Principes & paradigmes

Eric Lecolinet - Télécom Paris – Institut Polytechnique de Paris
http://www.telecom-paris.fr/~elc

Septembre 2023

Dans ce cours

Organisation du cours

- principaux **paradigmes et principes** des langages informatiques
- **orienté objet** illustré en **C++** (et comparaison **Java**)
- **autres concepts** et compléments
- **programmation événementielle** et interfaces graphiques **Java Swing**

Liens

- **http://www.telecom-paristech.fr/~elc/**
- **http://www.telecom-paristech.fr/~elc/inf224/** - lien vers page **eCampus**

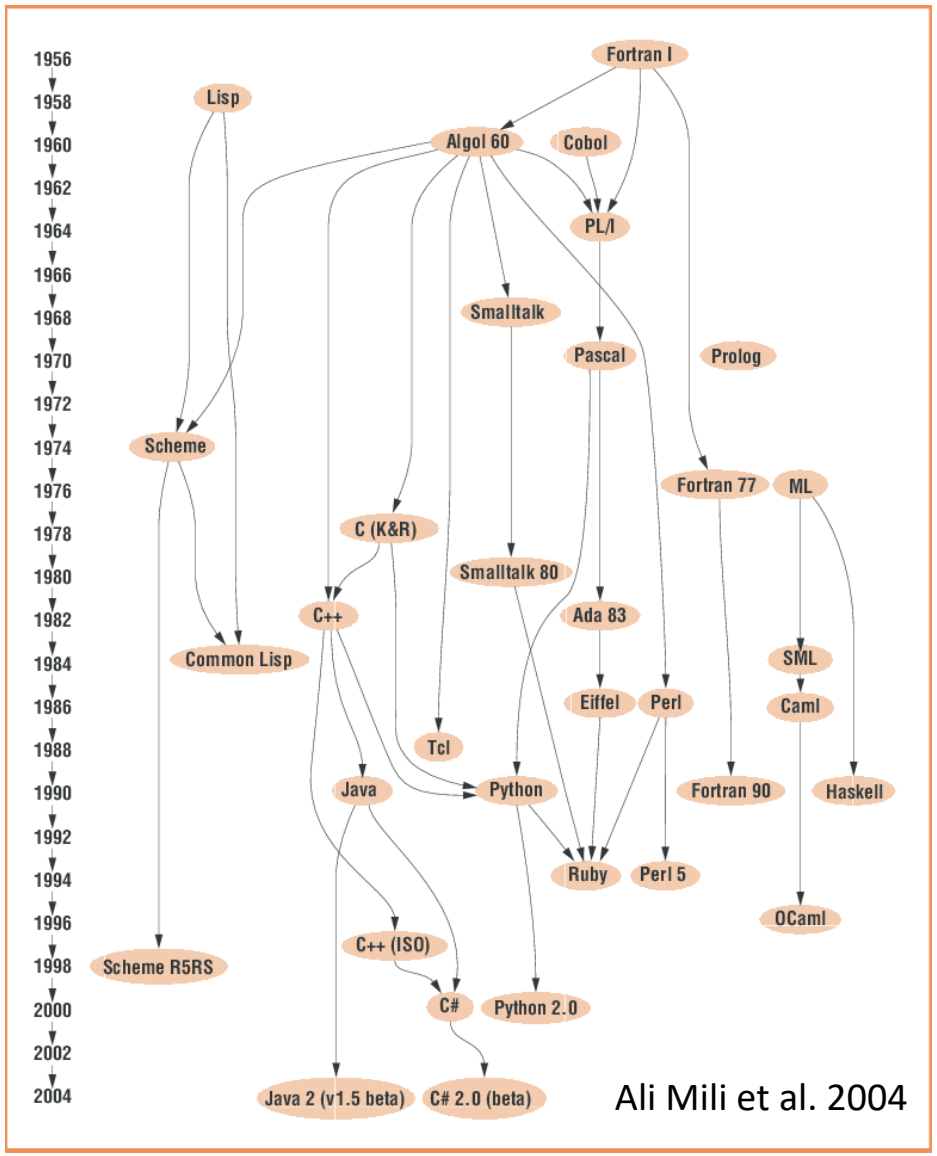
Paradigmes

Paradigme

- approche, philosophie
- la plupart des langages sont **multi-paradigmes**

Principaux paradigmes

- **Impératif**
  - exécution pas à pas des instructions
  - *Fortran, C, Pascal ...*
- **Fonctionnel**
  - évaluation de formules, lambda-calcul
  - *Lisp, Scheme, Caml, Haskell ...*
- **Orienté objet**
  - modélise des interactions entre des entités
  - *Smalltalk, C++, Python, Java, C#, Swift ...*



Paradigmes

Paradigmes (...)

