Object-Oriented Programming in Java

Guillaume Muller

Telecom Saint-Étienne, Laboratoire Hubert-Curien

7 october 2019

Object-Oriented Programming

- Who knows what it is???
- Who has already used it???
- Who can cite it core principles???
- Who can cite a few languages using it???

Programming Paradigms[1]- Why ?

- Why do we need paradigms?
- Complex System $\Rightarrow$ "Divide and Conquer"
- **Partitioning** the problem $\Rightarrow$ simpler discrete pieces
- Create **interfaces** & **interactions** between these pieces

[1] Thought patterns/« façons de voir le monde ».

Programming Paradigms[1]- How ?

- **Declarative** (what to execute ⇒ goal to achieve)
  - *SQL*, *Prolog*

[1] Thought patterns/« façons de voir le monde ».

Programming Paradigms[1]-

- **Declarative** (what to execute ⇒ goal to achieve)
  - *SQL*, *Prolog*
- **Imperative** (how to execute ⇒ explicit control flow)
  - **Procedural**
    - *C*, *Pascal*, *Python*, *Lisp*...
  - **Functional**
    - *Caml*, *Haskell*, *Erlang*, *(Java)*...
  - **Object-Oriented**
    - *Java*, *C♯*, *(Python)*...

[1] Thought patterns/« façons de voir le monde ».

Programming Paradigms[1]-

- **Declarative** (what to execute ⇒ goal to achieve)
  - *SQL*, *Prolog*
- **Imperative** (how to execute ⇒ explicit control flow)
  - **Procedural**
    - *C*, *Pascal*, *Python*, *Lisp*...
  - **Functional**
    - *Caml*, *Haskell*, *Erlang*, *(Java)*...
  - **Object-Oriented**
    - *Java*, *C♯*, *(Python)*...
- **Event-Driven**
  - *Javascript*, *VisualBasic*...

[1] Thought patterns/« façons de voir le monde ».

Programming Paradigms[1]-

- **Declarative** (what to execute ⇒ goal to achieve)
  - *SQL*, *Prolog*
- **Imperative** (how to execute ⇒ explicit control flow)
  - **Procedural**
    - *C*, *Pascal*, *Python*, *Lisp*...
  - **Functional**
    - *Caml*, *Haskell*, *Erlang*, *(Java)*...
  - **Object-Oriented**
    - *Java*, *C♯*, *(Python)*...
- **Event-Driven**
  - *Javascript*, *VisualBasic*...
- **Reactive** (streams)
  - *(Java)*...

[1] Thought patterns/« façons de voir le monde ».

Objects & Classes

Objects & Classes

- Cars of different types

Objects & Classes

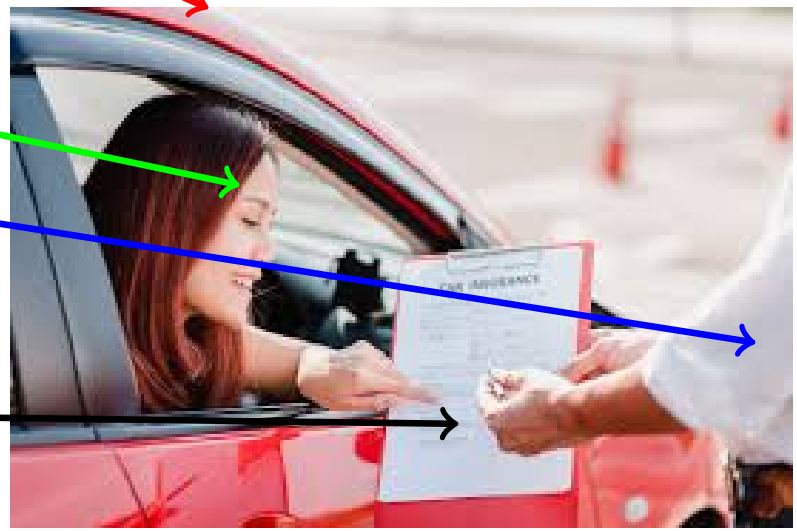
- Cars of different types
- Clients

Objects & Classes

- Cars of different types
- Clients
- Employees (authorizations?)

