3TC36: Object-Oriented Programming in Java

Introduction to the Object-Oriented paradigm (OO)

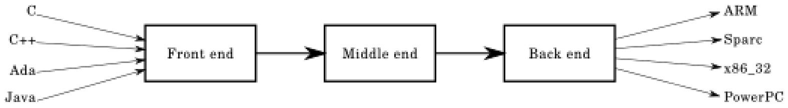
Dominique Blouin
dominique.blouin@telecom-paris.fr
Le February 17, 2026

Learning objectives for this period

- Programs and programming
- Introduction to the Object-Oriented Programming paradigm (OOP)

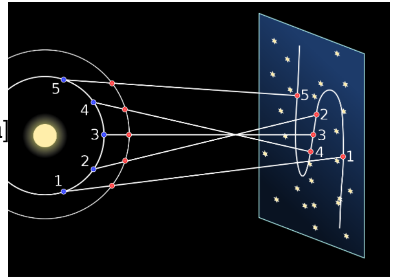
Programming languages

- A programming language allows the programmer to write their program using high-level concepts.
  - For example, all languages provide the concept of a **list** of numeric data.
- The **compiler** translates these high-level concepts into instructions for the ALU (Arithmetic Logic Unit).



- These instructions, encoded in bytes, will be loaded into memory for program execution.

Programming paradigms



- Paradigm: A programming paradigm is simply a pattern – a way of thinking about and organizing your code. Different paradigms give you different mental models for solving problems. [Wikipedia]
- Scientific paradigms:
  - Geocentrism vs. heliocentrism in astronomy.
  - Importance of paradigm **shifts**.
- Engineering paradigms: how to **solve a problem**.
  - Concern all stages of the system life cycle (design, implementation, maintenance)
  - Also concern the **environment** in which engineering occurs:
    - Methods and tools like models and their languages, development processes, etc.
    - For example, the Agile method can be seen as an engineering paradigm.
- Common characteristics between scientific and engineering paradigms:
  - Means to **characterize** a set of **artifacts** used in an **environment** to solve a problem.
- **Programming paradigms:**
  - Characterize a **programming language** (artifact) by its **syntax** and **semantics**.

Types of programming paradigms

- Imperative (or procedural) paradigms: programs consisting of **commands** whose execution is determined by **control structures.**
  - There are data structures like an array of numbers.
  - Procedures take these data structures as parameters to perform actions on the data and/or compute results from it.
  - Example: a procedure to sort (reorder) the elements of the array in ascending order.
- We must tell the computer **how** to do it.
- This is the most common type of programming.
  - Allows solving problems for which we can construct a sequence of commands that provide a solution.

Extensions of the imperative paradigm

- **Object-Oriented (OO)**: the language allows describing or modelling a problem by a **collection of objects** that communicate with each other by sending **messages**.
  - Java, C#, Smalltalk, Python, Simula, etc.
  - **Modelling** languages (Modelica, Simulink, Scade, etc.).
- OO brings ease of programming and a **clearer** vision of the problem.
  - Direct correspondence with **real-world objects**.
- The key shift: instead of **doing things to data**, we **ask objects to do things**.

Declarative paradigms

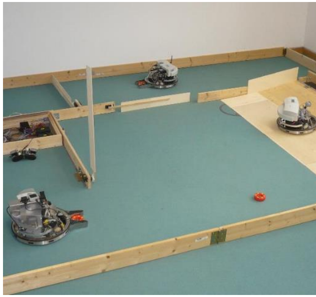
- Describe a **problem** (often in the form of logical formulas).
  - The execution of the program consists of finding one or more **solutions** to the given problem.
- Examples:
  - SQL: a `select` statement specifies **which data is required** (the WHAT or the problem), and the execution engine finds a table traversal (the HOW or a solution) to retrieve the data.
  - Prolog...
- Extensions of declarative paradigms:
  - Functional programming
  - Constraint programming
  - Graph-oriented programming
  - Etc.

The birth of OOP

- Introduced by the **Simula** language (SIMple Universal LAnguage) at the University of Oslo in 1967.
- The problem at hand was simulating a group of robots within a company.
- Trying to solve this problem using traditional imperative programming is a real challenge due to the many and unpredictable interactions between objects.
- A centralized program that controls the robots by modifying a data structure representing the state of **all the robots** and the **environment** is not impossible, but it is very difficult.
- Solution: Object-Oriented programming.

