = quarterOfTheDiscArea + 1 ;
```

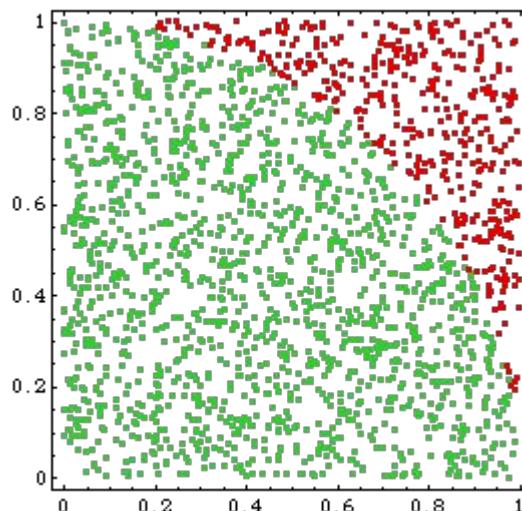
```
    End If
```

```
End For
```

```
approxPi = (4 * quarterOfTheDiscArea) / (n) ;
```

BONUS 1: Be more precise!

BONUS 2 : use a graphic API to represent the process as in the pic above.



Approximation de π par la méthode de Monte Carlo

- Idée de méthode de Monte Carlo : On génère des points au hasard dans un domaine dont l'aire est connue. Lorsque le nombre de points placés tend vers l'infini, la proportion des points « tombés » dans un sous-domaine permet d'obtenir son aire. Ici, on va en profiter pour calculer π .
- Votre but : arriver à une valeur proche du centième de π en développant l'algorithme suivant avec n très grand :

quartDisque = 0 ;

Pour i de 1 à n **Faire**

 x tirée aléatoirement dans [0;1] ;

 y tirée aléatoirement dans [0;1] ;

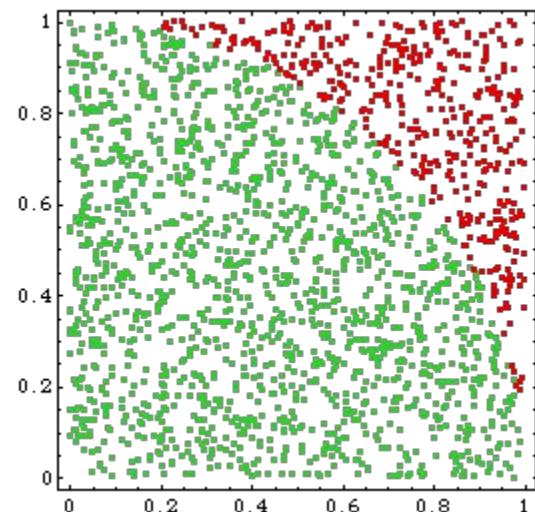
Si $x^2 + y^2 \leq 1$ **Faire**

 quartDisque = quartDisque + 1 ;

Fin Si

Fin Pour

approxPi = (4*quartDisque)/(n) ;



BONUS 1 : Cherchez à affiner π en fonction de n .

BONUS 2 : Avec une interface graphique, affichez différemment les points du cercle et de l'extérieur du cercle.

Π approximation

Practical
Work 7

CORRECTION

End of “Break n°2”

SIMULATION

Back to the lecture / Retour au cours



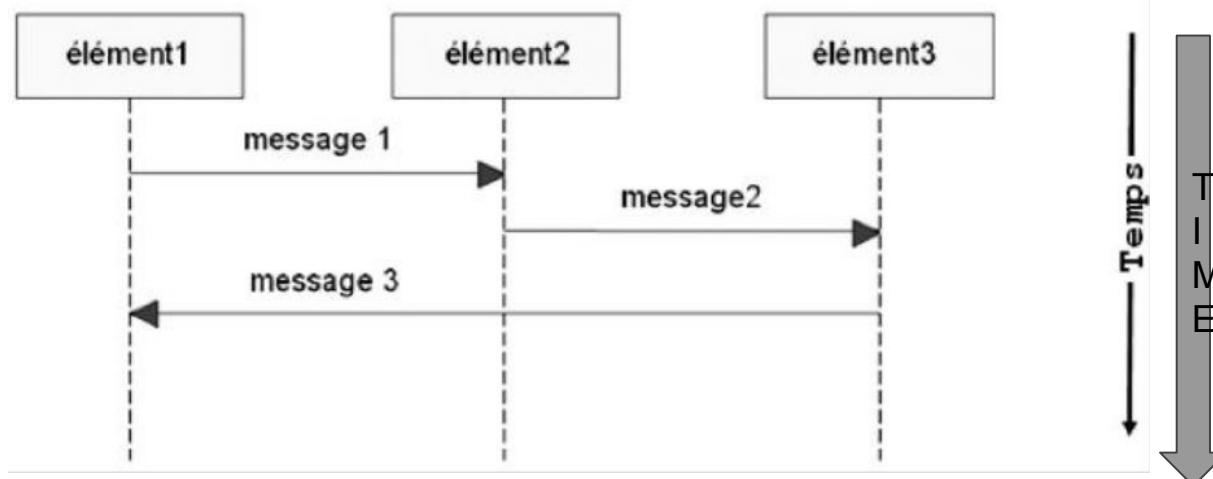
3.

Unified Modeling Language (UML)

3.5. Sequence diagram (*fr.* diagramme de Séquence)

Sequence Diagram

- Centered on the (line of) life of **objects**,
 - Creation and destruction sometimes specified;
- Show the **interactions between objects**,
- **Vertical timeline** (from top to bottom),
- Importance of objects sorted on the **horizontal axis** (from left to right)
 - The starting element to the right,
 - from the most important to the - important,

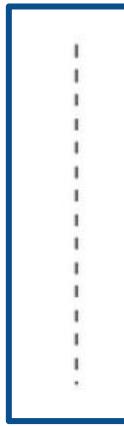


Sequence Diagram

BASIC ELEMENTS

Dashed line:

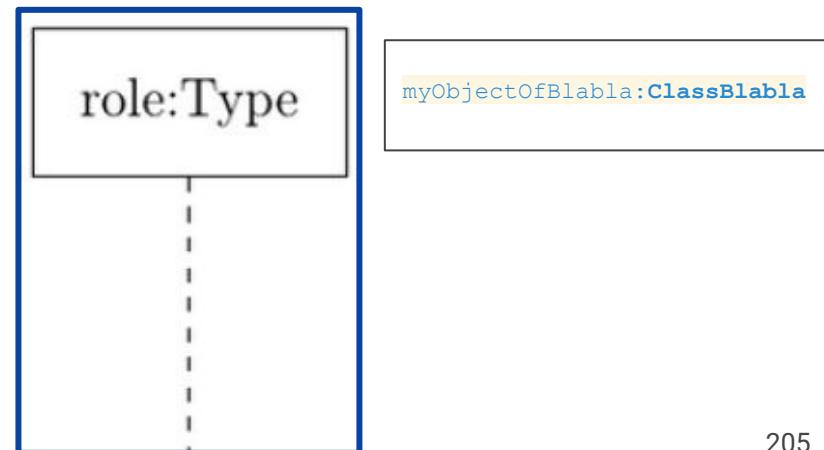
Object lifeline.



```
class myRandomClass
{
    void myRandomFunction()
    {
        ClassBlabla myObjectOfBlabla = new ClassBlabla();

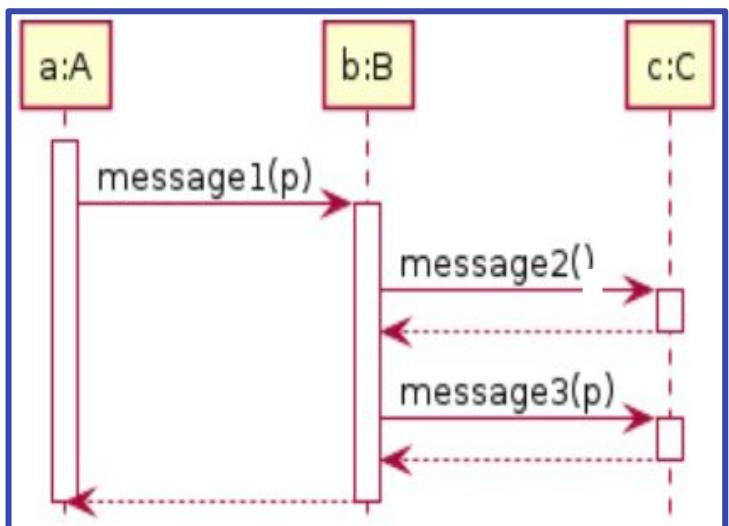
        // [...]
        //
    }
}
```

-
- > A role and / or a type is **required**.
 - > The ':' is **systematic**.
 - > The **role** identifies the object, it is often the name of the instance / **object**.
 - > The **Type** is often the **name of the class**.

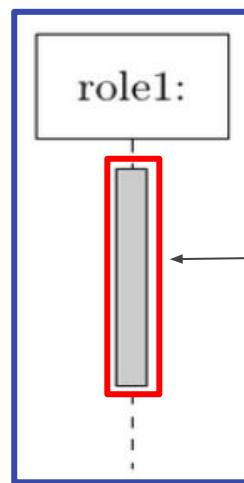


Sequence Diagram

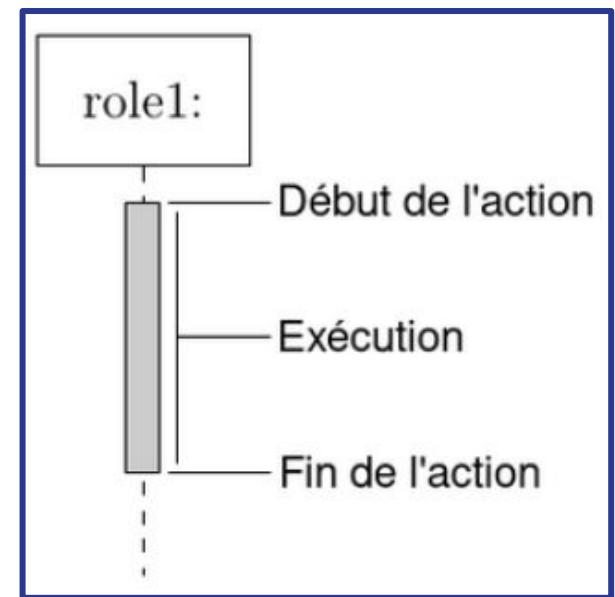
EXAMPLE



BASIC ELEMENTS

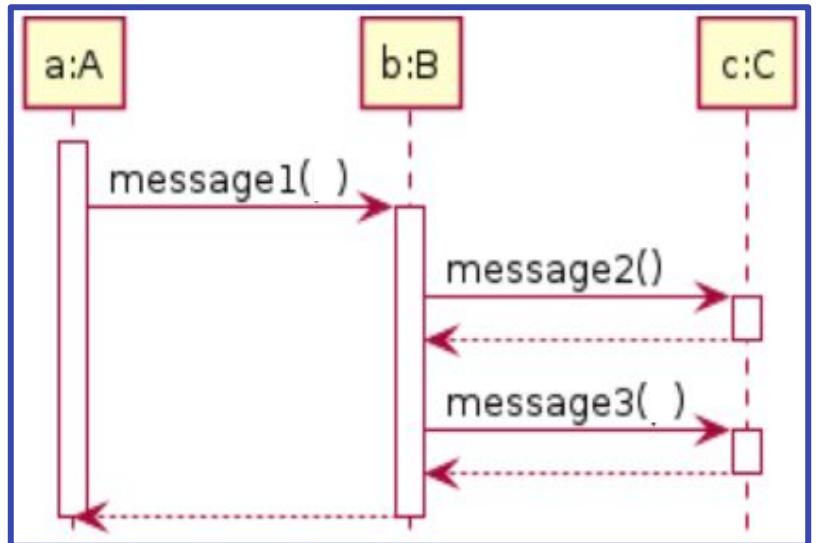


Rectangle representing the **amount of time taken by an action.**

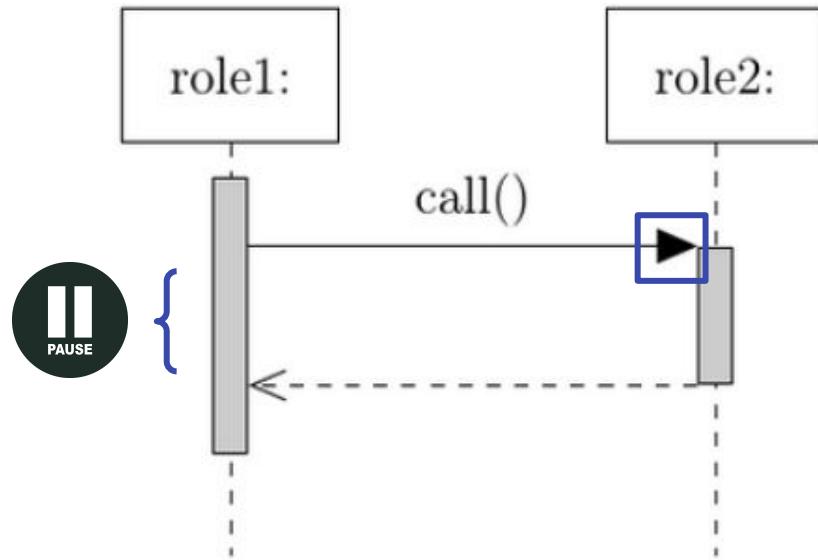


Sequence Diagram

EXAMPLE



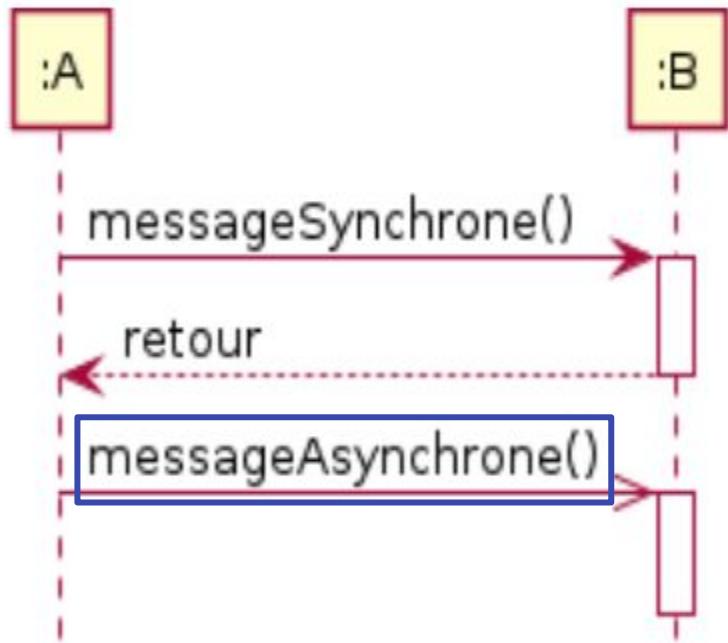
BASIC ELEMENTS Synchronous messages



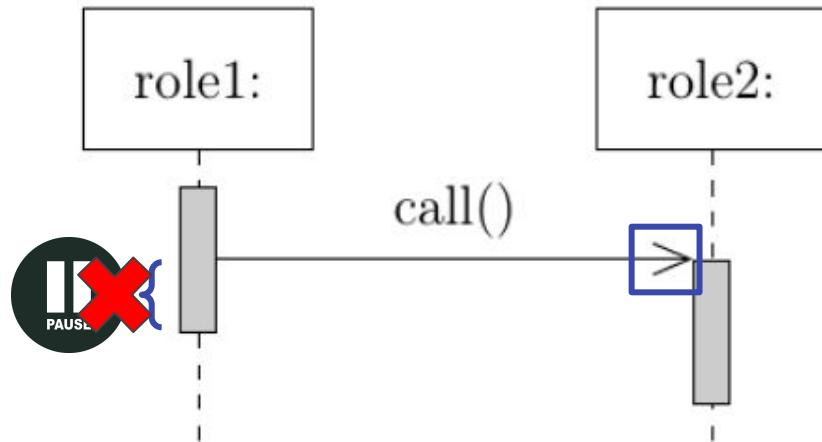
- The call **locks** role1 while role2 works
- **Synchronous** message is the most common.
-

Diagramme de séquence

EXAMPLE



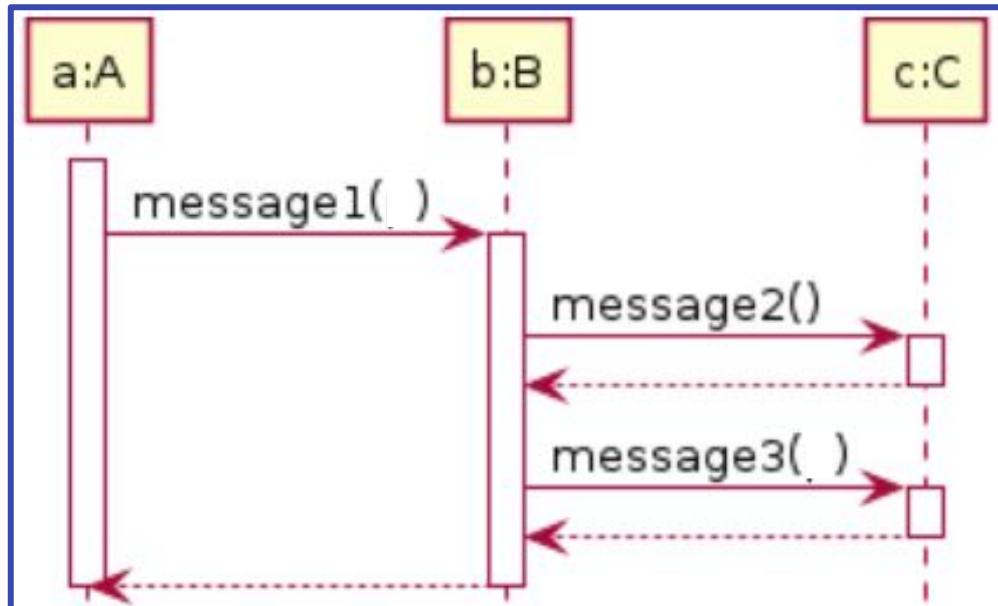
BASIC ELEMENTS Asynchronous messages



- The call **does NOT lock** role1,
- Role1 can **continue** to execute its code without a role2 return.
- **Asynchronous** message is rare.

Sequence diagram

Synchronous messages JAVA example



```
public class A {
    protected B b;

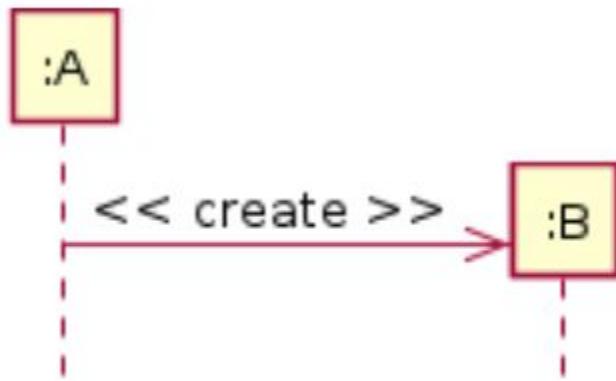
    public void firstMethodLaunchInTheMain()
    {
        b.message1();
    }
}
```