- **Logique**
  - recherche via des règles (Prolog ...)
- **Concurrent**
  - gestion multi-tâches
  - notion de **thread, mutex**, **future,** etc. dans divers langages (C++, Java ...)
  - langages orientés concurrence (Go, Erlang ...)
- **Evénementiel**
  - gérer les réponses à des événements
  - typique des interfaces graphiques, du Web ...
- **Langages spécialisés**
  - pour le **Web** (PHP, Perl, JavaScript, TypeScript ...)
  - langages de **scripts** (bash, zsh ...)
  - langages de **simulation** (circuits)**, réactifs/synchrones** (embarqué), etc.

Paradigme impératif

Principe

- un programme est :
  - une suite d'**instructions**
  - à l'intérieur de **structures de contrôle** emboîtées
- exécution **pas à pas** des instructions

```
int vec[100] ;
...
int somme = 0 ;
for (int i=0;i<100;i++)
    somme += vec[i] ;
```

La gestion de l'exécution incombe au programmeur

Structures de contrôle
de l'exécution :

- **if-then-else**, etc.
- boucles **for, while**, etc.
- appels de **fonctions**
- **exceptions**, etc.

Paradigme impératif

Principe

- un programme est :
  - une suite d'**instructions**
  - à l'intérieur de **structures de contrôle** emboîtées
- exécution **pas à pas** des instructions

```
int vec[100] ;
...
int somme = 0 ;
for (int i=0;i<100;i++)
   somme += vec[i] ;
```

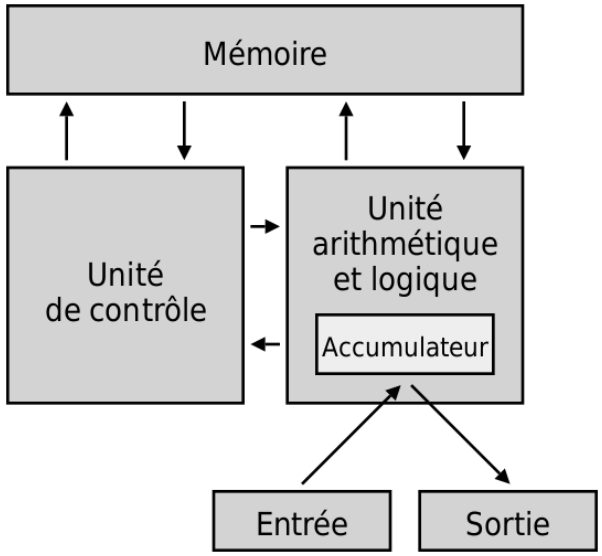
La gestion des données incombe au programmeur

- qui doit :
  - déclarer les **variables** (et souvent leur **type**)
  - decider de leur **durée de vie** (on y reviendra)
  - décider des **effets de bord** (affectation = ce que les variables contiennent)

Paradigme impératif

A l'origine du paradigme impératif

- la machine de **Turing**
  - modèle **abstrait** permettant des calculs arbitraires
- la machine de **von Neumann**
  - architecture **matérielle** d'un ordinateur (1945)
  - instructions numérotées :
    - *arithmétiques*
    - *de transfert de données*
    - *de rupture de séquence (**goto**)*



wikipedia

- l'instruction d'affectation :
  `<variable> := <expression> ;`
- l'instruction de *saut conditionnel* :
  `if <condition> goto <instruction> ;`
  et *inconditionnel* :
  `goto <instruction> ;`