Principles of OOP

- Describe each element of the problem by **classes**:
  - This concerns the robots, but also all the elements of the problem, such as:
    - the factory containing the robots
    - the rooms, the doors, the production machines, etc.
- A class describes the **data** contained in an object:
  - Example: A robot has **attributes** like:
    - **Energy level** (*e.g.,* 0 to 100).
    - **Position** (composed of two coordinates).
    - **Speed vector**, etc.



- **Attributes** are similar to a **data structure**, as learned in **C** (3TC31).
- Key difference: objects can send **messages** to each other.
- An **object** calls a method to perform **actions** (calculations) and **responds** with a result.

Simula robots

- Examples of messages:
  - The user sends a message to a robot to tell it to start or stop its operation.
  - A robot notifies the control system that its energy level is too low.
  - Two robots collide, and the sensors send messages.
  - Etc.
- The class also describes how objects **respond** to messages. These are the object's **methods**.
  - For example, if a robot receives the message `start()`, it starts and responds with `done` if the operation was successful or `failed` if it was unsuccessful.

Robot simulation

- The programmer describes each object in the factory using a **class**.
- When the program starts, the programmer creates **objects**:
  - A control station.
  - Robots.
  - Etc.
- Objects interact **by sending messages** to each other.
- Initial messages are sent by the user of the simulator.
- There is **no centralized control** managing all the actions of the robots and their environment.

Example with a simple graphical interface

- Everything that appears on a computer screen is part of the graphical interface: windows, menus, buttons, input fields, drawing areas, etc.
- The user, through interactions with the keyboard and mouse, triggers program **actions**:
  - Creation of a new window.
  - Actions associated with a button.
  - Etc.
- Some changes occur automatically, such as animations.
- As with robots, we can imagine a data structure describing the state of **the graphical interface and all its elements.**
- It is equally challenging to imagine a centralized control managing this graphical interface.

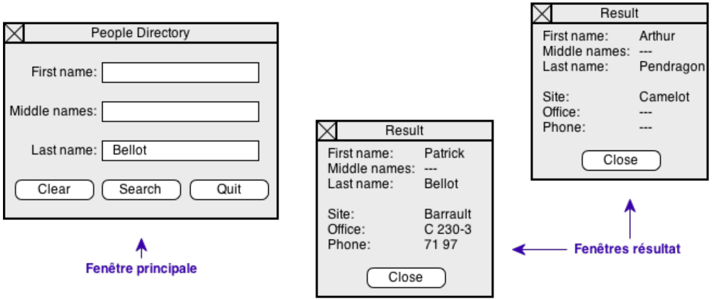
The OO approach

- In the OO approach to graphical interfaces, every element appearing on the screen is an **object.**
- Each object has **attributes**:
  - Its coordinates on the screen.
  - Its dimensions.
  - Its depth (what is in front of what?).
  - Etc.
- Each object has **methods** that allow it **to respond to messages** it receives.
- The attributes and methods of an object depend on its **type**:
  - Window, button, etc.
- Each type of object is described by a **class** containing the declarations of **attributes** and **methods**.

Messages in graphical interfaces

- The main sender of messages is the computer user:
  - A mouse click using one of the buttons.
  - Mouse movement.
  - Pressing a keyboard key.
  - Etc.
- In response to a user’s action on one of the objects in the graphical interface, that object can send messages to other objects in the interface.
- Let’s look at an example...

Example for a directory application



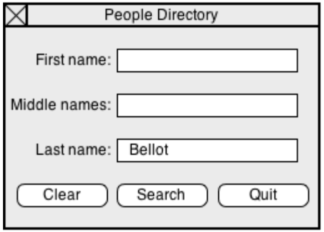
- In this application, the user enters data about a person they are searching for in the main application window.
- Clicking the `Search` button, a window is created to display the results.
- Clicking the `Clear` button resets the input fields.
- Clicking the `Quit` button terminates the application.