```
public class B {
    protected C c;
    public void message1()
    {
        c.message2();
        c.message3();
    }
}
```

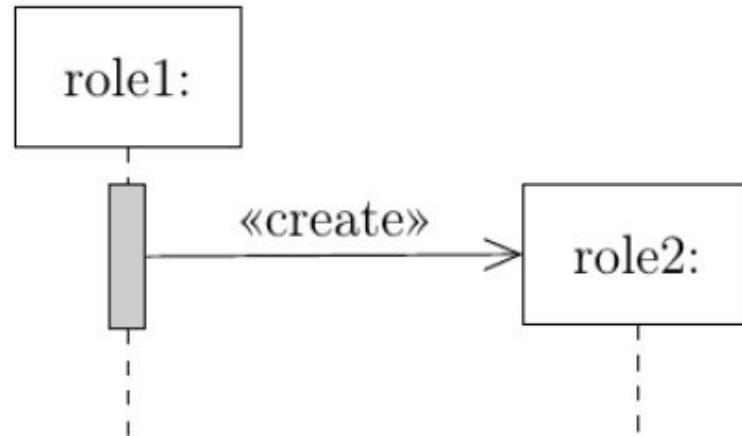
```
public class C {
    public void message2()
    {
    }
    public void message3()
    {
    }
}
```

Sequence diagram

EXAMPLE



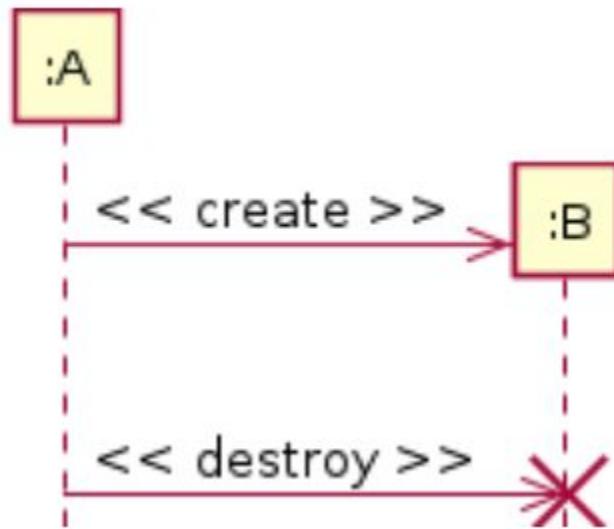
BASIC ELEMENTS Object instantiation



- The instantiation of one object by another is realised by:
 - A message targeting directly the “rectangle” of the new object,
 - Arrow name: **<<create>>**.

Sequence diagram

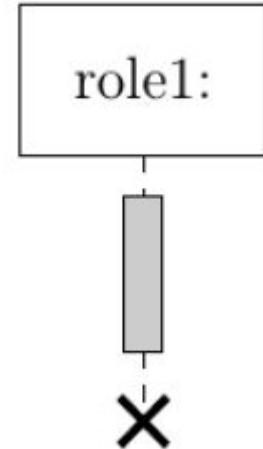
EXAMPLE



BASIC ELEMENTS Object destruction



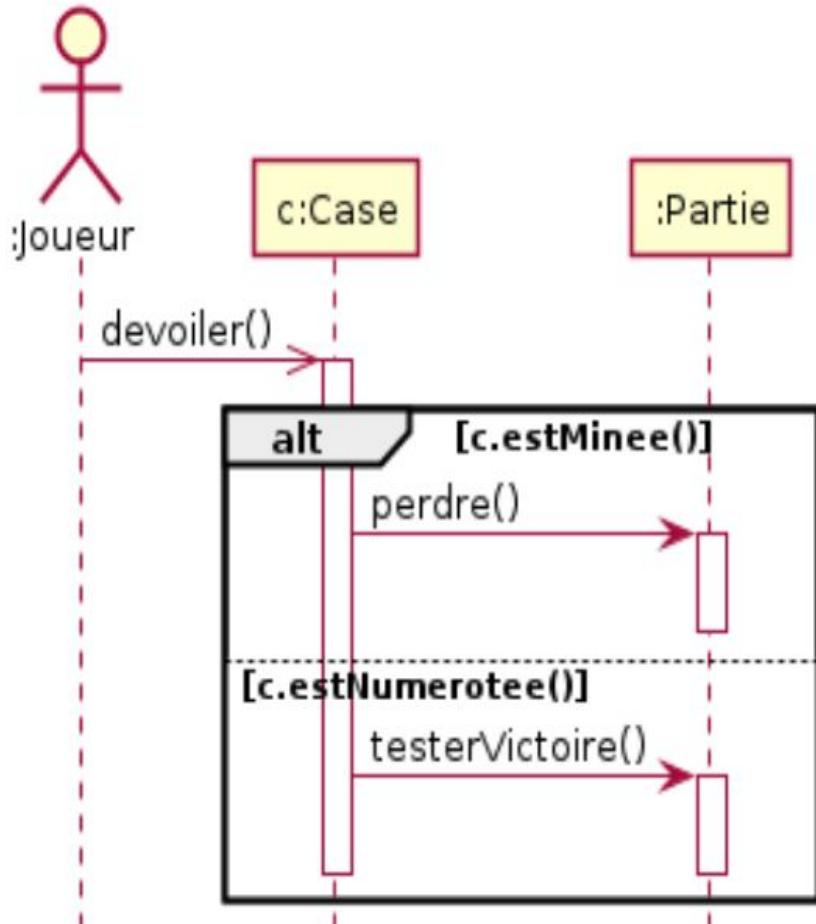
- **Explicit** destruction by another object.



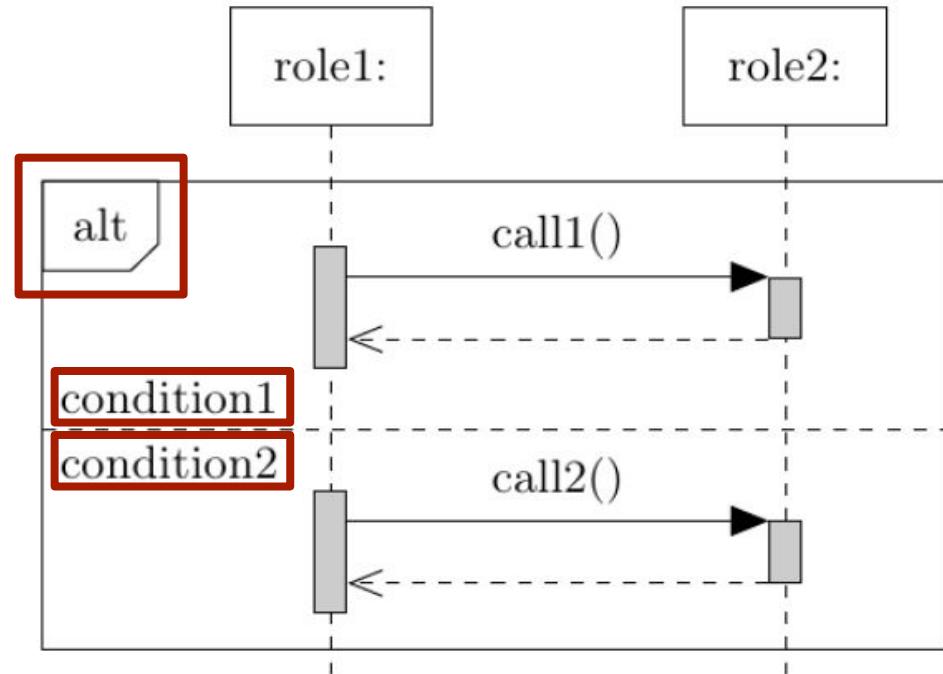
- **Implicit** destruction of the object.

Sequence diagram

EXAMPLE



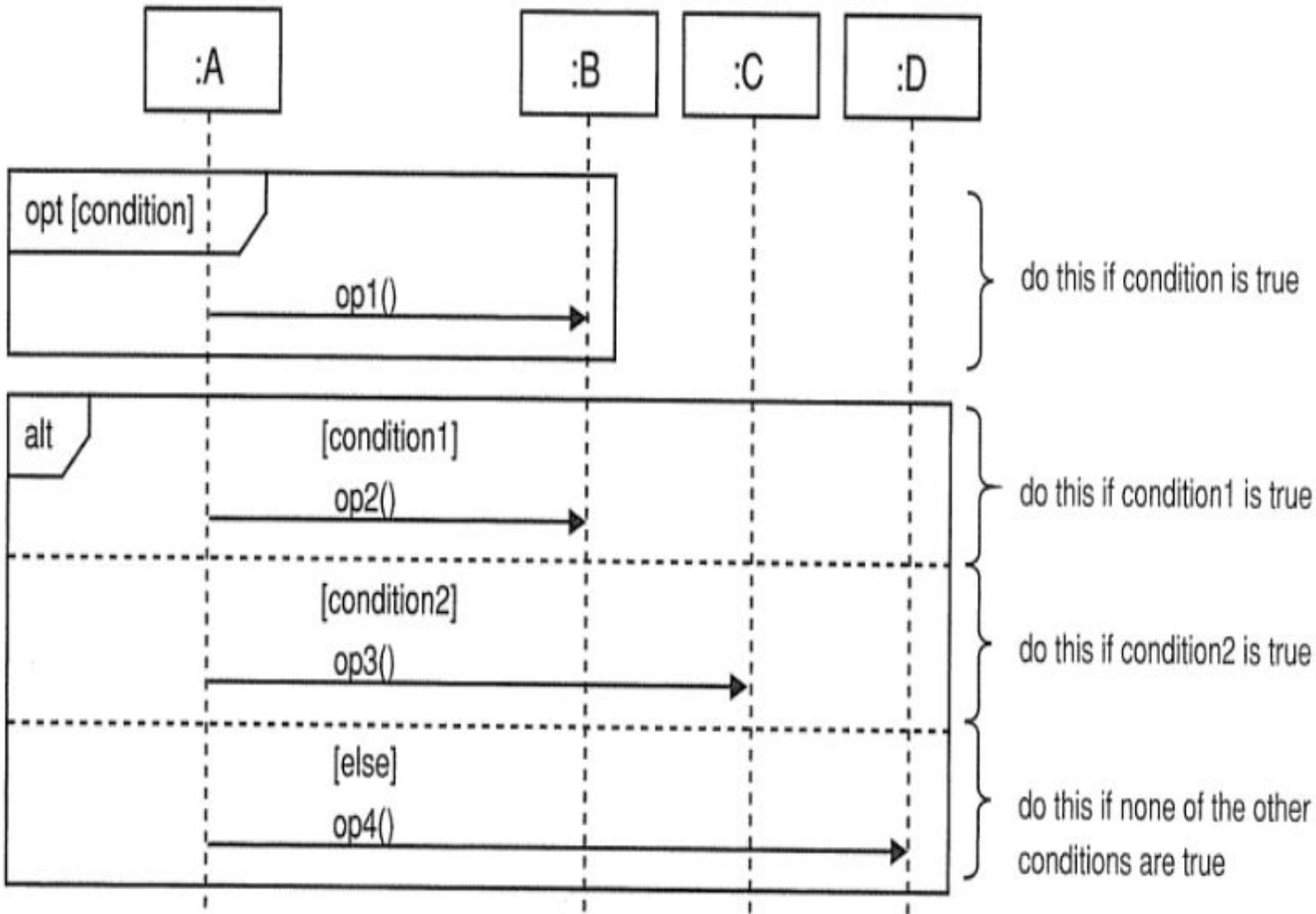
BASIC ELEMENTS “Switch-Case”



- **IF condition1 is TRUE**
 - *call1()* is called,
- **ELSE IF condition2 is TRUE**

Sequence diagram

“If-Then” & “Switch-Case”

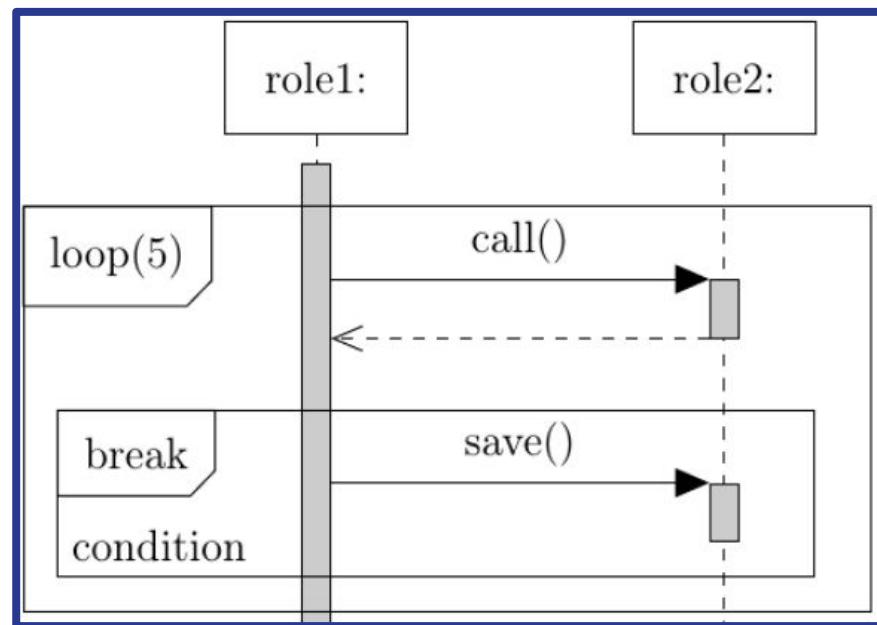
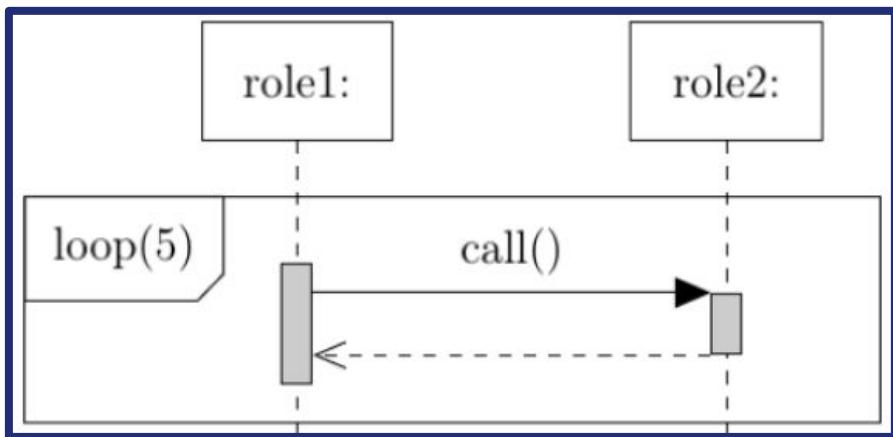


IF ...
Then.

Switch ... Case

Sequence diagram

FOR & WHILE/FOR

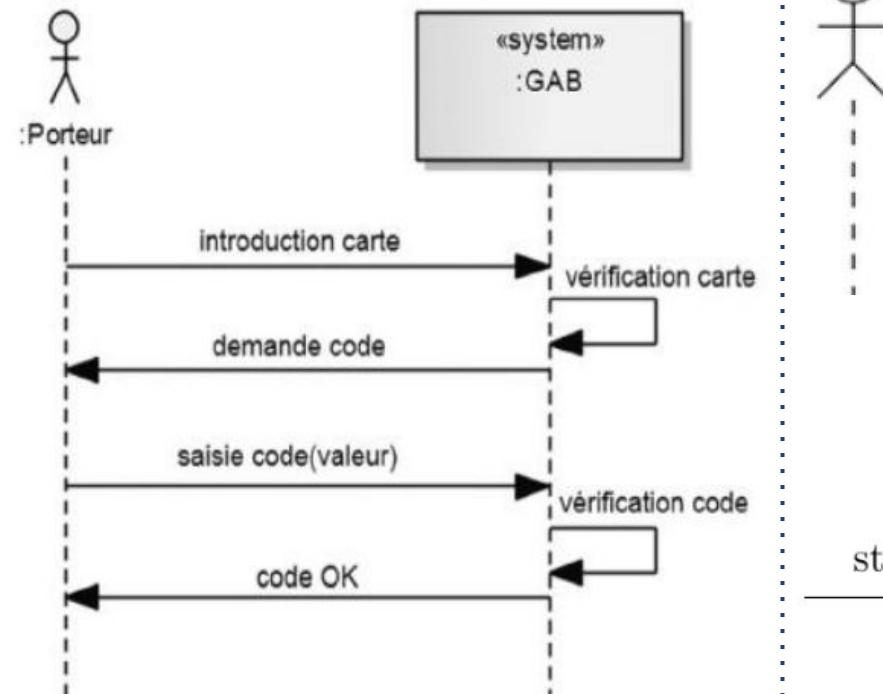


- **loop(n) case:**
 - We give the number of iterations.
 - We run all the instructions for each iteration.

- **loop(n) + break case:**
 - Stop the loop IF condition is TRUE.
- **loop + break case:**
 - typical while loop

Sequence diagram

EXAMPLE

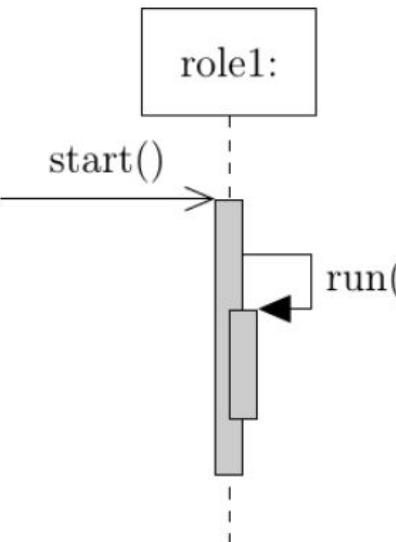


SPECIAL ELEMENTS



Actor **outside the system** (not integrated into the class diagram).

For example, it can represent the user of the software or the architecture you are building.



Several executions can overlap on the same lifeline.

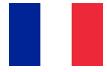
(e.g., a role1 makes a change on its own state)

Sequence diagram

- EXERCISE SERVEUR&CLIENT -



- Model a sequence diagram such as:
 - A **Server** instance creates threads (**Thread** class),
 - A **User** named *geek* wakes up the server via a call to a *wakeOnLan* method. Once the server responds, *geek* executes the server's *Get* method with the name of a file as parameter.
 - *Threads* also have a *Get* method.
 - *driveZ*, an **HDD** instance, contains the data which is read with *read(File)* method called by the *thread*. And the data is then returned.
- Draw the sequence diagram by correctly ordering the classes according to the series of method calls.



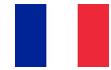
- Modéliser un diagramme de séquences tel que :
 - Une instance de **Serveur** crée des *threads* (classe *Thread*),
 - Un **User** nommé *geek* réveille le serveur via un appel à une méthode *wakeOnLan*. Une fois que le serveur lui répond, *geek* exécute la méthode *Get* du serveur paramétrée du nom d'un fichier.
 - Les *threads* ont également une méthode *Get*.
 - *driveZ*, instance de **HDD**, contient les données qui sont lues grâce à leur méthode *read(File)* appelée par le *thread*. Et les données (*data*) sont alors renvoyées.
- Tracer le diagramme de séquence en ordonnant correctement les classes selon la séquence d'appels aux méthodes.



You need to develop a management software for e-commerce. You simply start with the basket management with:

- 1 class diagram with **attributes** and **methods** (to be deduced) respecting the encapsulation of the data. Deduce the **classes** (written in bold) and their content from the points below:
 - An **Item** corresponds to an item in the shopping cart and is defined by a product (*Product*) and its quantity (*quantity*). We add an item to the basket by the *addItem()* method.
 - **ShoppingBasket** is a class that contains the list of the *Items* of the buyer (*Buyer*) Items. A buyer is identifiable by an integer *idBuyer*.
 - **Product**, which lives up to its name, has:
 - A *productId*, a *price*,
 - A *name* and a *description*.

- EXERCICE BASKETMANAGEMENT 1/2 -



Vous avez besoin de développer un logiciel de gestion pour du e-commerce. Vous commencez simplement par la gestion de panier avec :

- 1 diagramme de classe avec **attributs** et **méthodes** (à déduire) respectant l'encapsulation des données. Déduire les classes (écrites en gras) et leur contenu des points ci-dessous :
 - Un **Item** correspond à un élément du panier d'achat et est défini par un produit (*Product*) et sa quantité (*quantity*). On ajoute un *item* au panier par la méthode *addItem()*.