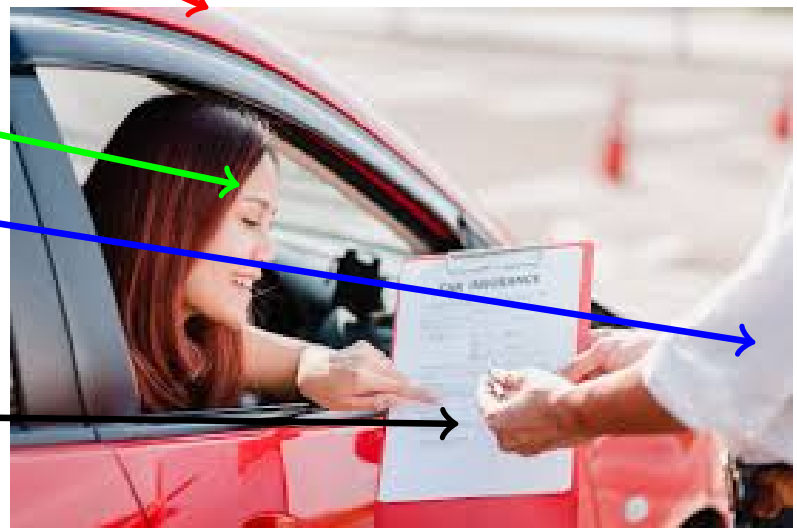
Objects & Classes

- Cars of different types
- Clients
- Employees (authorizations?)
- Contracts



Objects & Classes

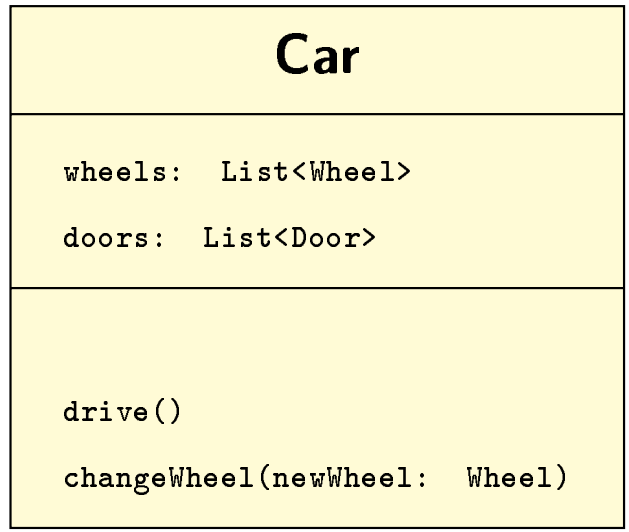
- Cars of different types
- Clients
- Employees (authorizations?)
- Contracts



World = Set of typed objects

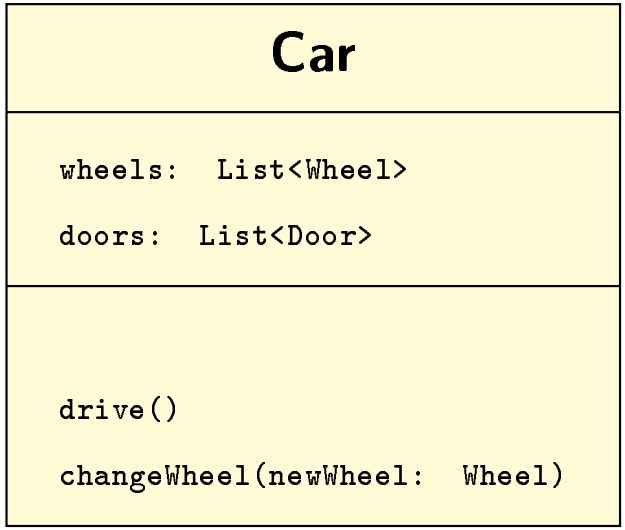
Objects & Classes

- **Classes** = Object *types/models*
  - Attributes = **Structure**
  - Methods (~~"functions"~~) = **Behaviour**