Paradigme impératif

Exemple : programme d'addition

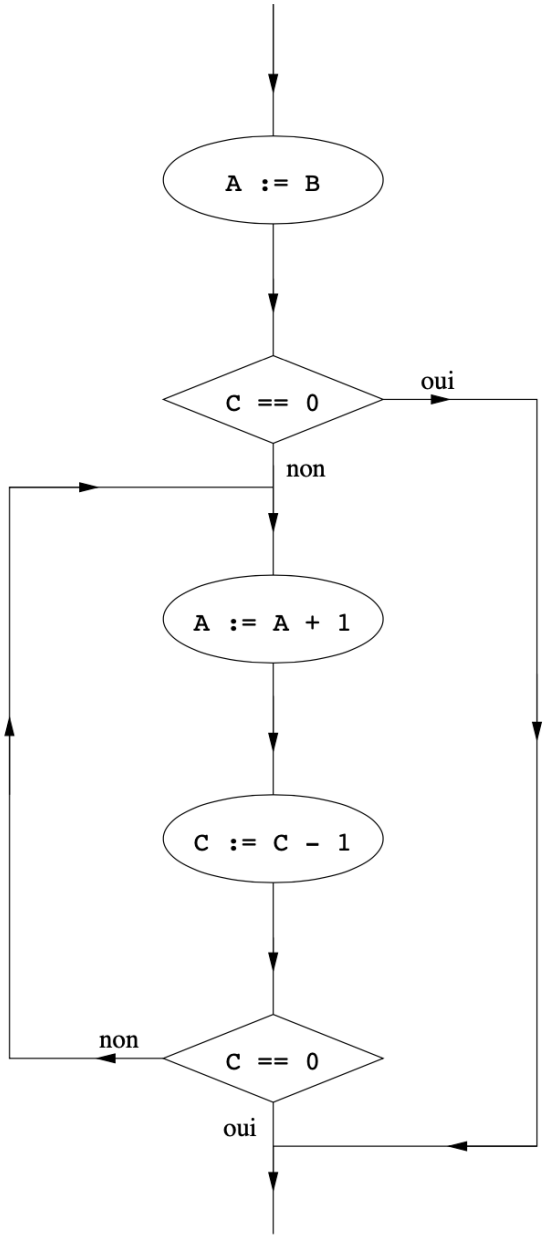
Pré-conditions

- B et C sont des variables contenant des entiers positifs
- on ne sait faire que +1 et -1

Post-conditions

- A vaut B + C ; C vaut 0 ; B inchangée

```
00 : A := B ;
01 : if (C == 0) goto 05 ;
02 : A := A + 1 ;
03 : C := C - 1 ;
04 : if (C <> 0) goto 02 ;
05 : end ;
```



organigramme

Paradigme impératif

Remarque

Pré-conditions

- doivent être vérifiées **avant** le traitement

Post-conditions

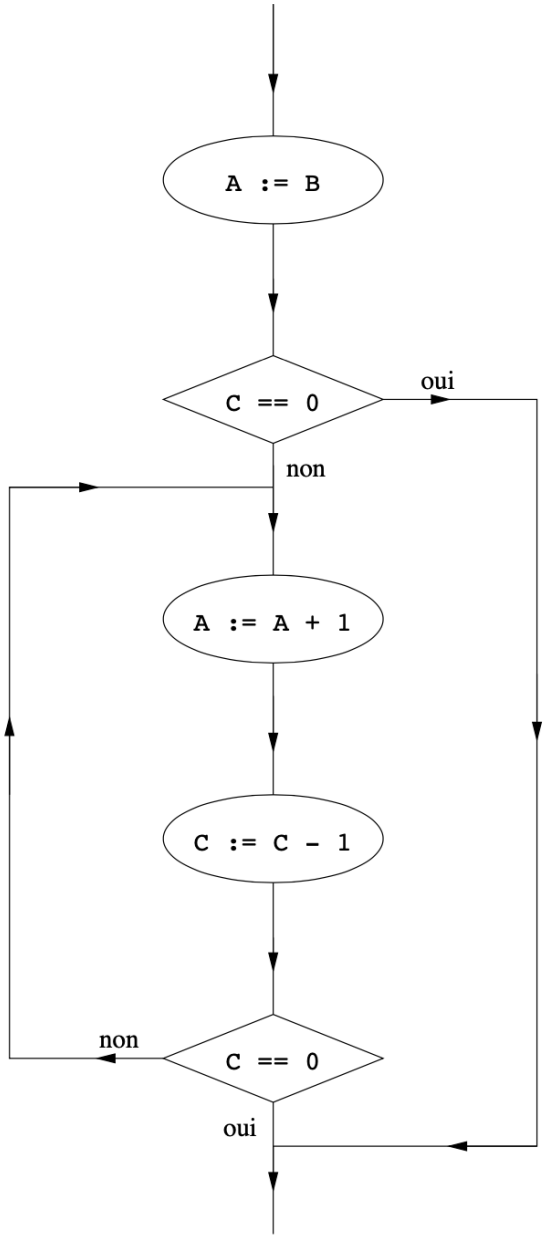
- doivent être garanties **après** le traitement