 - **ShoppingBasket** est une classe qui contient la liste des *Items* de l'acheteur (*Buyer*) et un acheteur identifiable par un *idBuyer* entier.
 - **Product**, qui porte bien son nom, a :
 - Un *productId*, un *price*,
 - Un *name* et une *description*.



The class diagram is accompanied by a sequence diagram involving:

- 1 Customer (the actor) who uses the software,
- An instance of the *ShoppingBasket* and *Item* classes.

This sequence diagram represents a process of basketing an Item by the Customer (use an *addItem()* method which will create the *Item* object) then the following management:

- If the user changes the quantity of the Item (*ChangeQuantity*), the appropriate method is called.
 - If this quantity becomes zero, the Item is destroyed.
- If the user chooses to delete the Item (*deleteItem*), it is deleted.

- EXERCICE BASKETMANAGEMENT 2/2 -



Le diagramme de classe est accompagné d'un diagramme de séquences mettant en jeu :

- 1 acteur qui utilise le logiciel, i.e., un *Custumer*,
- Une instance des classes *ShoppingBasket* et *Item*.

Ce diagramme de séquence représente un processus de mise en panier d'un *Item* par l'acteur (utilisez une méthode *addItem()* qui va créer l'objet *Item*) puis la gestion suivante :

- Si l'utilisateur change la quantité de l'*Item* (*ChangeQuantity*), la méthode adéquate est appelée.
 - Si cette quantité devient nulle, l'*Item* est supprimée.
- Si l'utilisateur choisi de supprimer l'*Item* (*deleteItem*), il est supprimé.

“Break n°3” related to the project

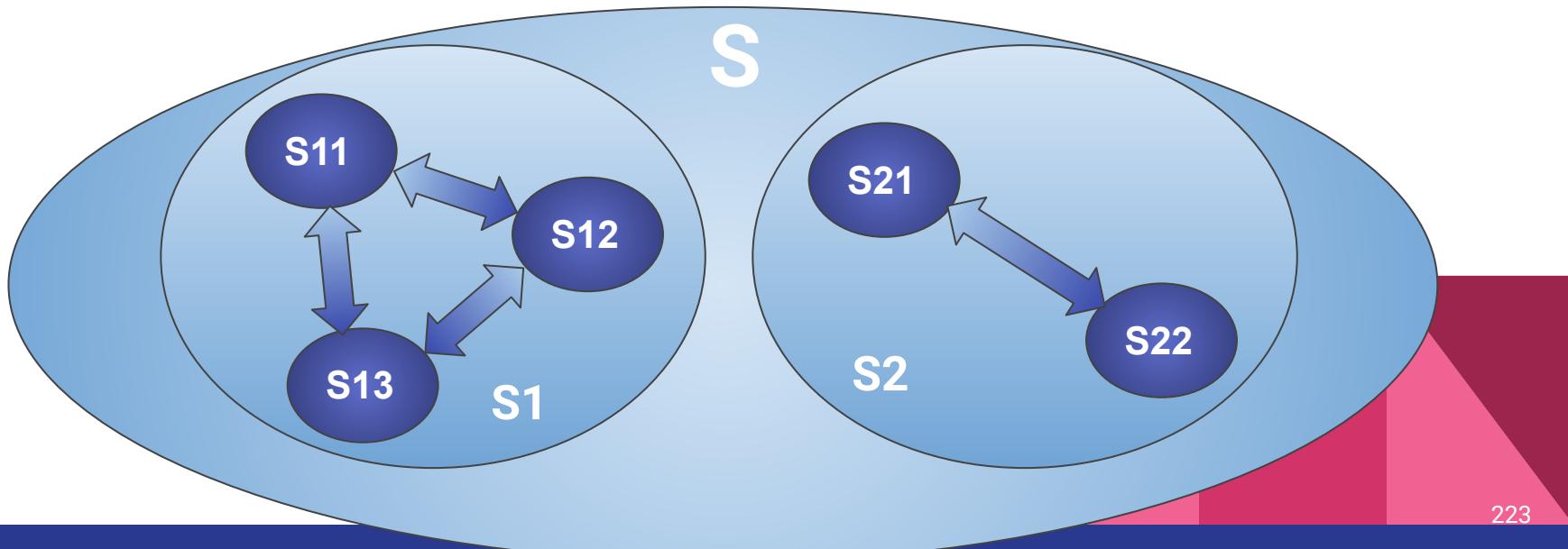
Systems & Models

Multi-Agent System

Discrete Simulation

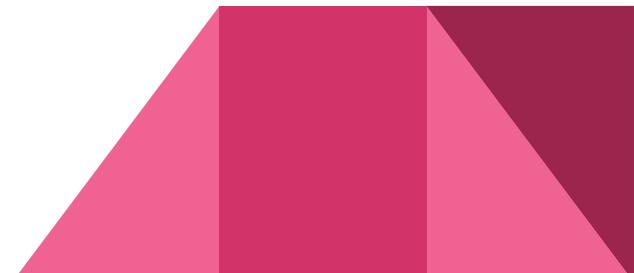
A System... in 1 slide...

- (Very) simple definition of system:
 - **A collection of interacting objects.**
- There are **static** and **dynamic** systems. These mean that **interactions can change the state of objects.**
- There is a whole **encapsulation of systems**. For example, an open subsystem will evolve according to the environment in which it is located, this environment is also a system.



A Model... in 1 slide...

- (Very) simple definition of a model:
 - **Abstraction simplifying the real system studied.**
- An **old notion** like... the heliocentric model of Copernicus!
- Minsky, 1965:
 - “To an observer B, an object A* is a model of an object A to the extent that B can use A* to answer questions that interest him about A.”
- Popper, 1973, 3 characteristics common to all models:
 - **A model must have a character of resemblance to the real system,**
 - **A model must constitute a simplification of the real system,**
 - **A model is an idealization of the real system.**

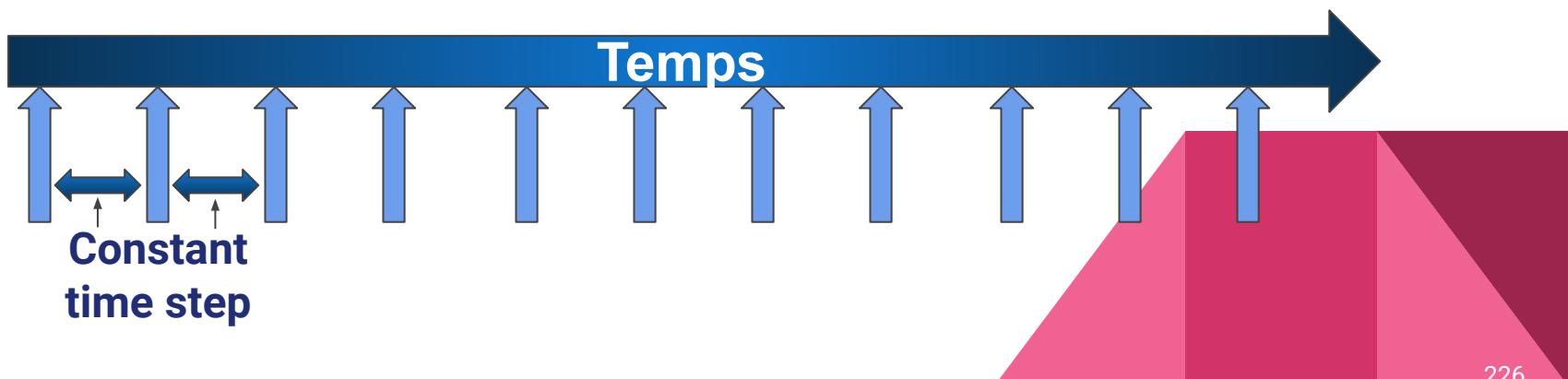


Discrete Simulation... in 1 slide...

- We immerse a **model in the time dimension...**
 - / On plonge un modèle dans le temps...
- Simulation definition (Hill, 1993) :
 - “**Simulation** is the process of changing an **abstraction** of a **system over time to help understand how that system works and behaves**, and to understand some of its dynamic characteristics in order to evaluate different decisions.”
- **Discrete Simulation:**
 - Set of possible values of variables of **finite size**
 - => Set of states of the finite size system;

Time management in Simulations (1/2)

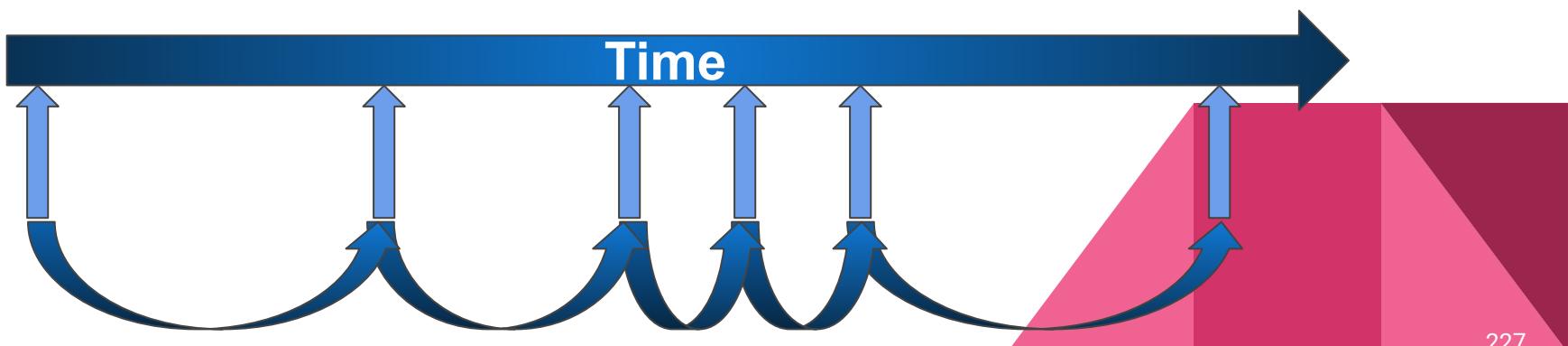
- **Clock driven / constant time step simulation**
 - / Simulation dirigée par horloge / à pas de temps constant
- Technically simple,
- Duration of the simulation discretized into a number of **intervals of identical size** (i.e., the time step),
 - / Durée de la simulation discrétisée en un certain nombre d'intervalles de taille identique (i.e., le pas de temps),
- **At each time step: search for events that must be triggered in this period,**
- **Main difficulty: finding the adequate time step.**



Time management in Simulations (2/2)

- Event driven / timeline simulation

- / Simulation dirigée par les évènements / à échéancier
- Technically **complicated**,
- Virtual time progresses **from one date of occurrence of events to another**,
 - Le temps virtuel progresse d'une date d'occurrence d'évènements à une autre,
- **No event search to process**,
- **Skipping unnecessary periods of time**,
- Main difficulty: **managing a schedule storing events ordered chronologically**.



Multi-Agent Systems (MAS) ... in 1 slide ...

/ Systèmes Multi-Agents (SMA)...en 1 slide...

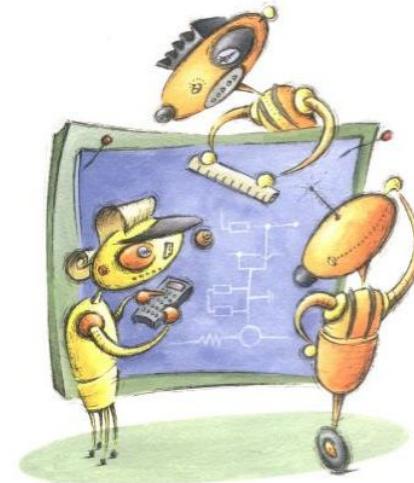
- 2 types of **multi-agent systems (MAS)**:

- **Reactive MAS:**

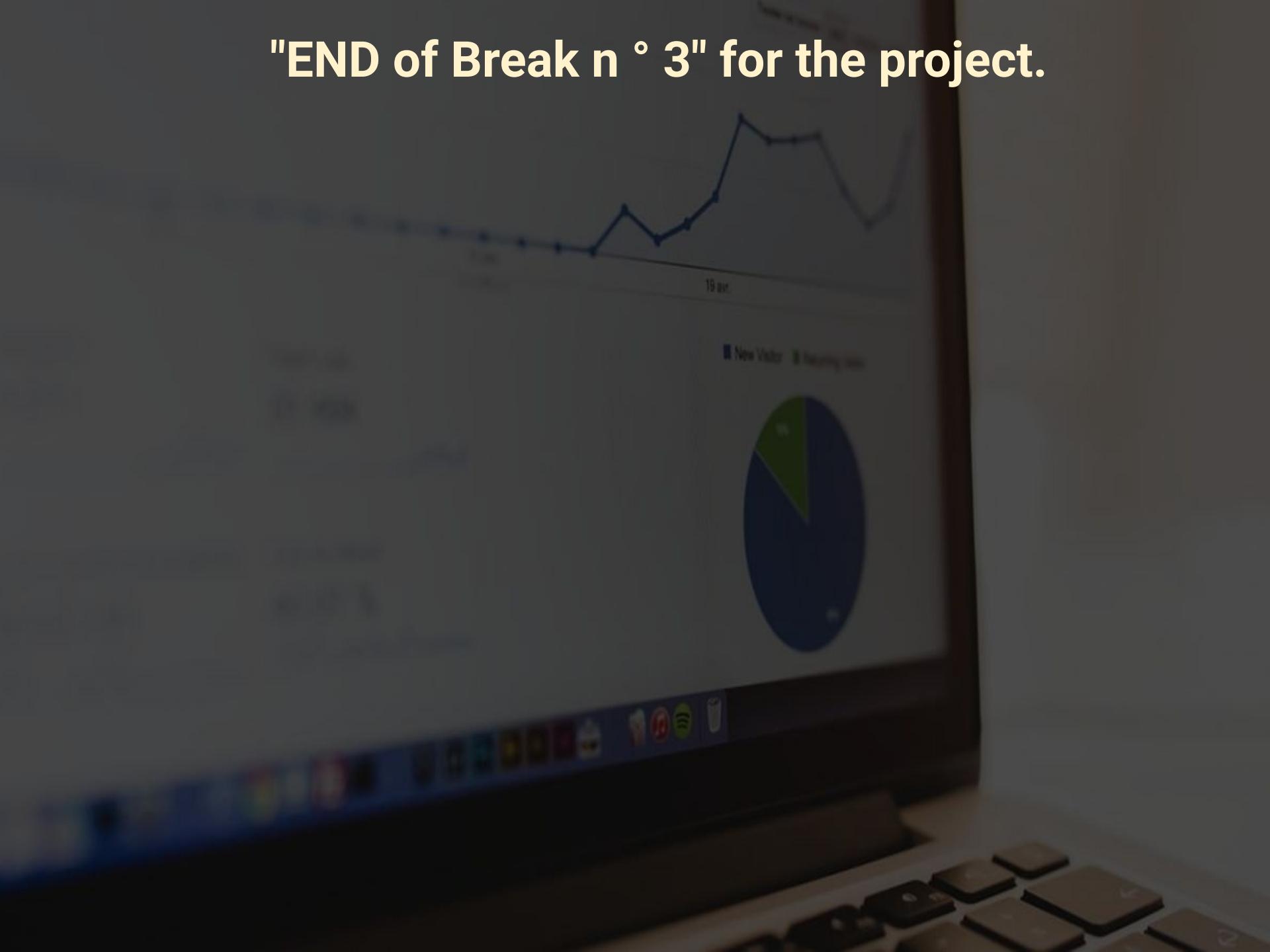
- Large number of agents,
 - Without memory,