Objects & Classes

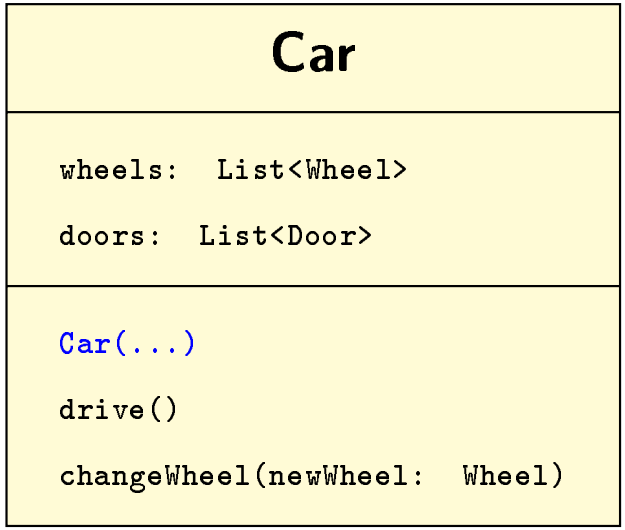
- **Classes** = Object *types/models*
  - Attributes = **Structure**
  - Methods (~~"functions"~~) = **Behaviour**
  - Attributes + Methods = « **members** »



Objects & Classes

- **Classes** = Object *types/models*
  - Attributes = **Structure**
  - Methods (~~"functions"~~) = **Behaviour**

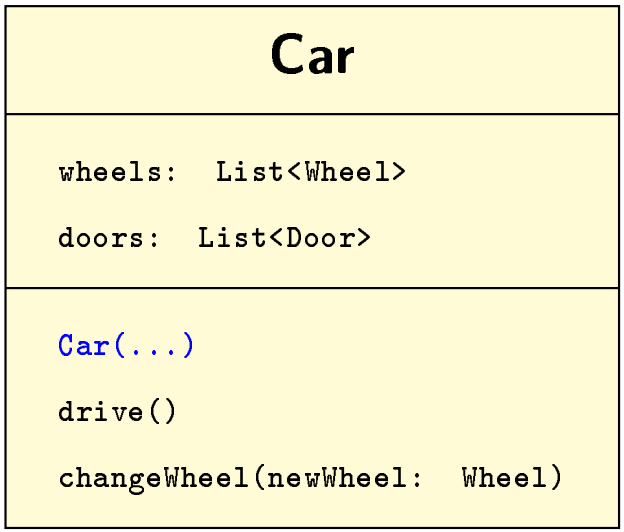
    constructors() = create instances
  - Attributes + Methods = « **members** »



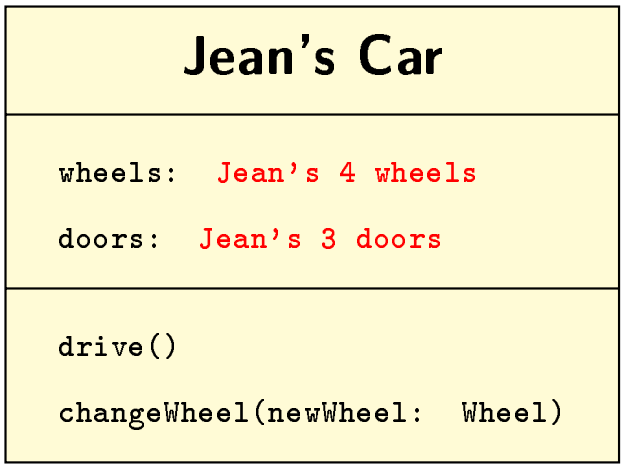
Objects & Classes

- **Classes** = Object *types/models*
  - Attributes = **Structure**
  - Methods (~~"functions"~~) = **Behaviour**

    constructors() = create instances
  - Attributes + Methods = « **members** »