Invariants

- conditions toujours vraies

Il faut :

- les expliciter (**documentation**)
- si possible les **vérifier par programme**
  - au moins en **phase de développement**



organigramme

Programmation structurée

Théorème de Boehm et Jacopini (en substance)

- toute **fonction calculable** peut être calculée en combinant des sous-programmes via **trois structures de contrôle** :
  - **séquencement**
  - **tests** (if-then)
  - **itérations** (boucles)

Programmation structurée

- dérivée de Boehm et Jacopini
- tout programme peut s'écrire en n'utilisant que **while** et **if-then** (donc sans **goto**)

```
00 : A := B ;
01 : if (C == 0) goto 05 ;
02 : A := A + 1 ;
03 : C := C - 1 ;
04 : if (C <> 0) goto 02 ;
05 : end ;
```

```
A := B ;
while (C <> 0) {
  A := A + 1 ;
  C := C - 1 ;
} ;
```

Programmation structurée

Structure **if-then**

```
if <condition> {
  <instruction 1>
  ...
  <instruction k>
} ;
```

devient :

```
n:      if (not <condition>) goto n+k+1 ;
n+1:    <instruction 1>
        ...
n+k:    <instruction k>
n+k+1:
```

Structure **while**

```
while <condition> do {
  <instruction 1>
  ...
  <instruction k>
} ;
```

devient :

```
n:      if (not <condition>) goto n+k+2 ;
n+1:    <instruction 1>
        ...
n+k:    <instruction k>
n+k+1:  goto n ;
n+k+2:
```

Paradigme fonctionnel

Origine

- dérivé de la théorie du **λ-calcul** (Church, années 30)
- langage **Lisp** (J. McCarthy**,** années 50)

Principe

- repose sur l'**évaluation de fonctions**
- pas d'**effets de bord** (en fonctionnel "pur")
- repose largement sur la **récursivité** (pas de boucles !)

```
(def fact(n)
  (if (= n 0)
    1
    (* n (fact (- n 1)))))
```

```
int fact(int n) {
  return
     (n == 0)
     ?1
     :(n * fact(n-1)) ;}
```

Factorielle en **Lisp** et en **C**

*notation **préfixe** en Lisp, **infixe** en C*

Lisp

Programme = donnée

- fonctions de "**première classe**" (peut être argument)
- une **fonction** est une **donnée** (c'est une liste)

```
? (defun twice(f x) (apply f (apply f x)))
= twice

? (defun z(x) (+ x 1))
= z

? (twice 'z 1)
= 3
```

Lambda = fonction anonyme

- notée **(λx .x + 1)** dans la théorie du **λ-calcul**

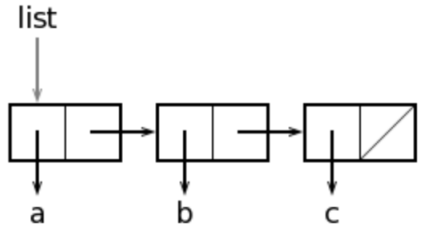
```
? (twice '(lambda (x) (+ x 1)) 1)
= 3
```

- cette notion :
  - a été reprise dans les **langages courants** (C++, Java, Python ...)
  - où elle est liée à la **capture de variables**

Lisp

Tout s'exprime sous forme de listes

- Listes **chaînées** composées de **doublets**
  - **(cons x y)** crée un **doublet**
  - **(car d)** renvoie la composante **gauche**
  - **(cdr d)** renvoie la composante **droite**
  - **nil** ou **()** est la liste **vide**
  - **(null x)** est vrai si x est vide
- Un **programme** est une **liste** !




```
? (defun last(x)
    (if (null (cdr x))
      (car x)
      (last (cdr x))))
= last
? (last '(a b c d))
= d
```

Gestion mémoire automatique

- le programmeur **n'a pas à gérér la mémoire !**
  - **ramasse-miettes** (garbage collector)
  - idée reprise dans Java, Python, etc.