 - Local vision of their environment;

- **Cognitive MAS:**

- Few agents,
 - With memory,
 - Good knowledge of the environment,
 - Based on communication protocols between agents,
 - Knowledge of agents among themselves;



"END of Break n ° 3" for the project.



3.

Unified Modeling Language (UML)

3.7. Activity Diagram *(/ diagramme d'activité)*

Activity Diagram

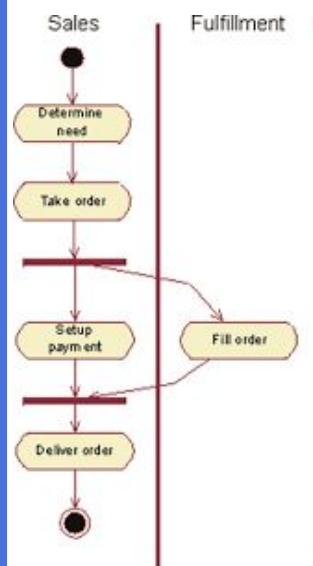
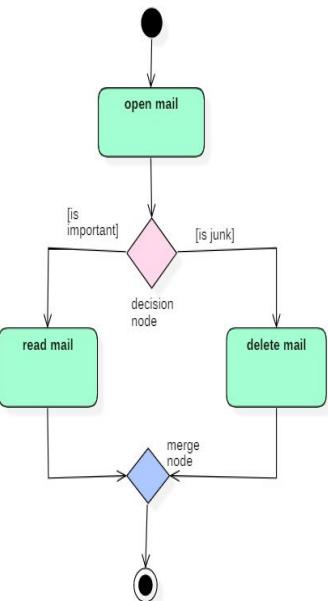
- Goal:**

- Represent the **sequential or parallel actions** within one of the processes of the system studied.
- / Représenter l'enchaînement séquentiel ou parallèle d'actions au sein d'un des processus du système étudié.

- Structure:**

- Use **objects** and **control flows**,
- Similar to an algorithmic scheme,
- Can partition actions according to object type like columns in sequence diagrams.

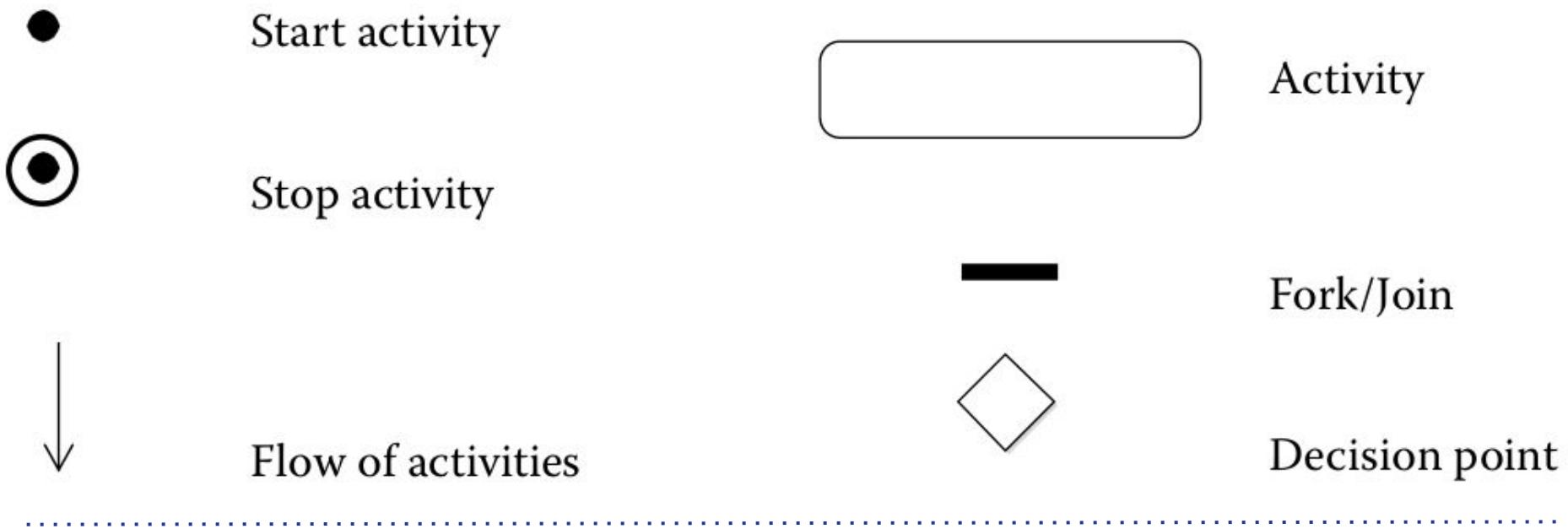
- When?** From the start of a project (informal "high level" vision) and complete until its end ("low level" vision close to algorithms).



Activity Diagram

Notations

(eng. / fr.)



Respectivement en français :

- Départ d'activité
- Fin d'activité
- Flot
- Activité / Action
- Débranchement
- Décision

Activity Diagram

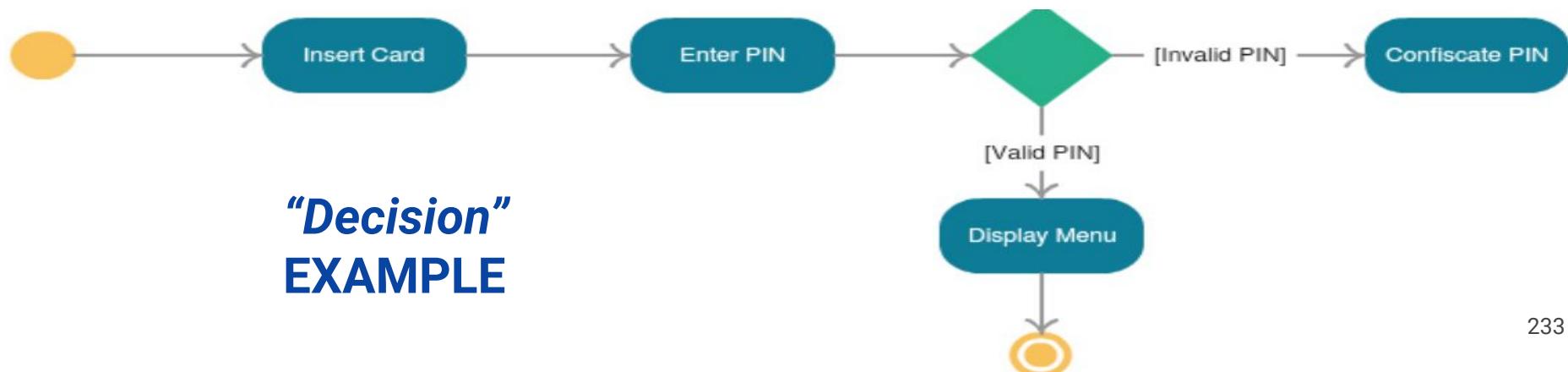
BASIC ELEMENTS

Activity/Action : represents treatment or transformation. An activity can contain several actions, the latter are the basic unit of these diagrams.

Flow : “sequencing control” arrow connecting the different elements of the activity diagram.

2 types for a single “diamond” (“decision and merge” nodes):

- **decision** : control node with 1 incoming flow and/or several outgoing [under conditions],
- **merge** : Control node with several inbound and 1 outbound.



**“Decision”
EXAMPLE**

Activity Diagram

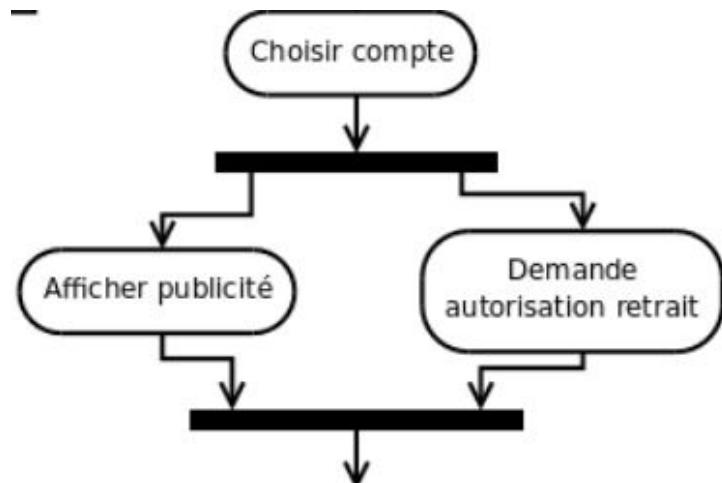
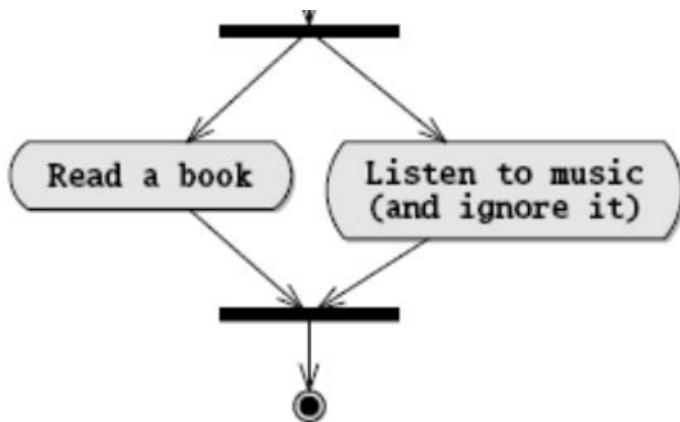
BASIC ELEMENTS

Fork (/ Débranchement) & **Join** (/ Rebranchement).

- Allows the start of **concurrent actions** (in parallel) (i.e., bifurcation node) then their **synchronization** (i.e., union node).

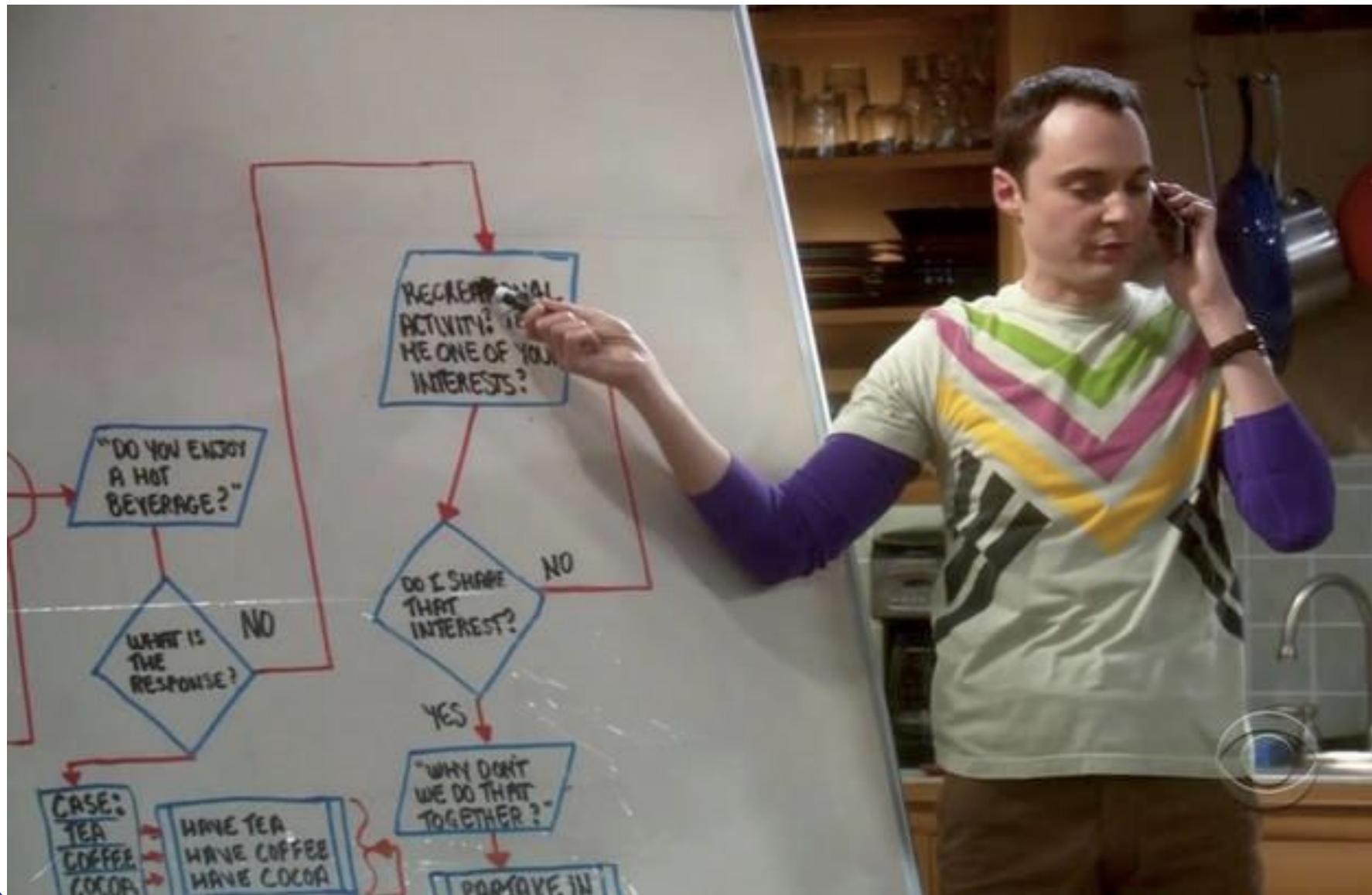
Fork has an incoming flow leading to several outgoing flows; the reverse is true for **Join**.

EXAMPLE



Activity Diagram

- Easy EXERCISE diagOfSheldon -



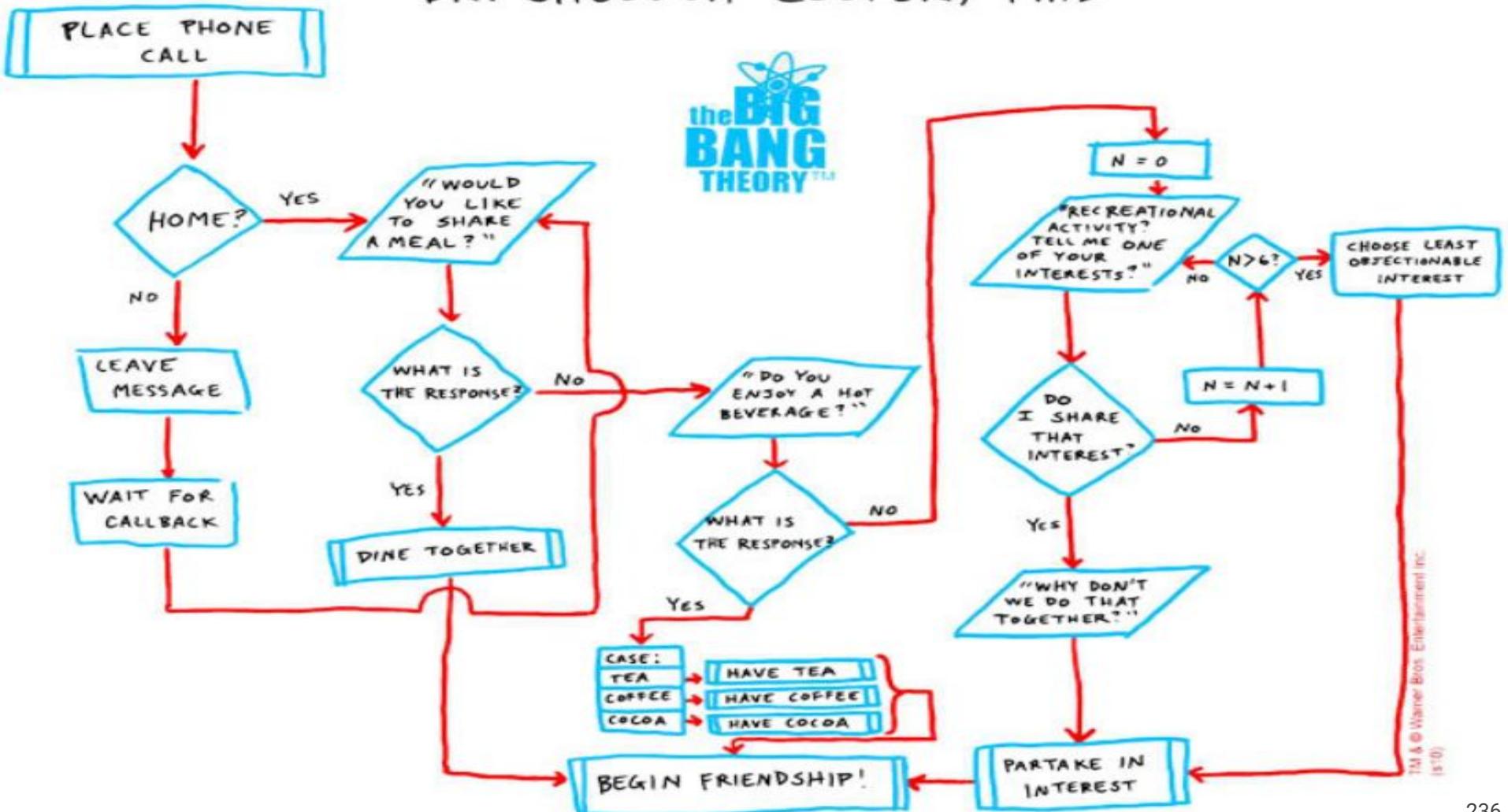
Activity Diagram

- EXERCISE diagDeSheldon - (S02E13)

Transform the Sheldon diagram into a UML activity diagram

THE FRIENDSHIP ALGORITHM

DR. SHELDON COOPER, Ph.D



Activity Diagram

Advanced Elements



Rep. Stop Flow and Stop Activity. The 1st (more rare) only terminates one stream (and does not terminate the other streams in parallel). The 2nd (more frequent) ends all activities.

Calling an Activity


“Super” Action calling another activity (here given in place of “Calling an Activity”), the small fork  being the distinctive sign. Once the call for the new activity is over, the flow starts again from the Super Action.

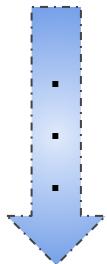
End of Month
Occurred


Hourglass (/sablier) blocking a flow for a certain time announced (here “End of Month Occurred”). Once the time is up, the flow starts again. An activity can also start on this element.

Activity Diagram

Advanced Elements

Sending Signals



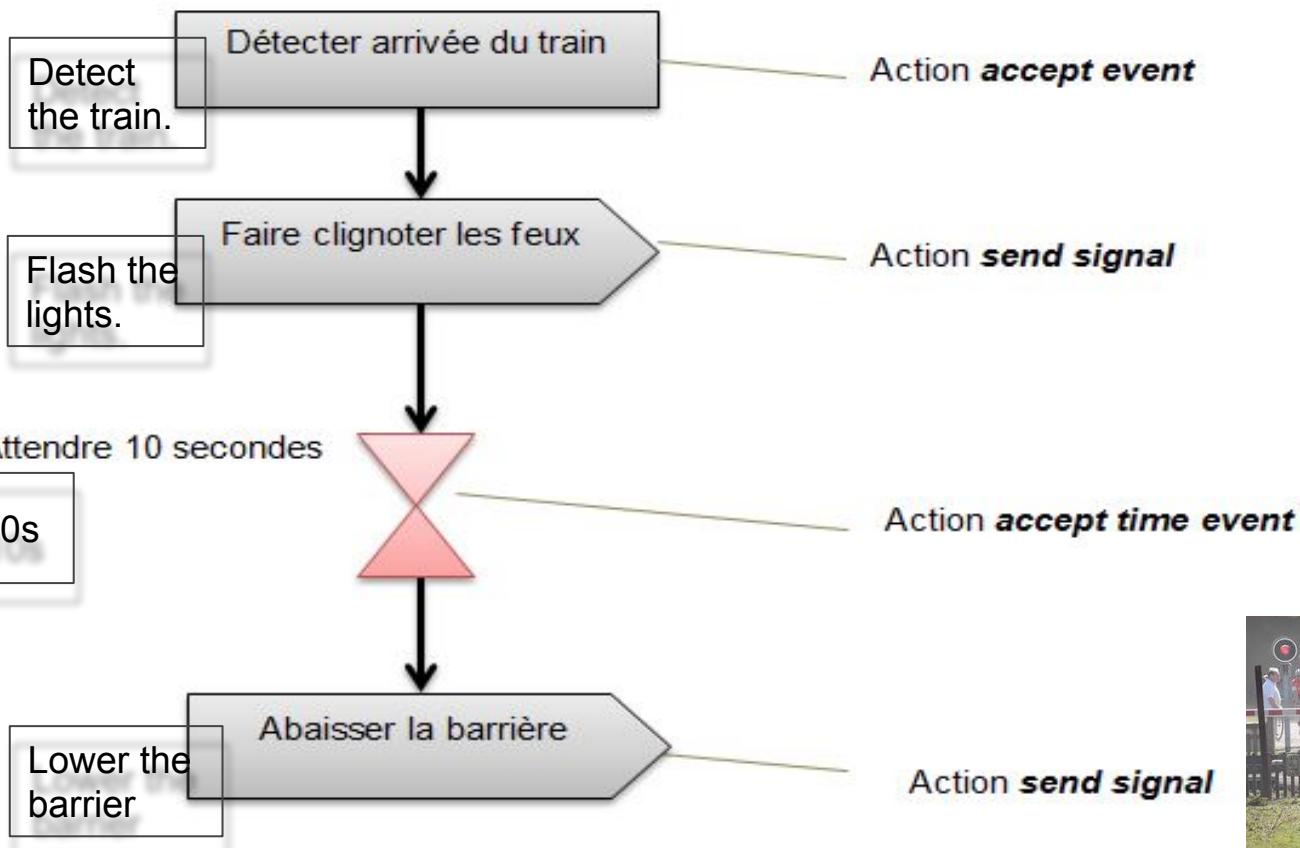
Accepting
an Event

Sends a signal to activate a flow on an “Accepting an Event” type action.

Action waiting to be executed. Once the event is “accepted”, a flood starts from this action.

Activity Diagram

SPECIAL ACTIONS - EXAMPLE “Railroad Crossing”

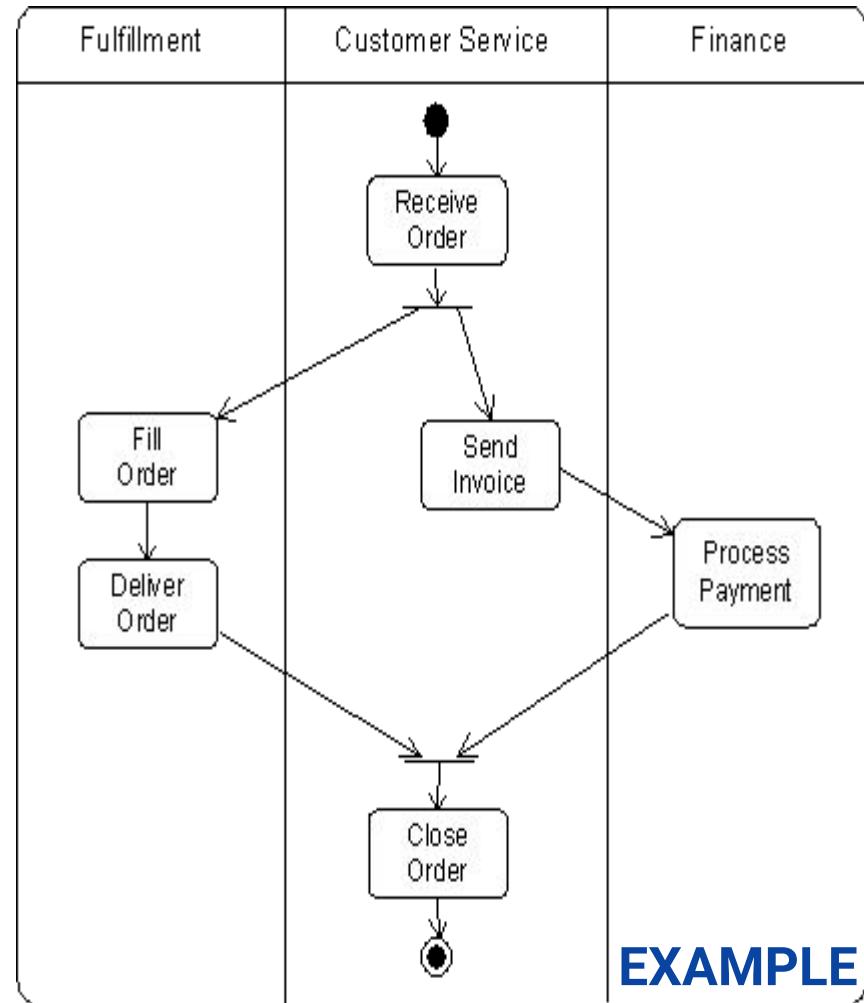


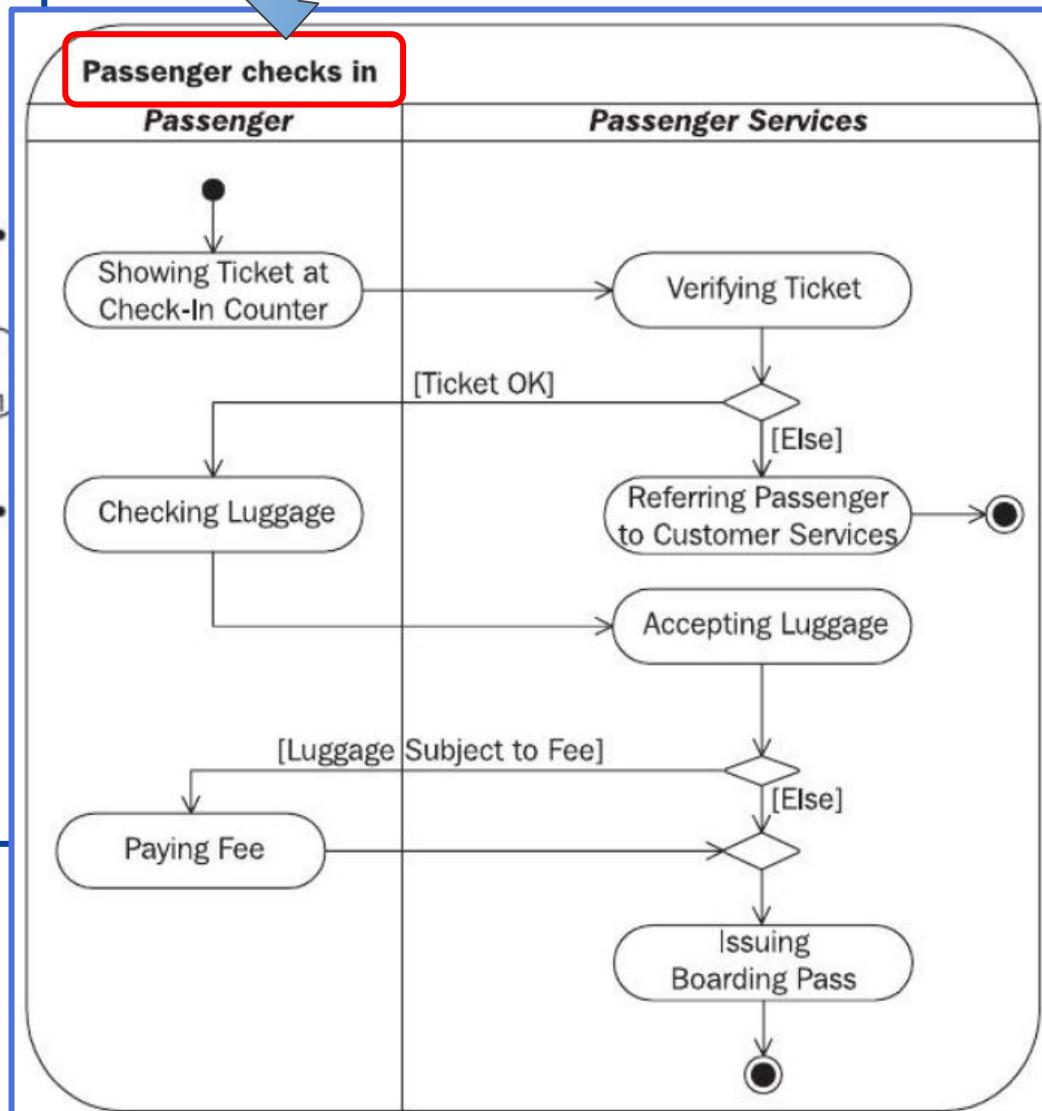
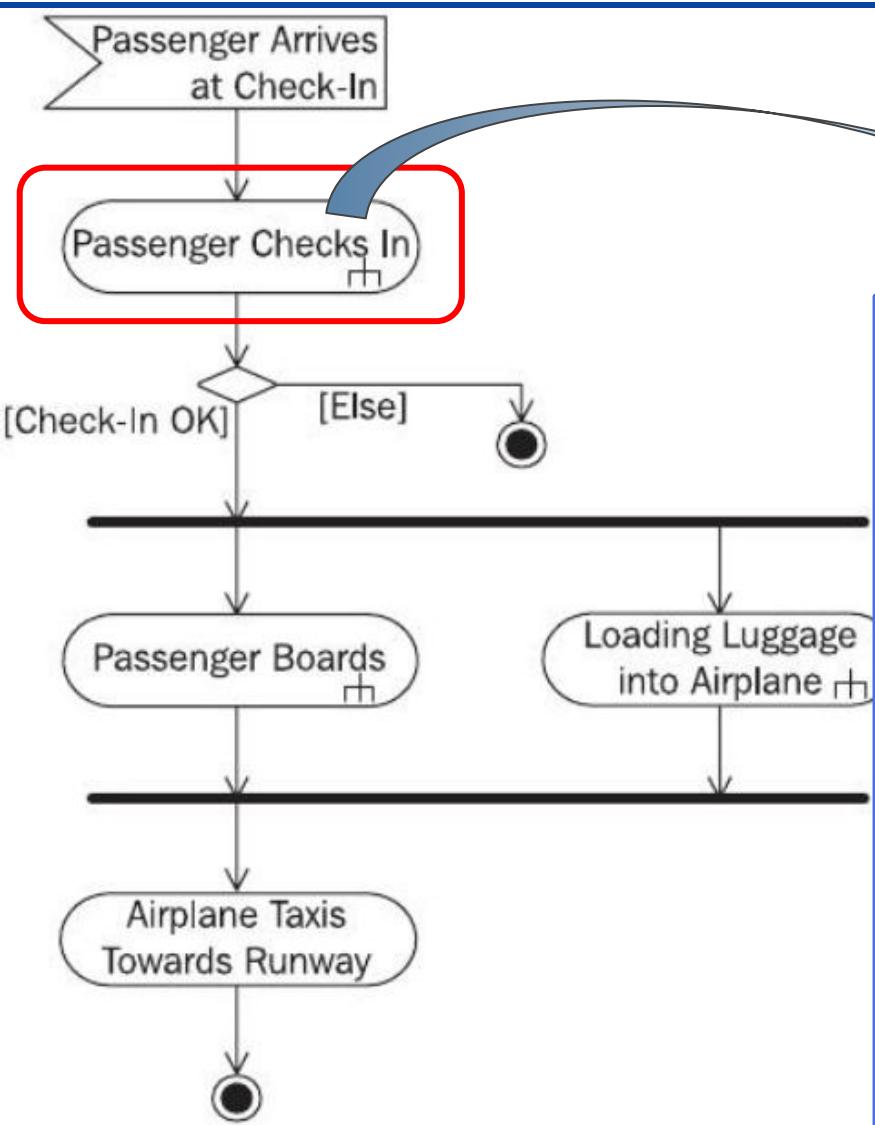
Activity Diagram

PARTITION

All the actions of an activity diagram can be partitioned into parts representing classes responsible for those actions. For example, we can find classes.

This partition is often vertical but can be horizontal.





Example

“Airport Arrivals”



Activity Diagram

- Prepare dinner -



Short exercise on Fork & Join

Tonight, you have to make dinner...

In your TO DO LIST you have to:

- (1) - Cook spaghetti,
- (2) - Mix Carbonara Sauce,
- (3) - Combine 1 & 2,
- (4) - Open Red Wine during (1), (2) and (3) if one is in mood for it.

Activity Diagram (think about which actions can be made in the same time)?



Activity Diagram

EXERCISE Multithreaded Chocolate Mousse



In order to prepare for 100% autonomous kitchen machines, a chef wants to put his recipes in the form of an activity diagram. He · She tries chocolate mousse first.