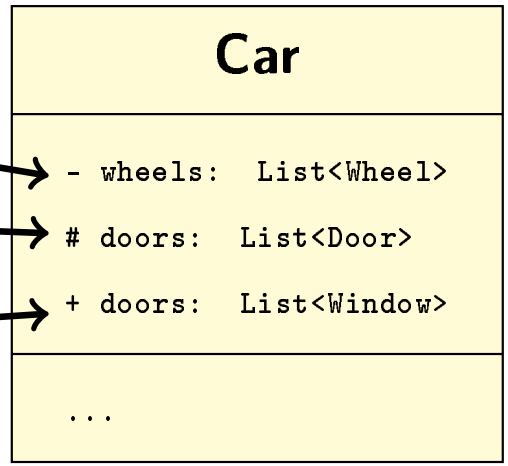


- **Objects** = *valued instances*



Visibility & Encapsulation

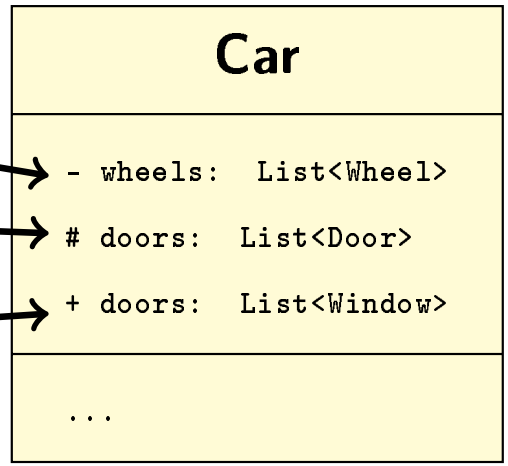
- \- = **private**
  - same class
- \# = **protected**
  - same class, same package, derived
- \+ = **public**
  - everybody



[2] cf. Wikipedia.

Visibility & Encapsulation

- \- = **private**
  - same class
- \# = **protected**
  - same class, same package, derived
- \+ = **public**
  - everybody



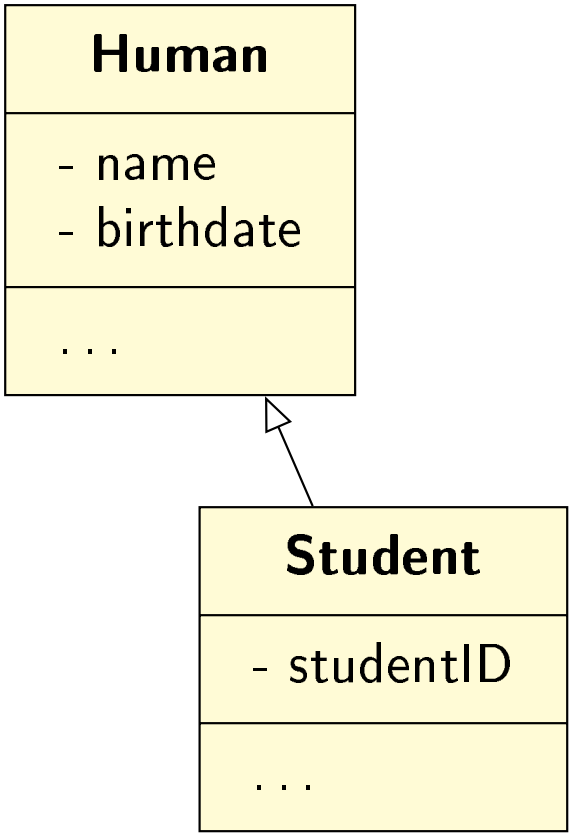
- **Good practice** (SOLID, KISS, YAGNI, GRASP...)[2]
  - **private** for all Attributes
  - public/protected **Accessors** (getXXX()/setXXX())
  - E.g.: only a `public String getName()`
    $\Rightarrow$ nobody can change the "`name`" attribute

[2] cf. Wikipedia.