Lisp

Reflexivité

- Lisp est écrit en Lisp (en théorie)
- Lisp est **réflexif**
- (un programme est une liste)

```
(defun toplevel()
  (forever
     (print (eval (read)))))
```

Reflexion

- **1)** **introspection** : un programme peut **examiner** son état
  - ou ses classes : **métaclasses** de Java, C#, Python, JavaScript, ...
  - ex : class **Class** de Java

- 2) un programme peut **modifier** son comportement (self-modifying code)
  - Lisp peut même redéfinir **print** ou **eval** en cours d'exécution !

Paradigme fonctionnel

Lisp a été source d'inspiration

- de **langages fonctionnels** plus récents (Scheme, Caml, Erlang, F# ...)
  - dans le monde **académique** mais aussi l'**industrie** (Erlang = Ericson language)
- des **langages courants** :
  - C++, Java, C#, Python, etc.
  - récursivité, ramasse-miette, lambdas, réflexion, etc.

Avantages / inconvénients du fonctionnel

- **plus robuste**, plus facile à **vérifier** et à **maintenir**
  - du fait de l'absence d'états / d'effets de bord
- généralement **moins performant** que l'impératif (voir plus loin)

Paradigme logique

Programmation logique

- issue de la **démonstration automatique**
- repose sur :
  - une base de **faits**
  - une base de **règles** logiques
  - un **moteur d'inférence**
- forme de **programmation déclarative**

Prolog

- Colmerauer, années 1970, Marseille
- basé sur les **clauses de Horn**

Clause Prolog : `p(x,...) :- q1(x,...), ... , qn(x,...).`

signifie : $p(x,\ldots)$ si $q_1(x,\ldots)$ et $\cdots$ et $q_n(x,\ldots)$
(clause de Horn)

Prolog

Faits :

atomes *terme composé*

```
pere(patrick,jerome).
pere(patrick,helene).
pere(patrick,camille).
pere(patrick,daniel).
mere(marianne,jerome).
mere(marianne,helene).
mere(fadila,camille).
mere(lam,daniel).

couleur(voiture(patrick), bleu).
```

Règles :

variables

```
parent(X,Y) :- pere(X,Y).
parent(X,Y) :- mere(X,Y).

grand-pere(X,Y) :- pere(X,Z), parent(Z,Y).
grand-mere(X,Y) :- mere(X,Z), parent(Z,Y).
```

Buts (queries) :

```
?- pere(patrick,camille).
OK
?- pere(patrick,julie).
NO
```

tester

```
?- pere(patrick,X).
X=jerome, OK
X=helene, OK
X=camille, OK
```

```
?- couleur(voiture(X),bleu).
X=patrick, OK

?- couleur(voiture(X),rose).
NO
```

chercher

Prolog

Méthode de résolution SLDNF

(Selection, Linear, Definite, Negation as failure)

- **Unification**
  - trouver les **valeurs** pour que deux termes soient **identiques**

g(f(X, Y), Z, 2, U) et **g(f(1, 3), Y, T, V)**

=> X = 1, Y = Z = 3, T = 2, U = V

- **Résolution** d'un **but** **p(a,b)** avec une **clause** **p(x,y) :- q1(...), ..., qn(...).**
  - unifier p(a,b) avec p(x,y)
  - si succès, résoudre q1(...), etc. dans l'environnement résultant
  - et ainsi de suite, récursivement

ex : parent(X, camille) avec:

```
parent(X,Y) :- pere(X,Y).
parent(X,Y) :- mere(X,Y).

pere(patrick,camille).
pere(patrick,daniel).
mere(marianne,jerome).
mere(marianne,helene).
mere(fadila,camille).
```

- **Négation par l'échec**
  - ce qui n'est pas vrai est faux

Prolog

En pratique

- la **combinatoire** sous-jacente peut rendre le temps excessif
  - => opérateur de **cut** qui détruit une branche de recherche
- l'ordre des clauses compte !
- peut vite devenir assez complexe ...