He · She knows he · she can chop chocolate before it melts, and, at the same time, that he · she can crack the eggs and then separate the whites from the yolks. This then allows him to add the egg yolks to the chocolate regardless of whipping the whites to stiff peaks.

Finally, he · she can mix the egg whites with the chocolate preparation before pouring everything into ramekins. These end up in the refrigerator. The recipe is complete (and can be considered successful) after 3 hours.

Note: A difficulty of this exercise is to separate sequential actions from parallel actions.





Activity Diagram

- EXERCICE mousseChocoMultithreadée -



Afin de se préparer aux robots de cuisine 100% autonomes, un·e chef veut passer ses recettes sous la forme d'un diagramme d'activité. Il·Elle s'essaye d'abord à la mousse au chocolat.

Il·Elle sait qu'il·elle peut casser le chocolat avant de le faire fondre, et, en parallèle, qu'il·elle peut casser les oeufs puis séparer les blancs des jaunes. Cela lui permet ensuite d'ajouter les jaunes d'oeuf au chocolat indépendamment de battre les blancs en neige.

Enfin, il·elle peut mélanger les blancs en neige avec la préparation chocolat avant de verser le tout dans des ramequins. Ces derniers finissent au réfrigérateur. La recette est terminée (et peut être considérée comme réussie) au bout de 3 heures.

Note : Une difficulté de cette exercice est de bien séparer des actions séquentielles des actions parallèles.



Activity Diagram

- EXERCISE Multithreaded Chocolate Mousse (1 year later) -

After multiple unsuccessful attempts, the chef renames his 100% autonomous robot as “Assistant”. Indeed, the robot is not skillful enough neither to break the eggs, nor to separate the whites from the yolks, nor to beat the whites to snow (a shame!). And, finally, nor to mix the whites and the egg whites. chocolate preparation.

Thus, he wants us to modify his activity diagram using a score in order to clearly mark what the assistant is doing and what he is obliged to do.



Activity Diagram

- EXERCICE mousseChocoMultithreadée1AnAprès -

Après de multiples tentatives infructueuses, le chef renomme son robot 100% autonome en “Assistant”. En effet, le robot n'est pas assez habile ni pour casser les oeufs, ni pour séparer les blancs des jaunes, ni pour battre les blancs en neige (un comble !) et, enfin, ni pour faire le mélange des blancs et de la préparation chocolat.

Ainsi, il veut que l'on modifie son diagramme d'activités en utilisant une partition afin de bien marquer ce que fait l'assistant et ce que lui est obligé de faire.



3.

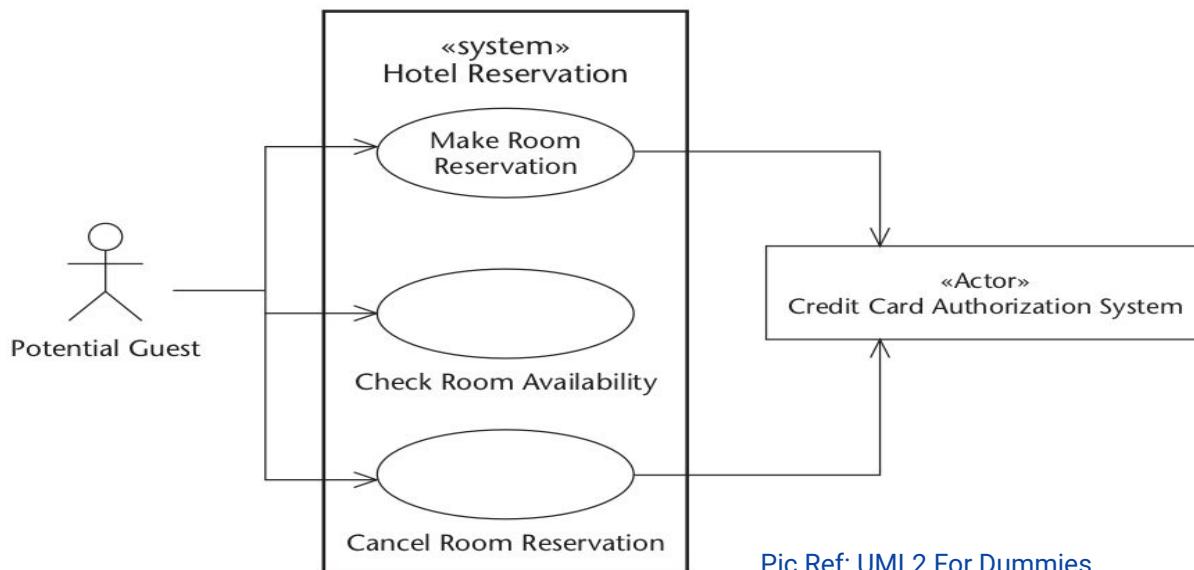
Unified Modeling Language (UML)

3.8. Use Case (*fr.* Cas d'utilisation)

USE CASE DIAGRAMS

- **Determine the users**
- **Defining use scenarios**
- **To describe needs from an external point of view**
 - between customers and developers
- **To keep you focused on your users' goals**
- Help for accomplishing a task and/or communicate with your peers

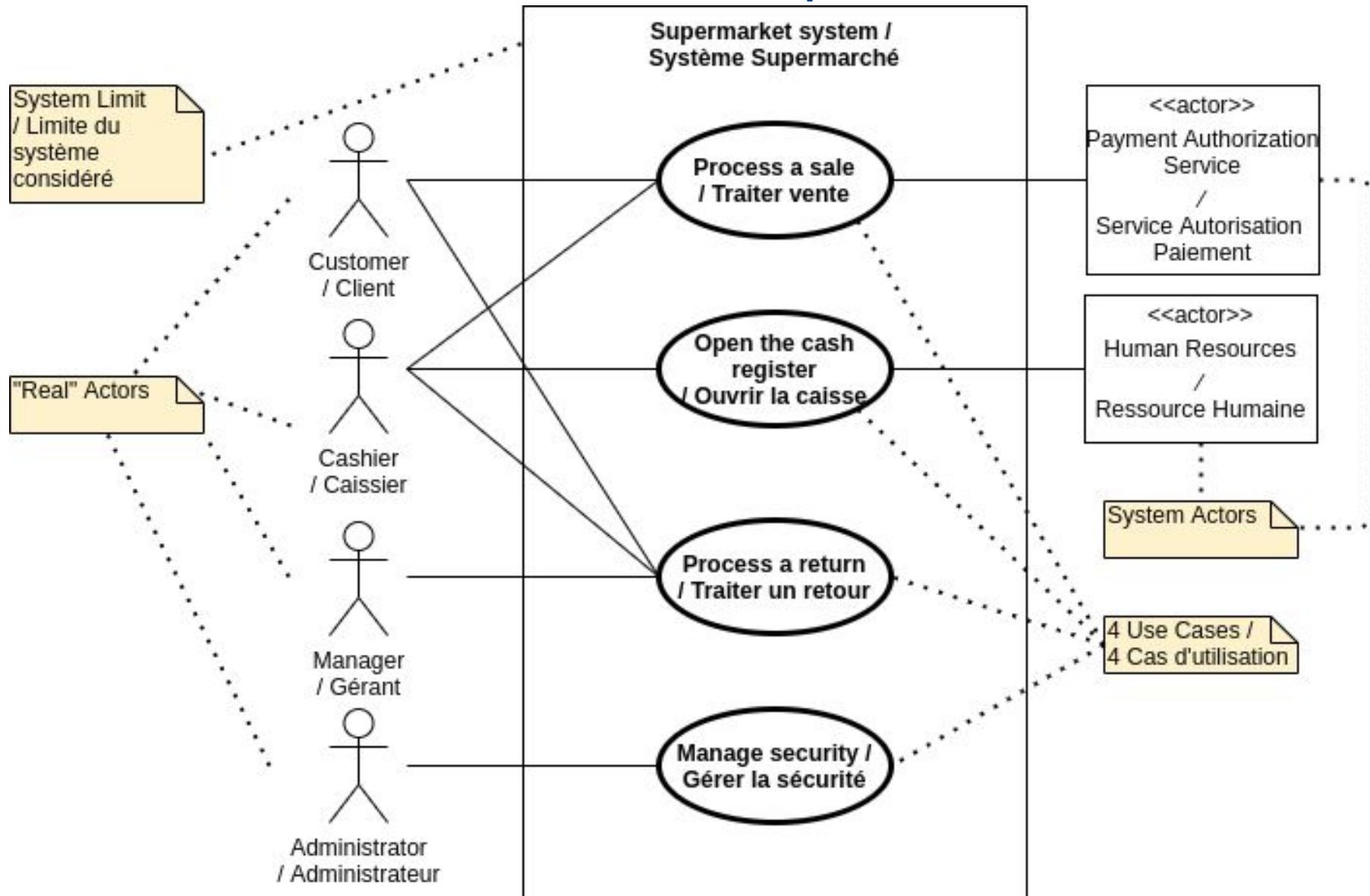
Example >



USE CASE DIAGRAMS

(fr. Cas d'utilisation)

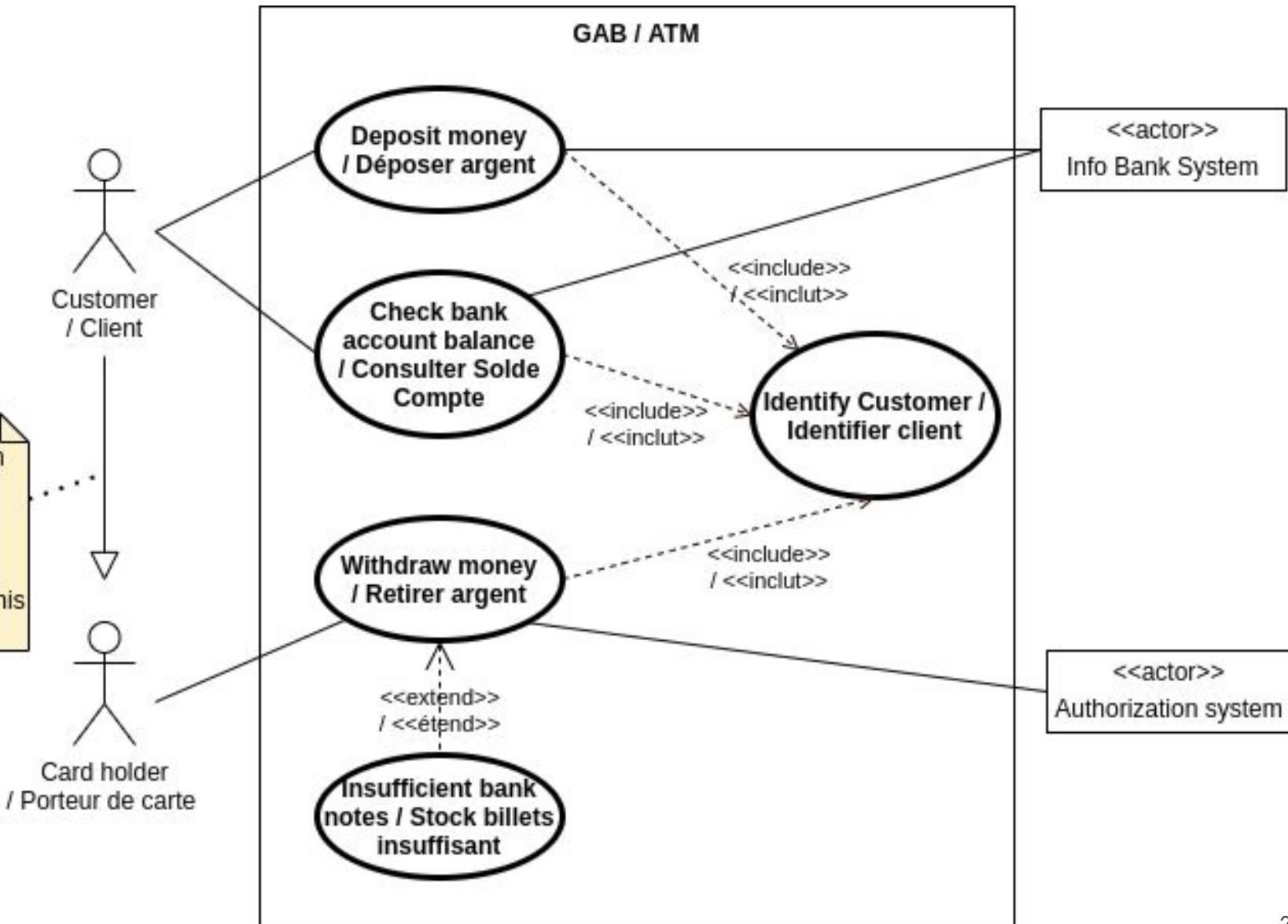
Example



USE CASE DIAGRAMS

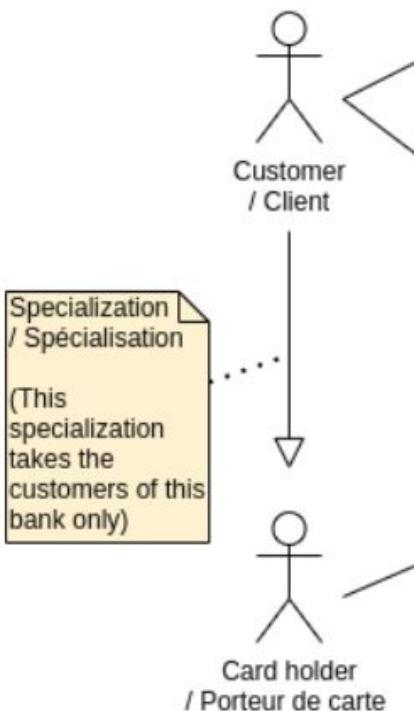
(fr. Cas d'utilisation)

Example



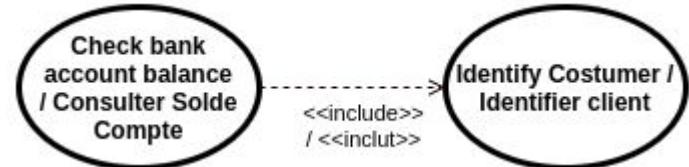
Specialization / Spécialisation
(This specialization takes the customers of this bank only)

Specialization / Spécialisation
Generalization / Généralisation

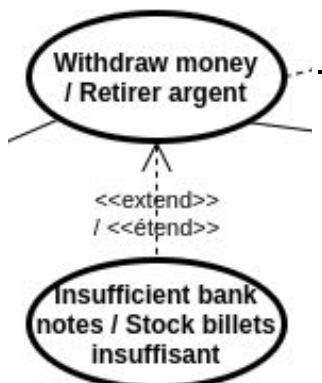


Example:

- only clients of the bank can deposit and check the bank account balance.
- Card Holder is a **generalization** of a Customer while Customer is a **specification** of a Card holder.

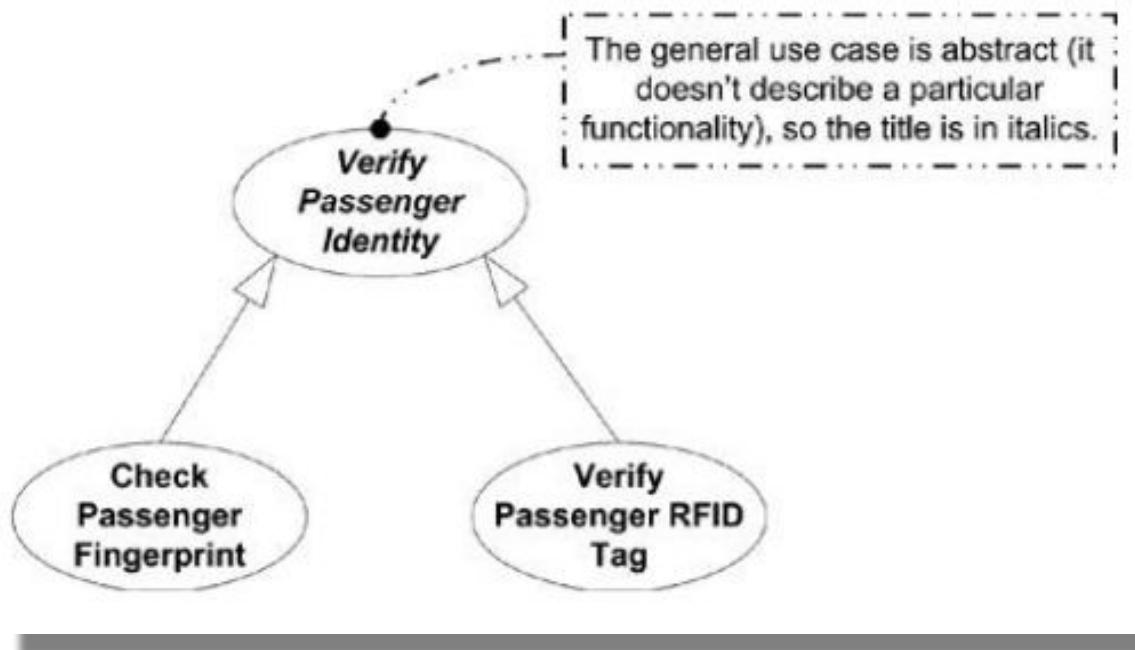


- Inclusion (**<<include>>**)
 - Precise the inclusion of a specific action in an use case.



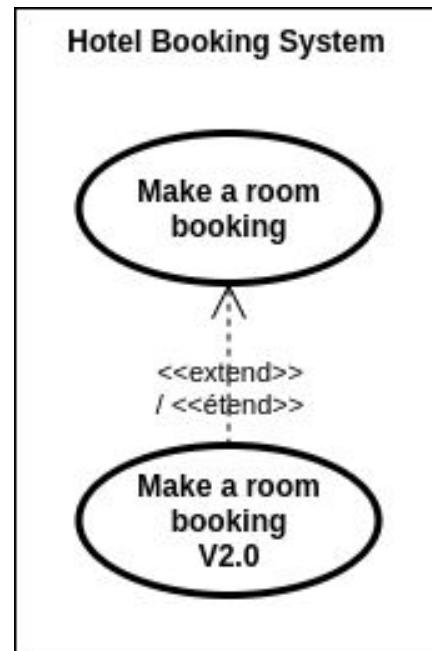
- Extension (**<<extend>>**)
 - Add actions, checks or upgrade in some use cases.

Generalization / Généralisation (Specialization / Spécialisation)



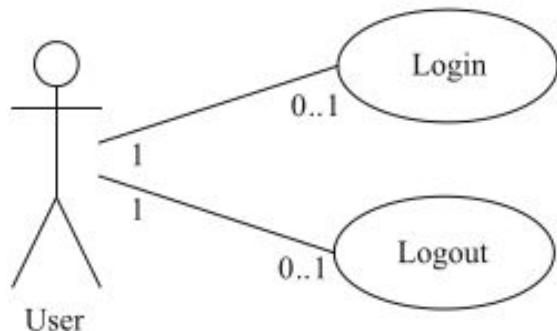
New release example (in order to display an update)

Extending / Extension
(<<extend>> / <<étend>>)

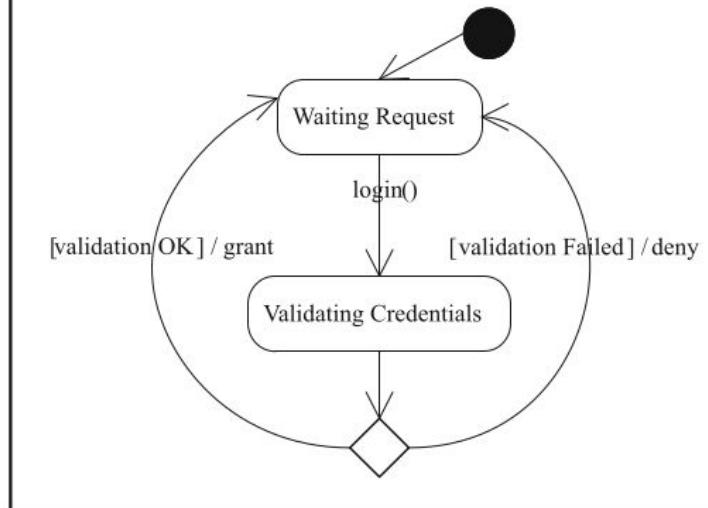


Few Remarks

uc login-logout

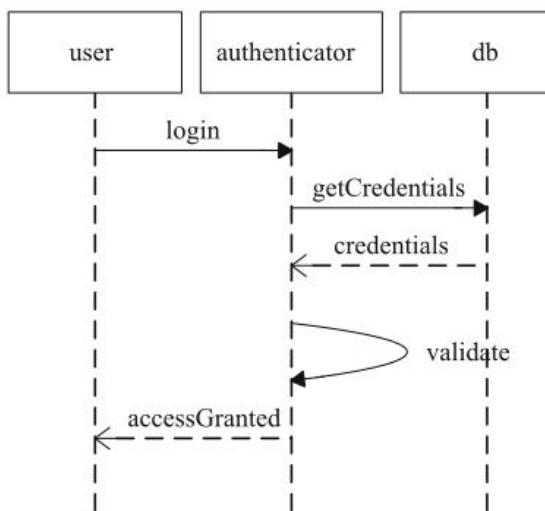


stm authenticator

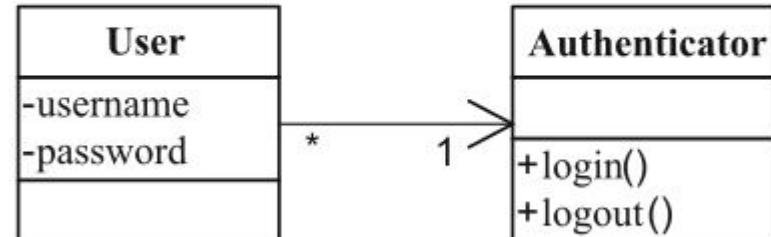


UML Generalities Frames

sd authenticator



class user

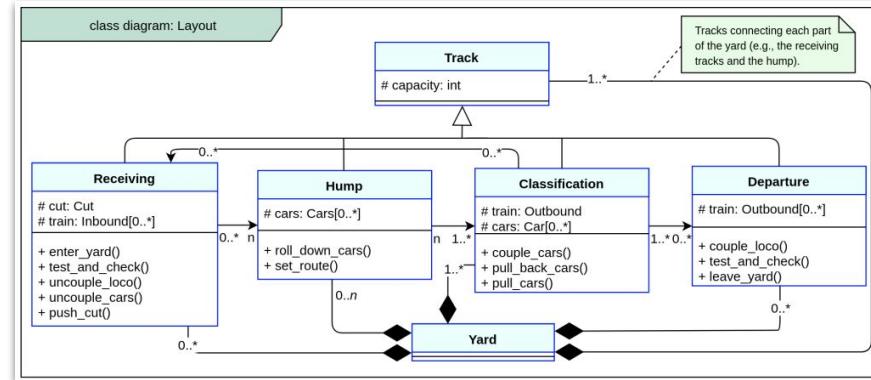


Before we are done with UML ...

Between code and UML diagrams ...

Generation of UML diagrams from code.
(reverse-engineering method)