Inheritance & Polymorphism

- Code Re-Use
  - Objects of a class also belong to a *more general* class
  - Subclass: **all members** of mother class + **own members**
  - Translation of **« to be »**
    (« all Students are Humans »)

- Example and vocabulary:
  - Human **generalizes** Student
  - Student **specializes** Human
  - Student **derives/inherits** from Human
  - Student is **subclass** of Human

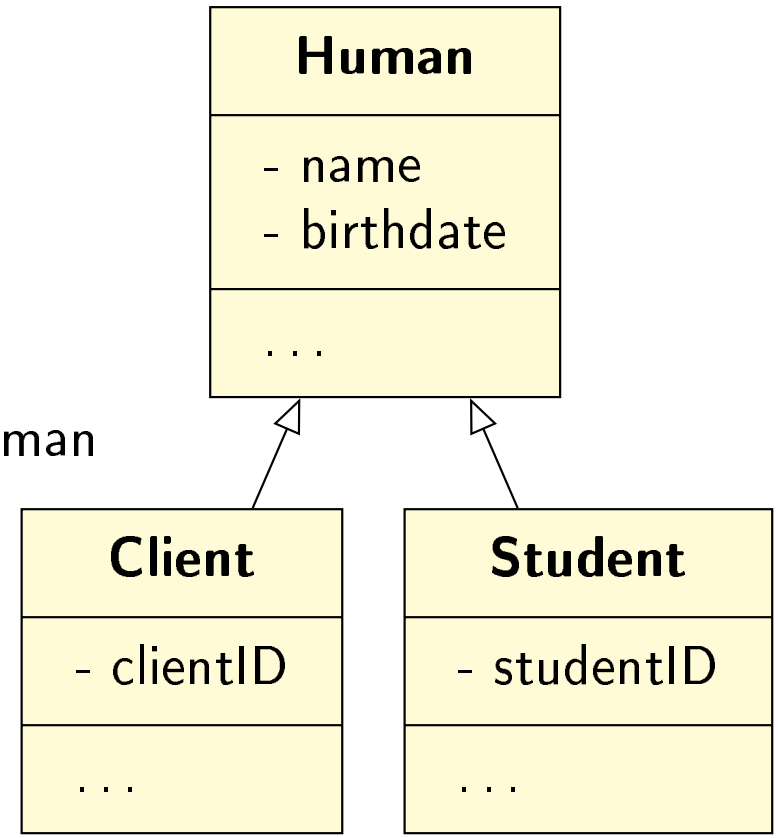


Inheritance & Polymorphism

- Code Re-Use
  - Objects of a class also belong to a *more general* class
  - Subclass: **all members** of mother class + **own members**
  - Translation of **« to be »**
    (« all Students are Humans »)

- Example and vocabulary:
  - Human **generalizes** Student
  - Student **specializes** Human
  - Student **derives/inherits** from Human
  - Student is **subclass** of Human

- **Poly[multiple]-morphism[forms]**
  - an object/instance can be **both** `Student` **AND** `Client`



Over-Loading & Over-Writing/Over-Riding

- Over-**Writing/Riding**  

```
1       public class Human {
2         public void setName(String fullName) { // same prototype
3           ...
4         }
5       }
6       ...
7       public class Student %\textbf{extends} Human {
8         public void setName(String firstName) { // same prototype
9           ...                                   // specific body
10        }
11      }
```