Object architecture (1)

- For an object to send a message to another object, it must **know** that other object.
- An object knows another object if it has a **reference** to that other object.
- A reference to an object is an **attribute** that identifies the referenced object.
- The programmer must determine the references between objects: Which object needs to know which other object?
  - Object's Environment.

Object architecture (2)

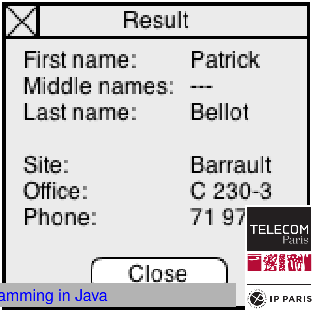
- The main window contains **three** objects of type **input fields**.
- Since it must be able to access them, it needs to **know** these fields.
  - The main window must therefore have three **attributes** that are **references** to the input fields.
- The main window must be able to **create** result windows.
- When the application terminates, it must also **remove** these windows.
  - The main window must therefore have an **attribute** that is **a list of references** to the result windows.


People Directory

First name:

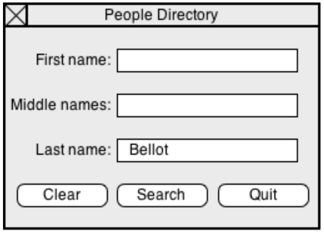
Middle names:

Last name: Bellot

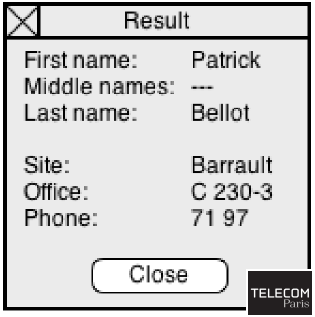

Result

First name:	Patrick
Middle names:	---
Last name:	Bellot
Site:	Barrault
Office:	C 230-3
Phone:	71 97

Object architecture (3)



Bellot

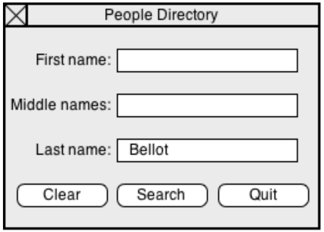


First name:	Patrick
Middle names:	---
Last name:	Bellot
Site:	Barrault
Office:	C 230-3
Phone:	71 97

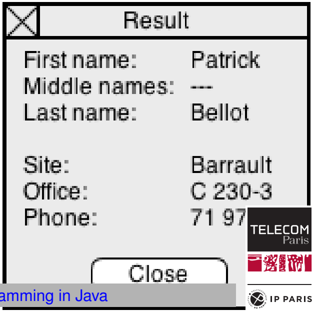
- A result window can be closed by clicking its `Close` button.
- However, we must not forget that the main window maintains **a list of result windows**.
- If a result window is closed via its `Close` button, it must **notify** (send a message to) the main window.
- Each result window must therefore have an **attribute** that is **a reference to the main window**.

Object architecture (4)

- All buttons trigger **actions**:
  - `Clear`, `Search`, and `Quit` in the main window.
  - `Close` in the result windows.
- When the user clicks on one of these buttons, the simplest approach is for the button to **ask** the **window containing it** to perform the task.
  - For example, `Clear` asks the main window to clear **its** fields.
- Therefore, each button must have an **attribute** that is a **reference** to the window containing it.

