Object-Oriented Programming in Java

Guillaume Muller

Telecom Saint-Étienne, Laboratoire Hubert-Curien

14 September 2020

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 ⇒ "Divide and Conquer"
- **Partitioning** the problem ⇒ 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 $\Rightarrow$ 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 $\Rightarrow$ goal to achieve)
  - *SQL, Prolog*
- **Imperative** (how to execute $\Rightarrow$ 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

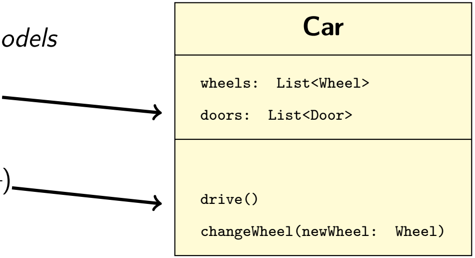
Objects & Classes

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

World = Set of typed objects

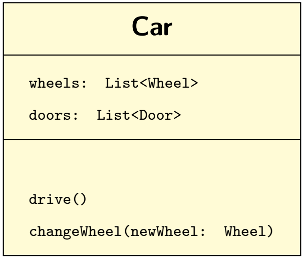
Objects & Classes

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



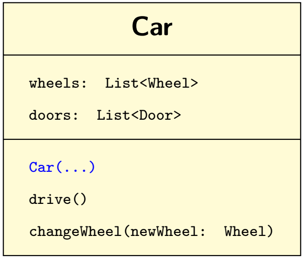
Objects & Classes

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



Objects & Classes

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



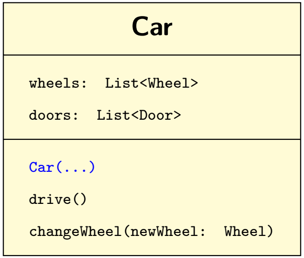
Car

```
wheels:  List<Wheel>
doors:  List<Door>
```

```
Car(...)
drive()
changeWheel(newWheel:  Wheel)
```

Objects & Classes

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

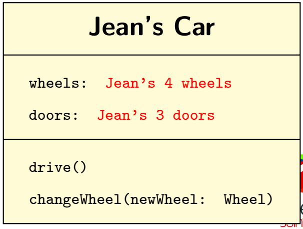


```
Car

wheels:  List<Wheel>
doors:  List<Door>

Car(...)
drive()
changeWheel(newWheel:  Wheel)
```

- **Objects** = *valued instances*



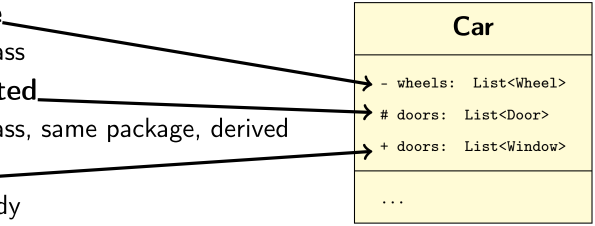
```
Jean's Car

wheels:  Jean's 4 wheels
doors:  Jean's 3 doors

drive()
changeWheel(newWheel:  Wheel)
```

Visibility & Encapsulation

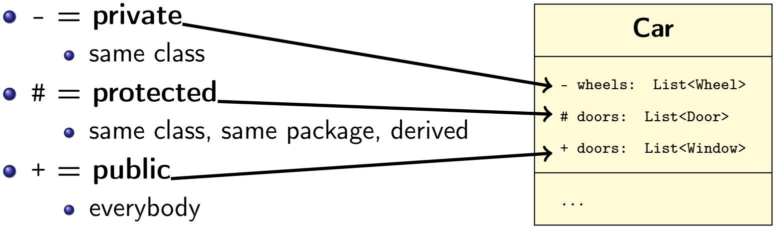
- - = **private**
  - same class
- # = **protected**
  - same class, same package, derived
- + = **public**
  - everybody



[2] cf. Wikipedia.

Visibility & Encapsulation

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

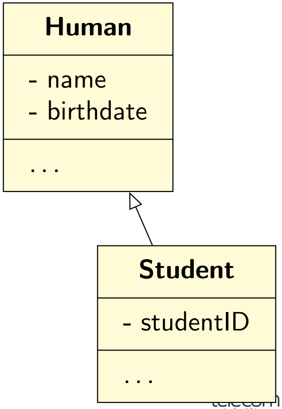


- **Good practice** (SOLID, KISS, YAGNI, GRASP...)[2]
  - **private** for all Attributes
  - public/protected **Accessors** (getXXX()/setXXX())
  - E.g.: only a `public String getName()`
    ⇒ nobody can change the "`name`" attribute

[2] cf. Wikipedia.

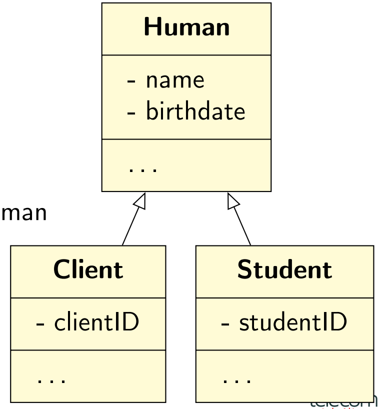
Inheritance & Polymorphism

- 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**

```
public class Human {
  public void setName(String fullName) { // same prototype
    ...
  }
}
...
public class Student \textbf{extends} Human {
  public void setName(String firstName) { // same prototype
    ...                                  // specific body
  }
}
```

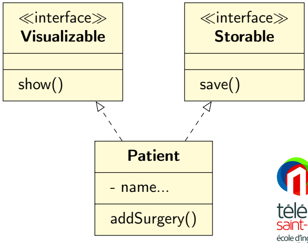
- Over-**Loading** [3]

```
public void setName(String fullName) // prototypes are different
  ...
}
public void setName(String firstName, String firstName) { // "HEAVIER"
  ...
}
```

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

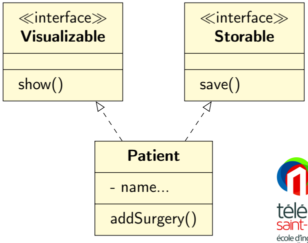
Interfaces & Abstract Classes

- **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)