```
81     self.mags.dt[i].color = colors
82     def on_key_press(self, symbol, modifiers):
83         "Delegate key press"
84         if self.context_index == -1:
85             if symbol == key.UP and not self.active_index == 0:
86                 self.menu_labels[self.active_index].color = [255, 255, 255, 255]
87                 self.active_index -= 1
88                 self.mags.dt = self.get_act_color_mag()
89             elif symbol == key.DOWN and not self.active_index == 3:
90                 self.menu_labels[self.active_index].color = [255, 255, 255, 255]
91                 self.active_index += 1
92                 self.mags.dt = self.get_act_color_mag()
93             elif symbol == key.ENTER:
94                 if self.active_index == 3:
95                     pyglet.app.exit()
96                 else:
97                     self.context_index = self.active_index
98             elif symbol == key.ESCAPE:
99                 if self.context_index == -1:
100                    pyglet.app.exit()
101                else:
102                    self.context_index = 1:
```



Generate code from UML diagrams.
(The one you will use for the project)

**GAME
OVER**

...UML

4. Design Pattern / Patron de conception

Design Patterns / Patrons de conception

- **Goal :**
 - Provide generic solutions for famous design problems
 - Do not reinvent the wheel!
 - Flexible and efficient code;
- **Structure :**
 - Objects and classes ;
- **Characteristics :**
 - Adaptability ; Validated by pairs ;
- **Difficulties :**
 - May increase code complexity.

To resume: - You have a problem with the design of software mechanism?
=> All of the design patterns probably have the answer...

Il était une fois...

Once upon a time...

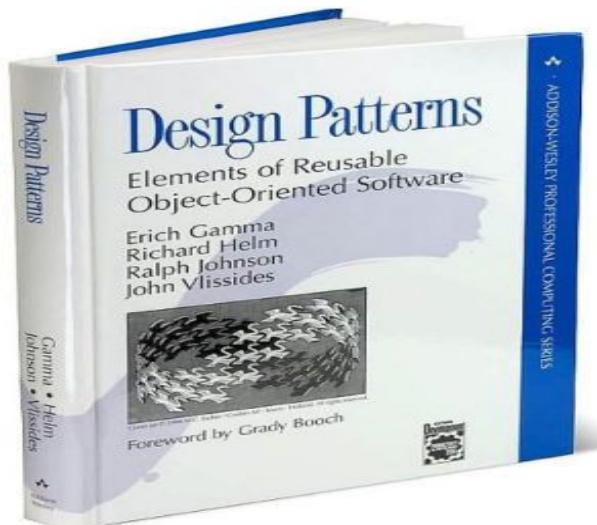
Les patrons de conception The design Patterns

(fr)

(eng)

SECRETS
d'HISTOIRE

- Appearing in the literature in **1994** :
- “Design Patterns: Elements of reusable **object oriented** software”,
- authors: **Gamma, Helm, Johnson et Vlissides**
 - Know as the **Gang of Four**.



Il était une fois...

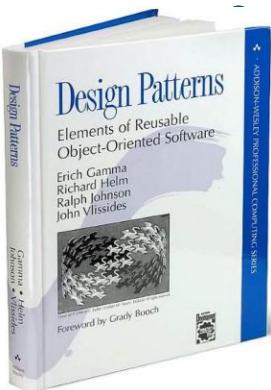
Once upon a time...

Les patrons de conception

SECRETS
d'HISTOIRE

The design Patterns

- The first **23** design patterns of the **gang of 4** are of three types:



Les patrons de création / **Creational Patterns**

- How to instantiate a class?
 - e.g., how to apply a single instance limit for a given class?
- How to configure an object?

Les patrons de structure / **Structural Patterns**

- How to interface?

○ Les patrons de comportement / **Behavioral Patterns**

- How to properly manage the behavior of an application?

(sorry... for the french... Catégorie de Patrons de Conception)

Les Patterns Créateurs

concernent l'instanciation des objets et fournissent tous un moyen de découpler un client des objets qu'il a besoin d'instancier.

Tout pattern qui est un **Pattern Comportemental** est en rapport avec les interactions des classes et des objets et la distribution des responsabilités.

Creational Patterns

Créateurs

Singleton
Prototype
Fabrication
Monteur
Fabrique abstraite

Structural Patterns

Décorateur
Composite
Poids-mouche
Proxy
Façade
Pont
Adaptateur

Comportementaux

Commande
Patron de méthode
Chaîne de responsabilités
Observateur
Interprète
Stratégie
Médiateur
Visiteur
État
Memento

Behaviour Patterns

Certains patterns (en gris) n'ont pas été abordés.
Vous en trouverez un aperçu dans l'annexe.

Les Patterns Structurels permettent de composer des classes ou des objets pour former des structures plus vastes.

Design Pattern **SINGLETON**

- **Goal:**
 - Limit the number of instantiation to only one object;
- **Particularities:**
 - It seems simple,
 - But (a bit) difficult to understand!
- **Examples:**
 - Limit the instantiation of one driver for each device,
 - 1 instance of a Buffer,
 - Etc.

Design Pattern Singleton

Some questions...

- How to instantiate?
- How to limit the access of the instantiation process?

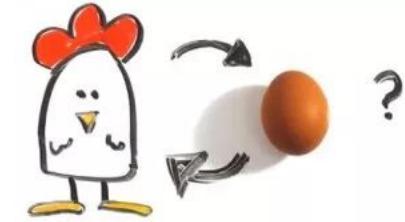
Design Pattern Singleton