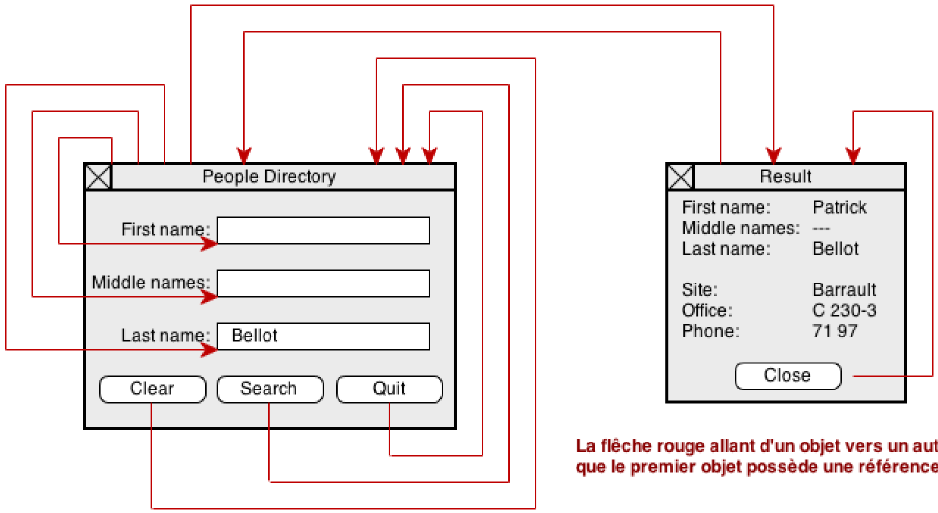

People Directory

First name:	
Middle names:	
Last name:	Bellot


Result

First name:	Patrick
Middle names:	---
Last name:	Bellot
Site:	Barrault
Office:	C 230-3
Phone:	71 97

Visualization of object architecture

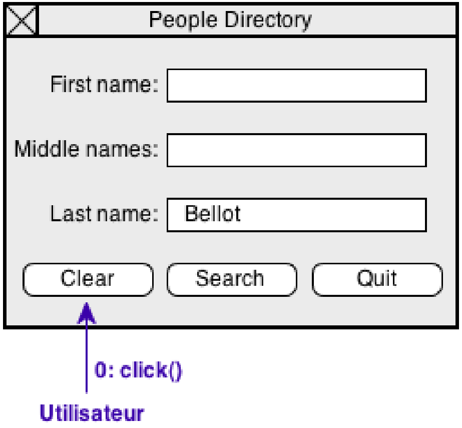


La flèche rouge allant d'un objet vers un autre signifie que le premier objet possède une référence sur le second.

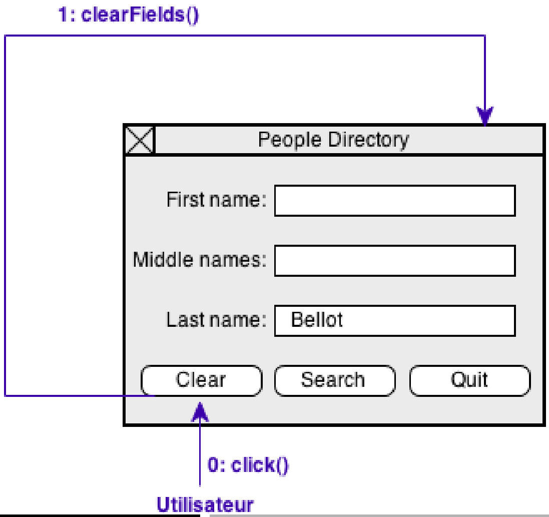
Message cascade

- What happens when the user clicks the `Clear` button?
  - The three input fields need to be **cleared**.
- When the user clicks the `Clear` button:
  - The button receives a `click()` message.
  - The button then sends a `clearFields()` message to the **window containing it**.
  - The window sends a `clear()` message to each of its input fields.
- Upon receiving the `clear()` message, each input field **clears** its text and sends an `ok` message back to the window.
- When the window has received three `ok` messages from the fields, it sends an `ok` message to the `Clear` button.

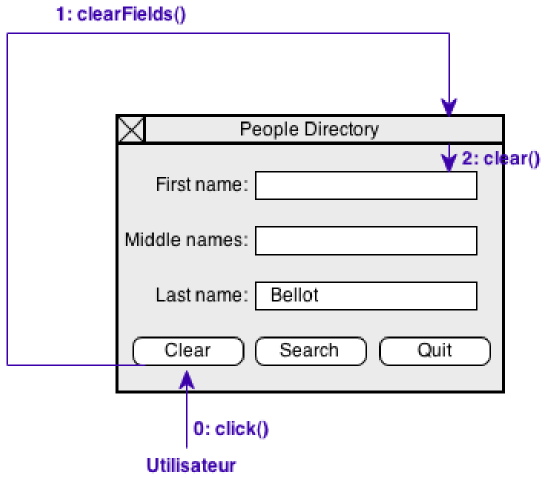
Illustration of messages sent between objects: The `Clear` button



