

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 its core principles???
- Who can cite a few languages using it???

Programming Paradigms¹- 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

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

Programming Paradigms¹- How ?

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

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

Programming Paradigms¹-

- **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)...*

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

Programming Paradigms¹-

- **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...*

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

Programming Paradigms¹-

- **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...*
- **Reactive** (streams)
 - *(Java)...*

¹Thought 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



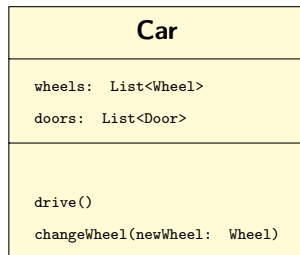
World = Set of typed objects

Objects & Classes

- **Classes** = Object *types/models*

- Attributes (or Fields)
= **Structure**

- Methods ("functions")
= **Behaviour**



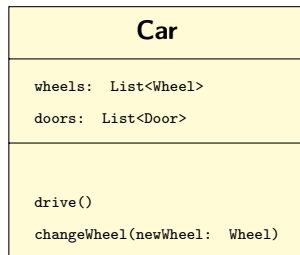
Objects & Classes

- **Classes** = Object *types/models*

- Attributes (or Fields)
= **Structure**

- Methods ("functions")
= **Behaviour**

- Attributes + Methods = « **members** »



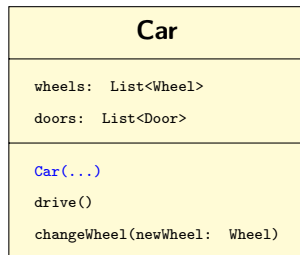
Objects & Classes

- **Classes** = Object *types/models*

- Attributes (or Fields)
= **Structure**

- Methods ("functions")
= **Behaviour**
constructors() = create instances

- Attributes + Methods = « **members** »



Objects & Classes

- **Classes** = Object *types/models*

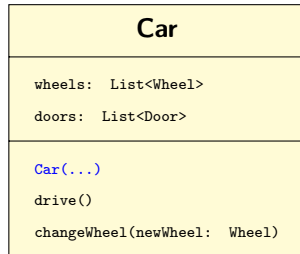
- Attributes (or Fields)
= **Structure**



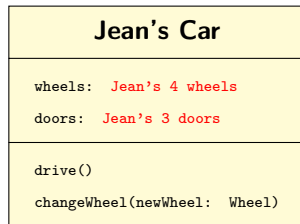
- Methods ("functions")
= **Behaviour**
constructors() = create instances



- Attributes + Methods = « **members** »

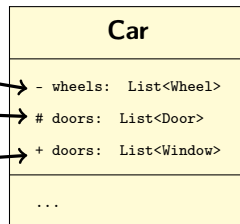


- **Objects** = *valued instances*



Visibility & Encapsulation

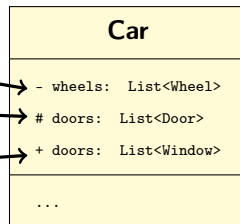
- - = **private**
 - same class
- # = **protected**
 - same class, same package, derived
- + = **public**
 - everybody



²cf. 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

²cf. Wikipedia.

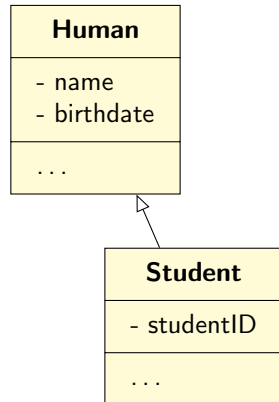
Inheritance & Polymorphism

- 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 »)

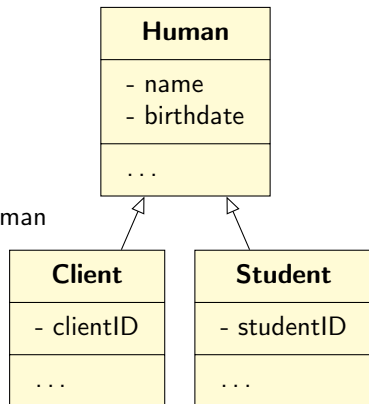
- Example and vocabulary:

- 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



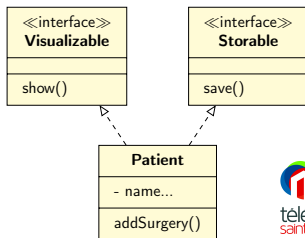
```
1      public void setName(String fullName) // prototypes are different
2          ...
3      }
4      public void setName(String firstName, String firstName) { // "HEAVIER"
5          ...
6      }
```

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

Interfaces & Abstract Classes

- **Interfaces**

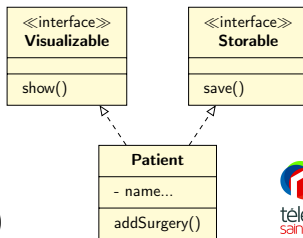
- Interfaces define standard signatures for methods
 - **only methods *signatures*, no attribute**
 - C++ equivalent: pure virtual classes
- Usage: « contracts »
 - Classes that implement an interface « ensure » they conform to its methods declarations
- A lot of interfaces are defined in java libraries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