```
1 public class SingletonALaClasse {  
2     private SingletonALaClasse() {};  
3  
4 }  
5  
6  
7  
8
```

Are you shocked by a private constructor?

How would you call this constructor?

Design Pattern Singleton



```
1 public class SingletonALaClasse {  
2     private SingletonALaClasse() {};  
3  
4 }  
5  
6  
7  
8
```

To call the **private** constructor for generating the first (and only) instance from a public method of the same class.

Why not...BUT..AT THE BEGINNING...:

How to call a method from a class which has no object/instance?

Design Pattern Singleton

```
2 public class SingletonALaClasse {  
3  
4     private SingletonALaClasse() {};  
5  
6     public static SingletonALaClasse getInstance()  
7     {  
8         return new SingletonALaClasse();  
9     }  
10 }
```

```
public class Main {  
    public static void main(String[] args) {  
        SingletonALaClasse.getInstance();  
    }  
}
```

By modifying a little the code above while continuing to use the properties of "static" elements, **how do you limit the number of instances to 1?**

Design Pattern

Singleton

```
2 public class SingletonALaClasse {  
3  
4     private static SingletonALaClasse lUnique;  
5  
6     private SingletonALaClasse() {}  
7  
8     public static SingletonALaClasse getInstance()  
9     {  
10         if (lUnique==null)  
11         {  
12             lUnique = new SingletonALaClasse();  
13         }  
14         return lUnique;  
15     }  
16 }
```

Singleton

- singleton : Singleton
- Singleton()
- + getInstance() : Singleton

```
public class Main {  
    public static void main(String[] args) {  
        SingletonALaClasse.getInstance();  
    }  
}
```

Design Pattern
Iterator

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Examples:

- browsing the main data structures (List, Vector, Tree etc.),
- browsing the contents of a directory in a file system,
- browsing the results of a query to a database.

We're looking for an object browsing into a collection of other objects. This object must be isolated from the intern structure of the collection.

Design Pattern Iterator

The Iterator class defines an interface for accessing the list's elements. An iterator object is responsible for keeping track of the current element;

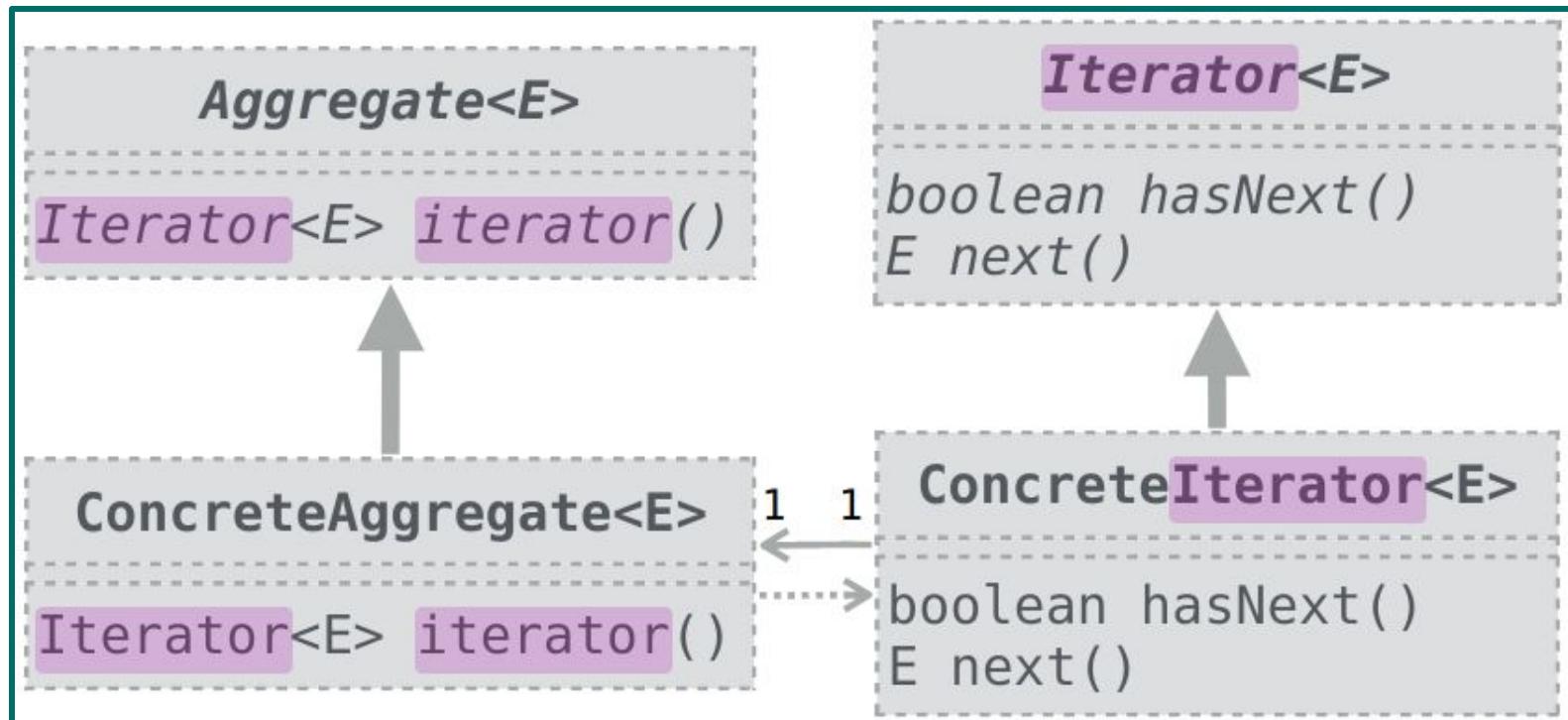
We can change the aggregate class without changing client code. We can do this by generalizing the iterator concept to support **polymorphic iteration**.

Design Pattern Iterator

To summarize:

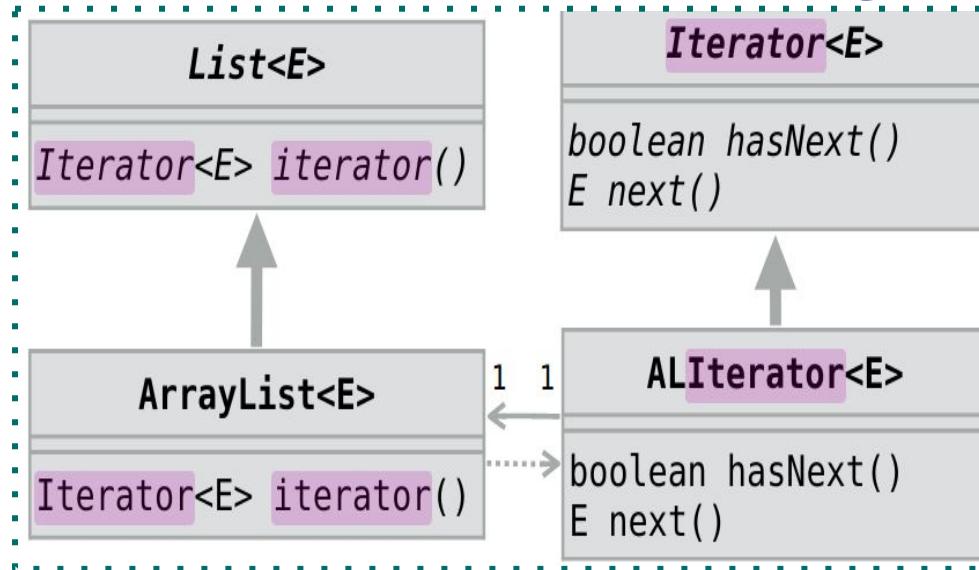
- to **access** an **aggregate object's contents** without exposing its internal representation.
- to support **multiple traversals** of aggregate objects.
- to **provide a uniform interface for traversing different aggregate structures** (that is, to support polymorphic iteration).

Design Pattern Iterator: class diagram



Design Pattern Iterator: class diagram

Example:



```
public interface Iterator<E> {
    // ... méthodes hasNext, next, etc.
}
public final class ArrayList<E>
    implements List<E> {
    // ... autres méthodes de List
    public Iterator<E> iterator() {
        return new ALIterator(this);
    }
    private final static class ALIterator<E>
        implements Iterator<E>{
            // ... méthodes hasNext, next, etc.
        }
    }
}
```

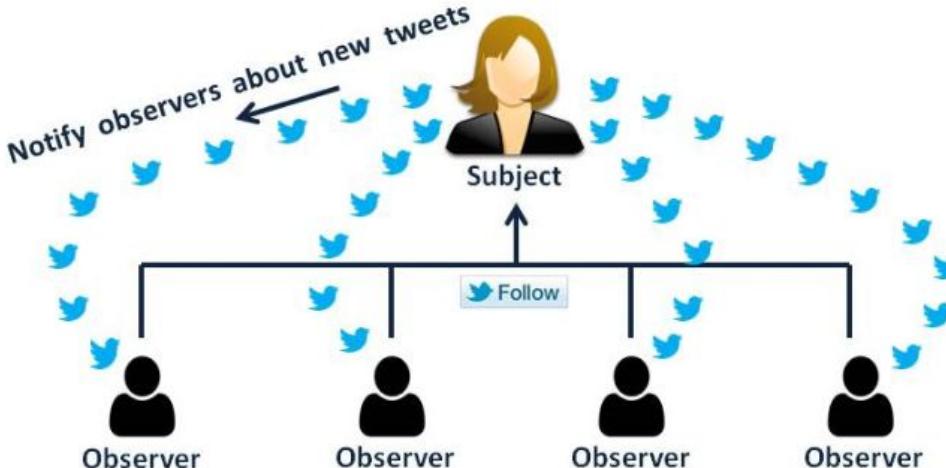
Behaviour Design Pattern

Design Pattern Observer

/ Observateur

- Goal:
 - Allow an association 1..* between objects:
 - when an object changes state, all those who depend on it are notified and updated automatically;
- Goal reached without nightmare for implementing and maintaining,
- It is the most famous design pattern. You can find it in:
 - Videos games,
 - HIM,
 - etc.

Observer Design Pattern



2 roles:

- The **Subject** / Observed:
 - a. can change state,
 - b. notifies the **Observers** that he has changed,
 - c. can furnish a new state;
- The **Observers**:
 - a. can subscribe and unsubscribe to the **subject** state modifications,
 - b. are notified if there are subscribers,
 - c. can retrieve the state of the **subject**.

Design Pattern

Observateur / Observer

Observed

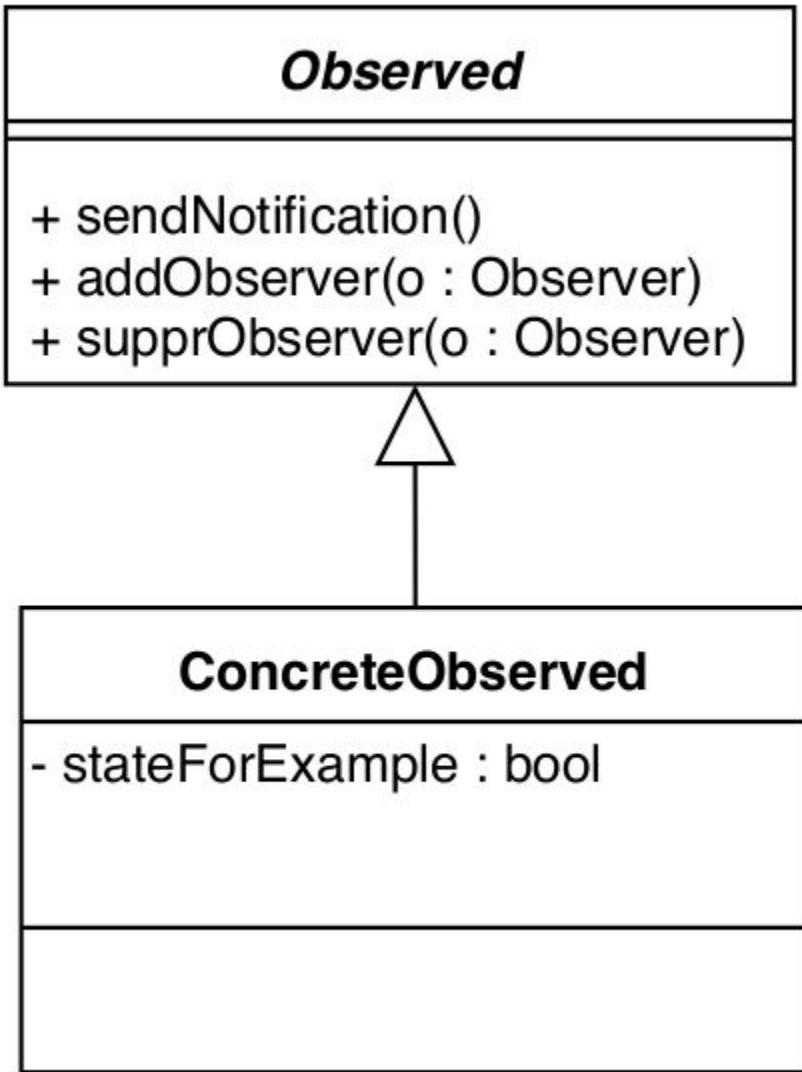
- + sendNotification()
- + addObserver(o : Observer)
- + suprObserver(o : Observer)

The **Subject** / *Observed*

Common methods:

- Can send notification,
- Can add an observer,
- Can delete an observer.

Design Pattern Observateur / Observer



The Subject / Observed

Common methods:

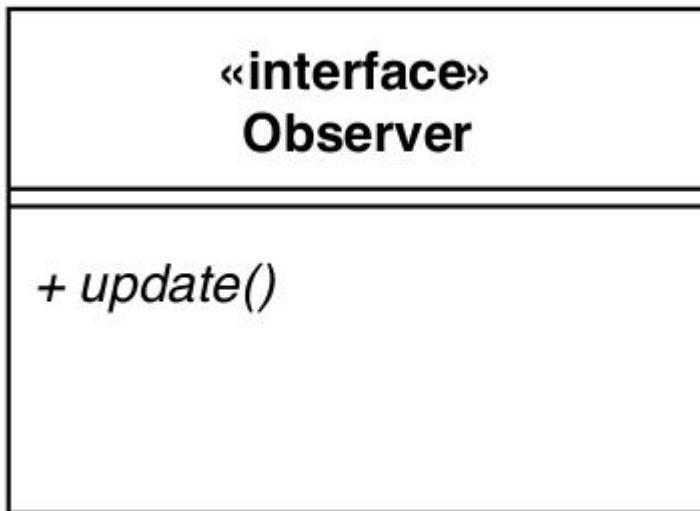
- Can send notification,
- Can add an observer,
- Can delete an observer.

Specified Attributes:

- Related to the state of the Subject object (Observed).

Design Pattern Observateur / Observer

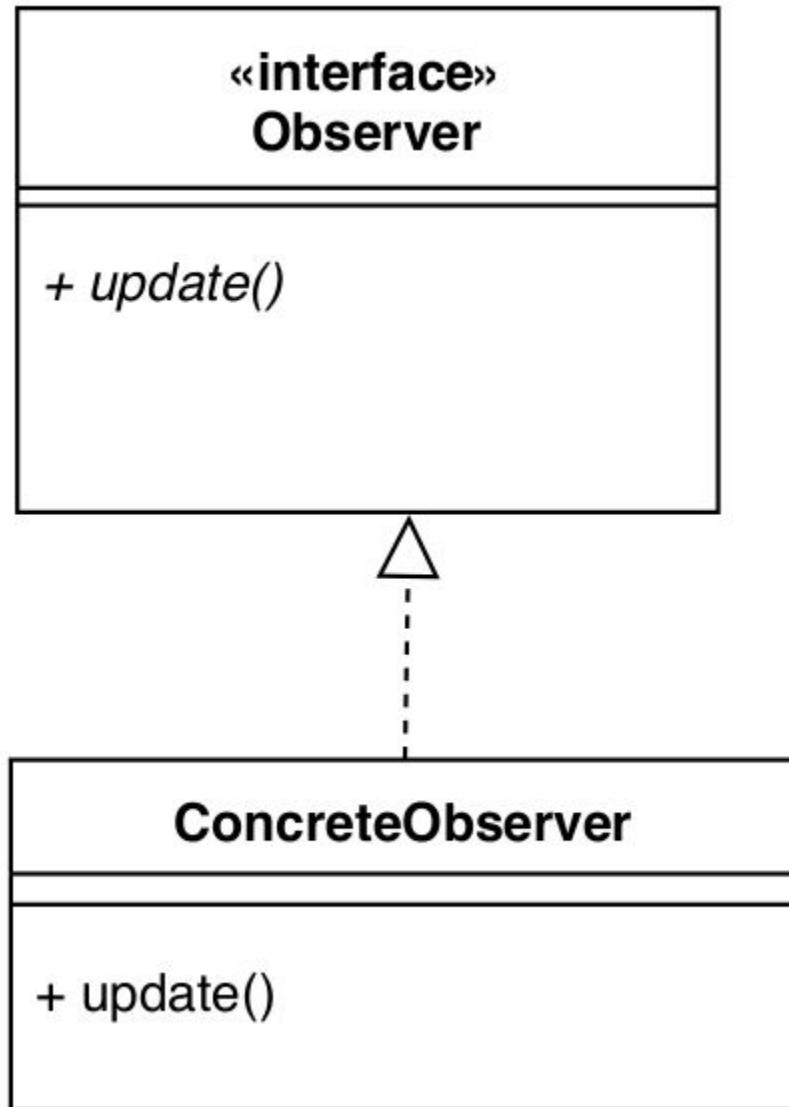
In Java: an <<interface>>.



- The observer has an update method:
 - e.g., a graph whose data has changed,
 - the update () method must be re-implemented according to the specificities of the observer.

Design Pattern

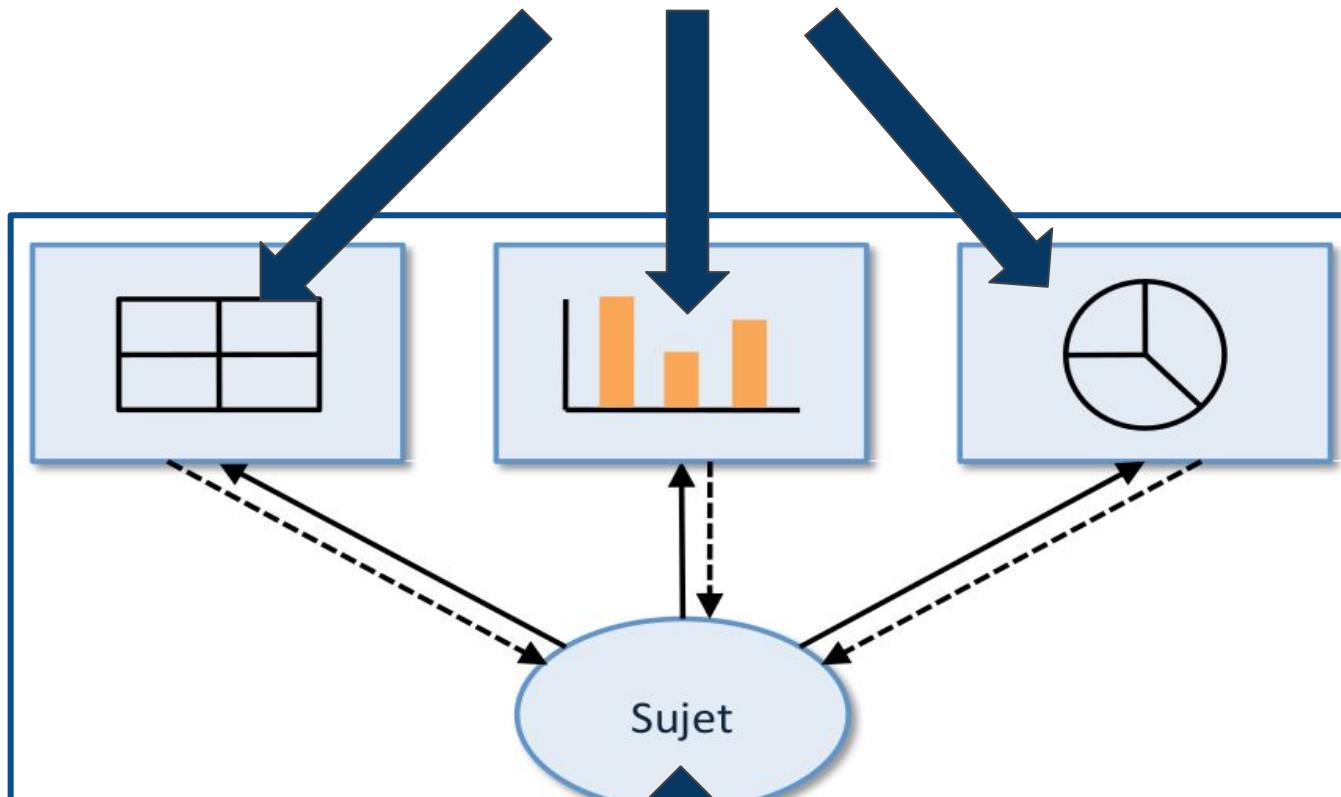
Observateur / Observer



- The observer has an update method:
 - e.g., a graph whose data has changed,
 - the update () method must be re-implemented according to the specificities of the observer.

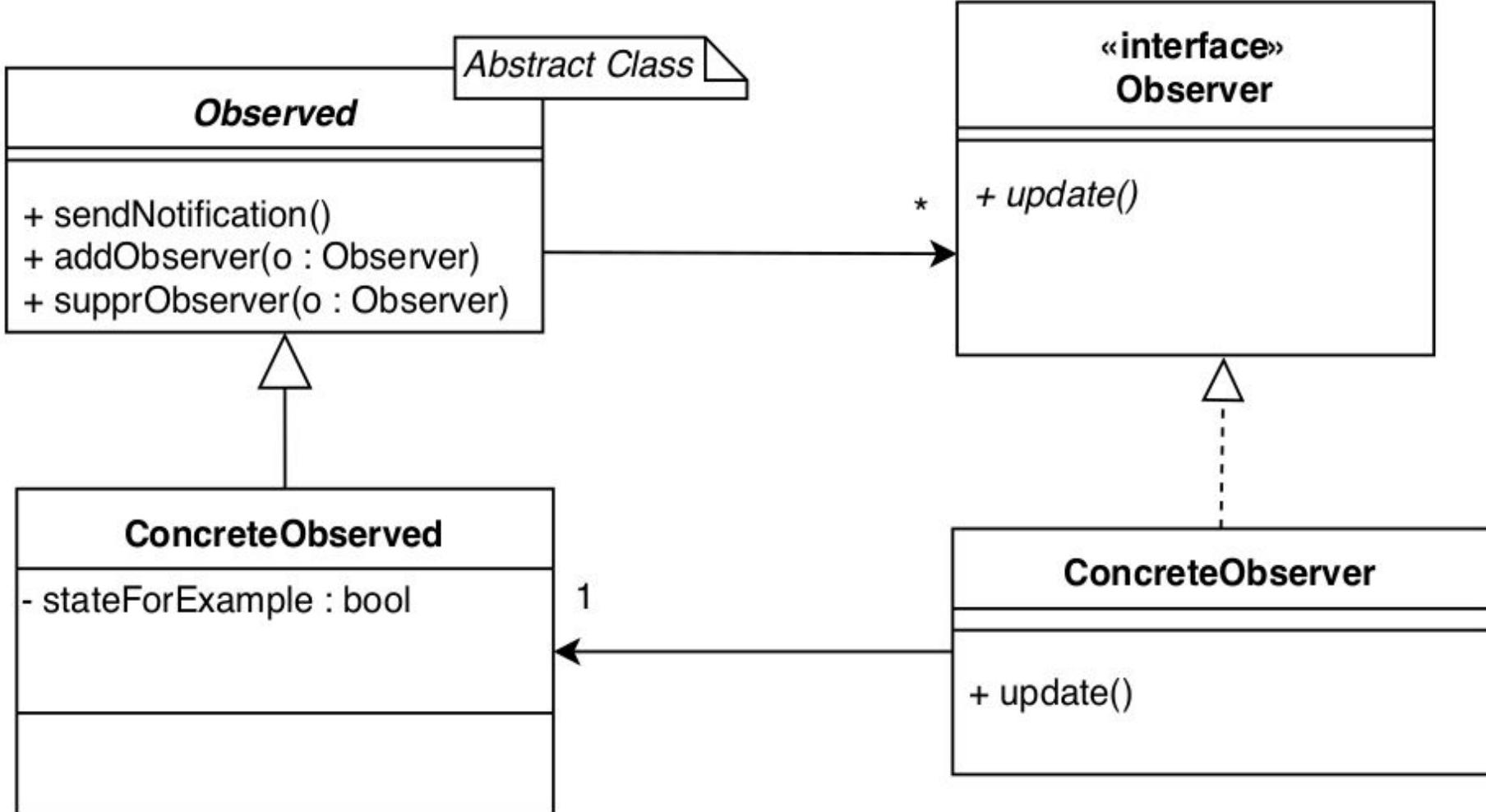
Design Pattern - Observateur / Observer

3 observers



**1 Subject
(Observed)**

Design Pattern Observer / Observateur



Design Pattern Observer

So well known that some of the work is done in java!

Inheritance of the Observed