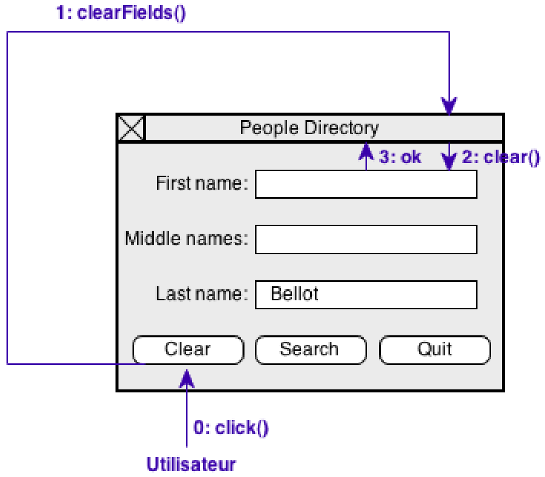
The `Clear` button



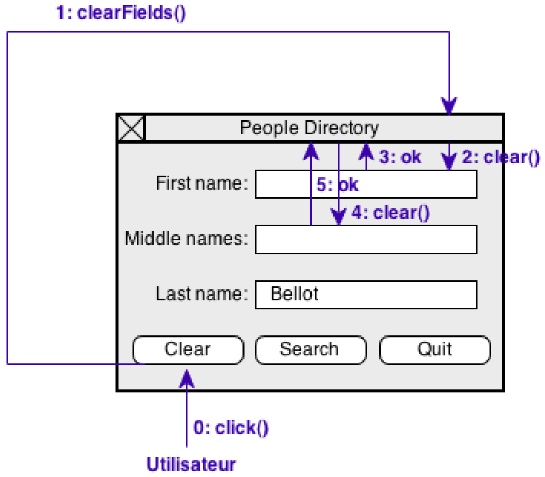
The `Clear` button



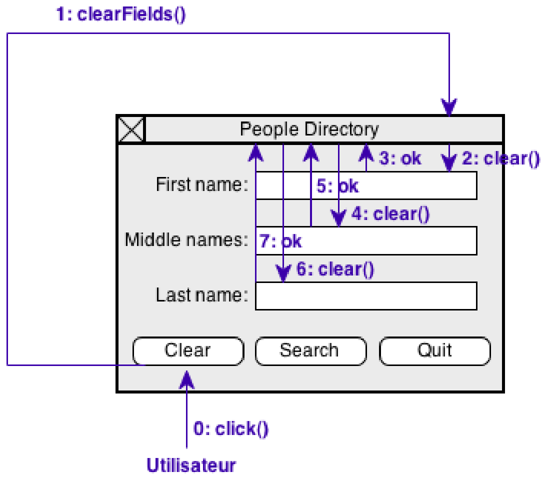
The `Clear` button



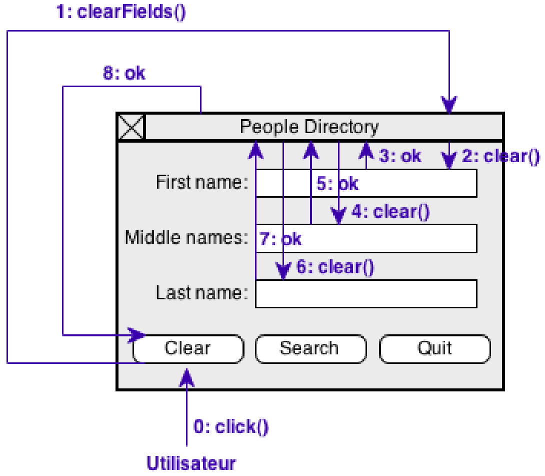
The `Clear` button



The `Clear` button



The `Clear` button



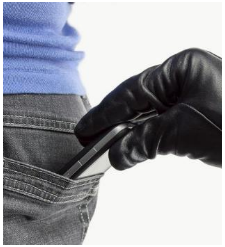
Advantages of OOP

- For each object, it is sufficient to program the **methods** that determine the **responses** to messages received by the objects.
  - These methods are usually very simple.
- Example: the method corresponding to the `click()` message for the `Clear` button:
  - Redraw itself as **pressed**.
  - Send the `clearFields()` message to the main window.
  - Wait for the `ok` response.
  - Redraw itself as **released**.