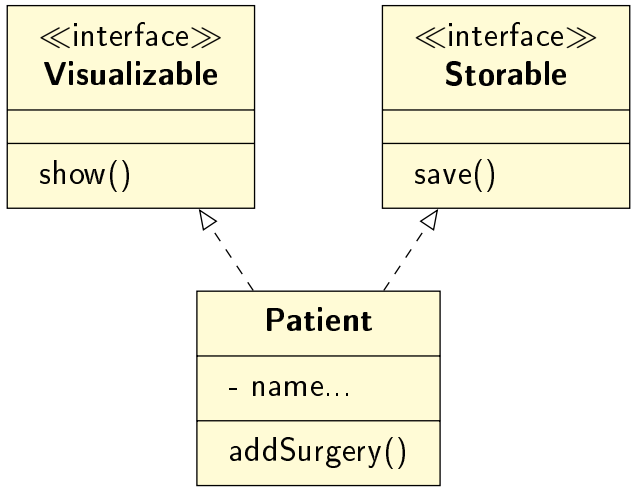
- Over-**Loading**  [3]

```
1       public void setName(String fullName) // prototypes are different
2         ...
3       }
4       public void setName(String firstName, String firstName) { // "HEAVIER"
5         ...
6       }
```

[3] prototype/signature = name + arguments&types (~~return~~).

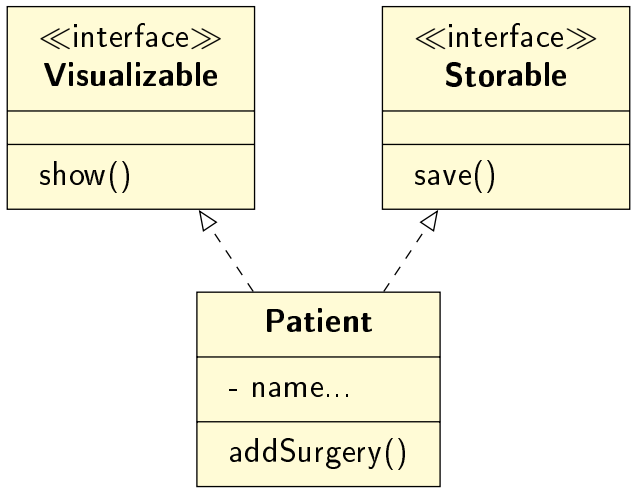
Interfaces & Abstract Classes

- **Interfaces**
  - Interfaces define standard signatures for methods
    - **only methods *signatures***, **no** attribute
    - C++ equivalent: pure virtual classes
  - Usage: « contracts »
    - Classes that implement an interface « ensure » they conform to its methods declarations
  - A lot of interfaces are defined in java libraries
    `Collection`, `Serializable`, `Component`...
  - Multiple implementation ⇒ **OK**



Interfaces & Abstract Classes

- **Interfaces**
  - Interfaces define standard signatures for methods
    - **only methods *signatures***, **no** attribute
    - C++ equivalent: pure virtual classes
  - Usage: « contracts »
    - Classes that implement an interface « ensure » they conform to its methods declarations
  - A lot of interfaces are defined in java libraries
    `Collection`, `Serializable`, `Component`...
  - Multiple implementation ⇒ **OK**



- **Abstract classes**
  - Can have attributes
  - Methods can be implemented
  - Multiple inheritance ⇒ **KO** (in Java)

Writing re-usable code

- Java By Comparison

  https://www.amazon.com/Java-Comparison-Become-Craftsman-Examples/dp/1680502875/

  https://github.com/GMTSE/ProjetsJavaMaterial/blob/master/JavaByComparisonSumUp.md

  - Rule♯5: always check for **null args**
  - Rule♯8/♯14: **group** code / **indent**
  - Rule♯72: log in file not console
  - Rule♯11: favor for-each
  - Rule♯15: use string format
  - Rule♯68: 1 **code style** for the team
  - Rule♯21: template for **comments**
  - Rule♯29: select **good names**
  - Rule♯34: catch most specific Exception
  - Rule♯39: close resources
  - Rule♯42: template for Unit Tests
  - Rule♯50: treat edge test cases
  - Rule♯54: favor immutable
  - Rule♯64: use optionals rather than nulls