```
import java.util.Observable;
public class ObservableValue extends Observable
{
    private int n = 0;
    public ObservableValue(int n)
    {
        this.n = n;
    }
    public void setValue(int n)
    {
        this.n = n;
        setChanged();
        notifyObservers();
    }
    public int getValue()
    {
        return n;
    }
}
```

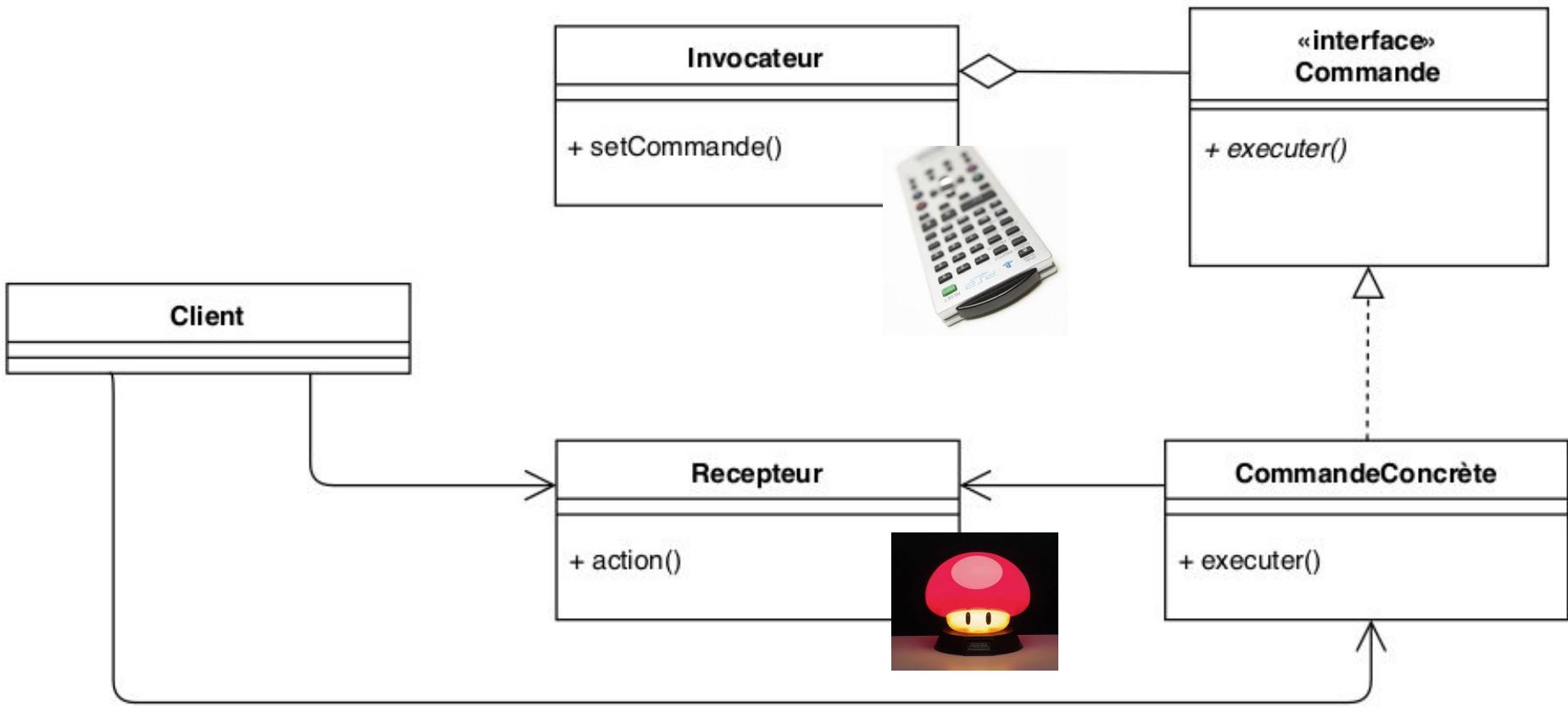
Implementation of an observer

```
import java.util.Observer;
import java.util.Observable;
public class TextObserver implements Observer
{
    private ObservableValue ov = null;
    public TextObserver(ObservableValue ov)
    {
        this.ov = ov;
    }
    public void update(Observable obs, Object obj)
    {
        if (obs == ov)
        {
            System.out.println(ov.getValue());
        }
    }
}
ObservableValue ov = new ObservableValue(0);
TextObserver to = new TextObserver(ov);
ov.addObserver(to);
main()
```

Design Pattern **Command**

- Goals:
 - Perform queries on objects without having to know their structure:
 - It makes it possible to completely separate the initiating code of the action, from the code of the action itself;
 - Easy addition of new commands:
 - Without modification of existing classes;
- Method: decoupling between the calling object and the executing one, in order to pass the request encapsulated in an object.
- All command objects implement the same interface, which contains only one method.

Command Design Pattern: the class diagram



Client: It instantiates a **ConcreteCommand** object and configures a **Receiver**.

Receiver: He knows the specific action to be performed.

ConcreteCommand: It makes the link between the **Receiver** and an action.

Summoner: He asks the **Command** to perform a request.

Command: Interface to execute an operation.

Design Pattern COMMAND

TelecommandeSimple.java

```
1 // << INVOCATEUR >>
2 public class TelecommandeSimple {
3     Commande emplacement;
4     public TelecommandeSimple() {}
5     public void setCommande(Commande commande) {
6         emplacement = commande;
7     }
8     public void boutonPresse() {
9         emplacement.executer();
0     }
}
```

Lampe.java

```
1 // << RECEPTEUR >>
2 public class Lampe {
3     public void marcheArret()
4         { System.out.println("Bouton ON/OFF");}}
```

Commande.java

```
1 // << COMMANDE >>
2 public interface Commande {
3     public void executer(); }
```

CommandeAllumerLampe.java

```
1 // << COMMANDE CONCRETE >>
2 public class CommandeAllumerLampe
3                         implements Commande {
4     Lampe lampe;
5     public CommandeAllumerLampe(Lampe lampe) {
6         this.lampe = lampe;
7     }
8     public void executer() {
9         lampe.marcheArret();
10    }
}
```

TestTelecommande.java

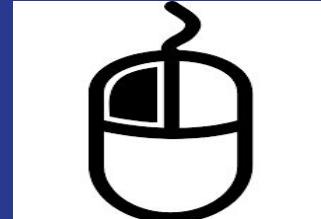
```
1 public class TestTelecommande {
2     public static void main(String[] args) {
3         TelecommandeSimple telecommande =
4             new TelecommandeSimple();
5         Lampe lampe = new Lampe();
6         CommandeAllumerLampe lampeAllumee =
7             new CommandeAllumerLampe(lampe);
8         telecommande.setCommande(lampeAllumee);
9         telecommande.boutonPresse();
10    }
}
```

Here: the client exists in the main.

Practical Work

/ Travaux Pratiques

Design Patterns



```

3 public class Bouilleur {
4     protected boolean vide;
5     protected boolean bouilli;
6
7     public Bouilleur() {
8         vide = true;
9         bouilli = false;
10    }
11    public void remplir() {
12        if (estVide()) {
13            vide = false;
14            bouilli = false;
15        }
16    }
17    public void vider() {
18        if (!estVide() && estBouilli()) {
19
20            vide = true;
21        }
22    }
23    public void bouillir() {
24        if (!estVide() && !estBouilli()) {
25
26            bouilli = true;
27        }
28    }
29    public boolean estVide() {
30        return vide;
31    }
32    public boolean estBouilli() {
33        return bouilli;
34    }
35 }

```

Design Pattern Singleton

After having set fire to your chemical plant a dozen times, you understand that the problem lies with your boiler. Indeed, its controller manages several instances of the Boiler class then attached to the same machine...

After having imagined what could have happened with 2 instances, you set about adapting the Design Pattern Singleton to your Bouilleur class in order to solve the problem.

Practical
Work 9



```

3 public class Bouilleur {
4     protected boolean vide;
5     protected boolean bouilli;
6
7     public Bouilleur() {
8         vide = true;
9         bouilli = false;
10    }
11    public void remplir() {
12        if (estVide()) {
13            vide = false;
14            bouilli = false;
15        }
16    }
17    public void vider() {
18        if (!estVide() && estBouilli()) {
19
20            vide = true;
21        }
22    }
23    public void bouillir() {
24        if (!estVide() && !estBouilli()) {
25
26            bouilli = true;
27        }
28    }
29    public boolean estVide() {
30        return vide;
31    }
32    public boolean estBouilli() {
33        return bouilli;
34    }
35 }

```

Design Pattern Singleton

Après avoir mis le feu une bonne dizaine de fois à votre usine chimique, vous comprenez que le problème vient de votre bouilleur. En effet, son contrôleur gère plusieurs instances de la classe Bouilleur alors rattachées à la même machine...

Après avoir imaginé ce qui a pu se passer avec 2 instances, vous employez à adapter le *Design Pattern* Singleton à votre classe Bouilleur afin de régler le problème.

Practical
Work 9

“Post Office”

- You are a customer of the post office who expects to receive new messages every day.
- So, every day, you go to the post office to check if new messages have arrived.
- To model this process, we will use three classes in JAVA:
 - **PostOffice**,
 - **Person**,
 - **Main**.



“Il y a écrit la poste”

- Vous êtes un client du bureau de poste qui s'attend à recevoir chaque jour de nouveaux messages.
- Ainsi, chaque jour, vous allez au bureau de poste pour vérifier si de nouveaux messages sont bien arrivés.
- Pour modéliser cela ce processus, nous allons utiliser trois classes en JAVA :
 - **PostOffice**,
 - **Person**,
 - **Main**.

Of course, this is an easy way to deal with postal mail. However, this poses a lot of problems since a large number of customers will physically check that new mail is available. For example :

- One could fix the problem by going through the customer list and checking if they have new mail, but that would mean the post office would be too busy with too many requests.
- Thanks to a well-known Design Pattern, but without using Java classes (i.e., from the library), structure your code in order to make such a system more efficient. For example, it will be easy to have clients of different types (companies, people etc.).

Your program should include the functionality to add and remove mail subscribers. Set up a mechanism that alerts all customers whenever new mail is received.

Bien entendu, il s'agit d'une manière simple de gérer le courrier postal. Cela pose cependant beaucoup de problèmes dès lors qu'un grand nombre de client va physiquement vérifier qu'un nouveau courrier est disponible. Par exemple :

- On pourrait résoudre le problème en parcourant la liste des clients et en vérifiant s'ils ont un nouveau courrier, mais cela signifierait que le bureau de poste serait inondé de demandes.
- Grâce à un *Design Pattern* bien connu, mais sans utiliser les classes Java qui sont propres, structurez votre code afin de rendre un tel système plus efficace. Il sera, par exemple, facile d'avoir des clients de types différents (entreprises, personnes etc.).

Votre programme doit comprendre les fonctionnalités d'ajout et de suppression d'abonnés à la poste. Mettez en place un mécanisme qui alerte tous les clients chaque fois qu'un nouveau courrier est reçu.

Recopy
and then run the
following code.

```
1 import java.lang.*;                                Main.java
2
3 class Main{
4
5     public static void main(String [] args)
6     {
7         PostOffice postOffice=new PostOffice();
8         Person subscriber=new Person("Chris")
9         subscriber.checkMail(postOffice);
10
11        postOffice.addMail("Chris");
12        subscriber.checkMail(postOffice);
13
14    }
}
```

Person.java

```
1 import java.lang.*;
2
3 class Person {
4
5     String name;
6
7     Person(String n){
8         name=n;
9     }
10
11    public void checkMail(PostOffice office){
12        if(office.checkMail(this)){
13            System.out.println("Mail found");
14        }else{
15            System.out.println("Mail not found");
16        }
17    }
18
19    public String getName(){
20        return name;
21    }
22}
```

PostOffice.java

```
import java.lang.*;
import java.util.ArrayList;

class PostOffice{
    ArrayList<String> allMail;

    PostOffice(){
        allMail=new ArrayList<>();
    }

    public boolean checkMail(Subscriber person){

        for(int i=0; i < allMail.size(); i++){

            if(allMail.get(i)==person.getName()){
                return true;
            }
        }
        return false;
    }

    public void addMail(String s){
        allMail.add(s);
    }
}
```

Practical
Work 10

Bibliothèque & Design Pattern



You continue to take care of the management of a library, this time for the development of the Database.

You make sure that, in order not to lose Data, there will only be a maximum of one Database. Only one method is available to retrieve this database, it has for parameter a name which will be used to name the Database, but, if this one already exists, this parameter will have no effect (only the first one can name the database.).

Find and implement the right Design Pattern that will solve the problem.
Unroll your code over two attempts to retrieve the Database.



Bibliothèque & Design Pattern



Vous continuez à vous occuper de la gestion d'une bibliothèque et vous vous attaquez désormais à un gros morceau : celui du développement de la Base de Données.

Vous vous assurez que, pour ne pas égarer de Données, il n'existera qu'au maximum une seule Base. Une seule méthode est disponible pour récupérer cette base de données, elle a pour paramètre un nom qui sera utilisé pour nommer la Base, mais, si celle-ci existe déjà, ce paramètre n'aura aucun effet (seul le premier peut nommer la base).

Cherchez et implémentez le bon Design Pattern qui pourra régler le problème. Dérouler votre code sur deux tentatives de récupération de la Base.



Object Concept

It's over.