Advantages of OOP

- Each object has its own **methods** to respond to **messages.**
  - Algorithms are **broken** down into **simpler** parts **distributed** among the objects.
- Each object is **responsible** for interactions with its **environment.**
- This environment consists of objects that the **object knows** and objects that **know the object.**
- The algorithms of each object are easier to design because they are simpler than a **global** algorithm that would try **to manage everything** at once.

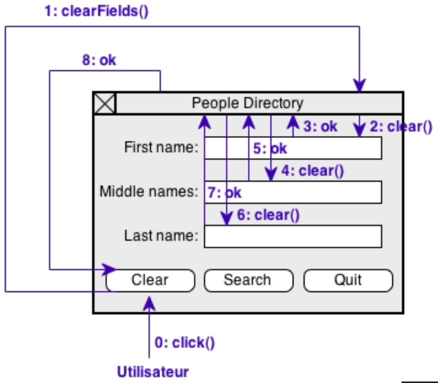
Two important principles of OOP: Non-intrusion and delegation



- **Non-Intrusion principle**: Avoid working on an object **from outside** the object.
- For example, if I need to borrow a phone from someone, I will not take it **directly** from them (**non-intrusive**).
- Instead, I will **ask** them **to lend** me their phone.
  - Send a **message** to the person and **delegate** the task of **lending** me the phone.
- Thus, the person owning the phone can **prepare** it before giving it to me.
  - For example, unlocking it if it's protected by a password.

Example with the directory software

- Thus, the `Clear` button **will not empty** the input fields in the main window **itself**, but will instead **ask** the window to do so.
- Benefit: If a new input field is **added** to the window, only the main window's method needs to be modified, as it **knows** its fields.
- Benefit: We **encapsulate** and **localize** the code within a **limited** number of classes.



Execution flow

- The **execution flow of a program** refers to the sequence of **actions** executed by the program.
- As we will see, these actions can be:
  - Sending messages.
  - Calculating values.
  - Input/output of data.
  - Reading and writing data in memory.
  - Etc.
- These actions are executed **sequentially**.

In summary...

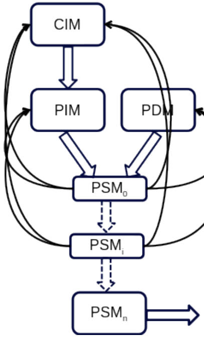
- **Objects** are computing entities that communicate by **sending messages.**
- Objects contain **values** called **attributes.** Among these attributes, there may be **references to other objects.**
  - Object environment; the objects it knows.
- A reference to an object allows sending it a **message.**
- For each type of message an object can receive, the object knows an associated **method** for that message type.
- This method is a **function** or **procedure** (or in C language, a function pointer) that is **executed by the object** when it receives the associated message type.

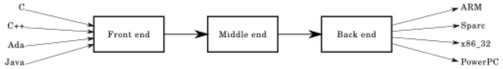
In summary...

- An **object type** is described by a **class**.
  - The class describes the attributes: name and value type.
  - The class describes the methods used to respond to messages.
- The programmer can create objects from the class. This is the process of **instantiation**. Objects are said to be **instances** of the **class**.
- Two main categories of OO programming languages:
  - Class-based languages: Smalltalk, Java, C#, C++, etc.
  - Prototype-based languages: JavaScript, Lua.
- Beyond programming languages, OO is essential for **modelling**.

Modelling instead of programming...

- Design the system with **models** (at a higher level of **abstraction**), **verify** the design models, and **generate the code automatically**!
  - `CIM` (Computation-Independent Model): Describes the system's requirements and how it operates in its environment, **without detailing** the structure of the application or its realization.
  - `PIM` (Platform-Independent Model): Describes the system's details without showing **platform-specific execution details** or particular technology.
  - `PDM` (Platform Description Model): Describes **the execution platform** (processors, memory, buses, etc.).
  - `PSM` (Platform-Specific Model): Describes the software **deployed** on the execution platform.
  - $\rightarrow$: Iterative analysis and modification of design models.
  - $\Longrightarrow$: Model transformation.





- A large portion of modelling languages are **object-oriented**:
  - For example, UML, SysML, AADL, Modelica, etc.