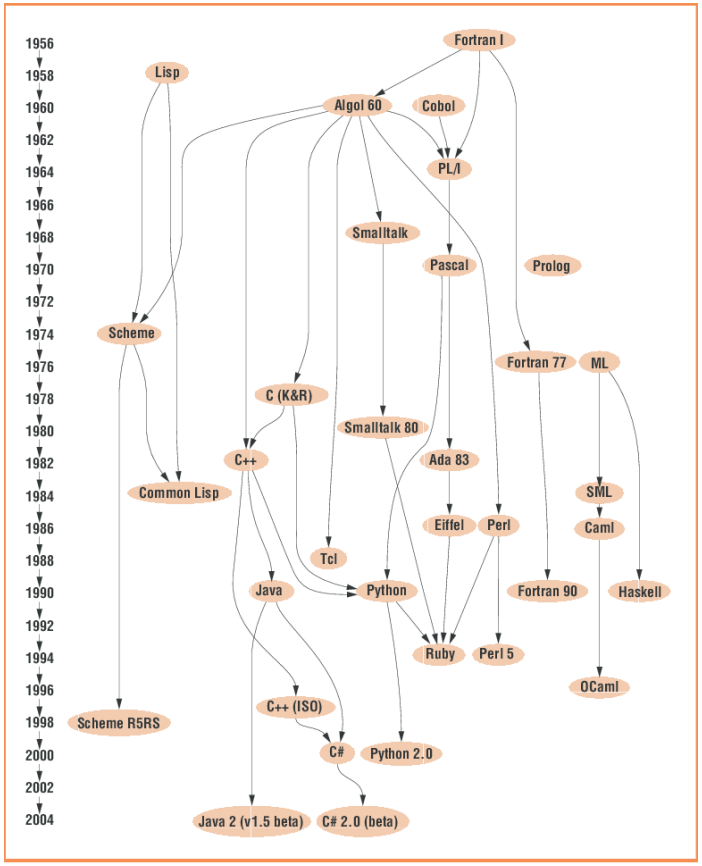
Utilisation, évolutions

- utilisé principalement en **IA** et traitement linguistique
- diverses extensions (équations, arbres infinis, contraintes, etc.)

Paradigme objet

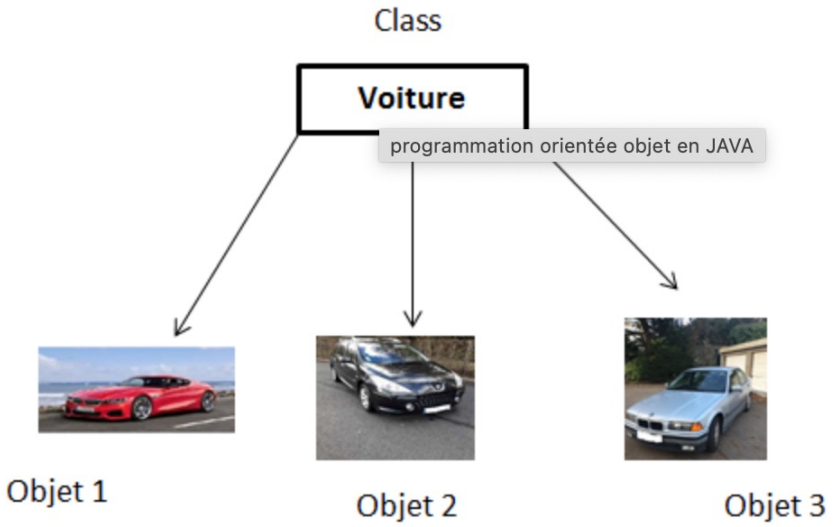
Historique

- **origine** : Smalltalk, programmation structurée
- fort développement dans les années 80
- devient **dominant** dans les années 90
- *ex : C++, Eiffel, Java, C#, Python, Swift ...*



Principe

- modélise des **interactions** entre des **entités** appelées **objets**
- les **objets** sont des instances des **classes**
- sauf pour les langages de **prototypes** :
  - *ou certains **objets** servent de **prototype** aux autres objets (ex : JavaScript)*



Paradigme objet

Point de vue

- modèle "social" d'**interaction** entre **acteurs**

Alice (manager)

- **ne fait pas le rapport** (ce n'est pas son **rôle**)
- **ne "farfouille" pas** dans les affaires de Bob !

Bob (ingé thermique)

- **est responsable** du rapport **thermique** (c'est sa spécialité)

Paradigme objet

L'orienté objet c'est pareil !

L'objet Alice

- **ne fait pas le rapport => envoie un message**
- **ne "farfouille" pas => les données de Bob sont privées**

L'objet Bob

- **est responsable** du rapport **=> décide des actions à exécuter**

Paradigme objet

L'orienté objet c'est pareil !

principes de l'OO qu'on verra plus loin

L'objet Alice

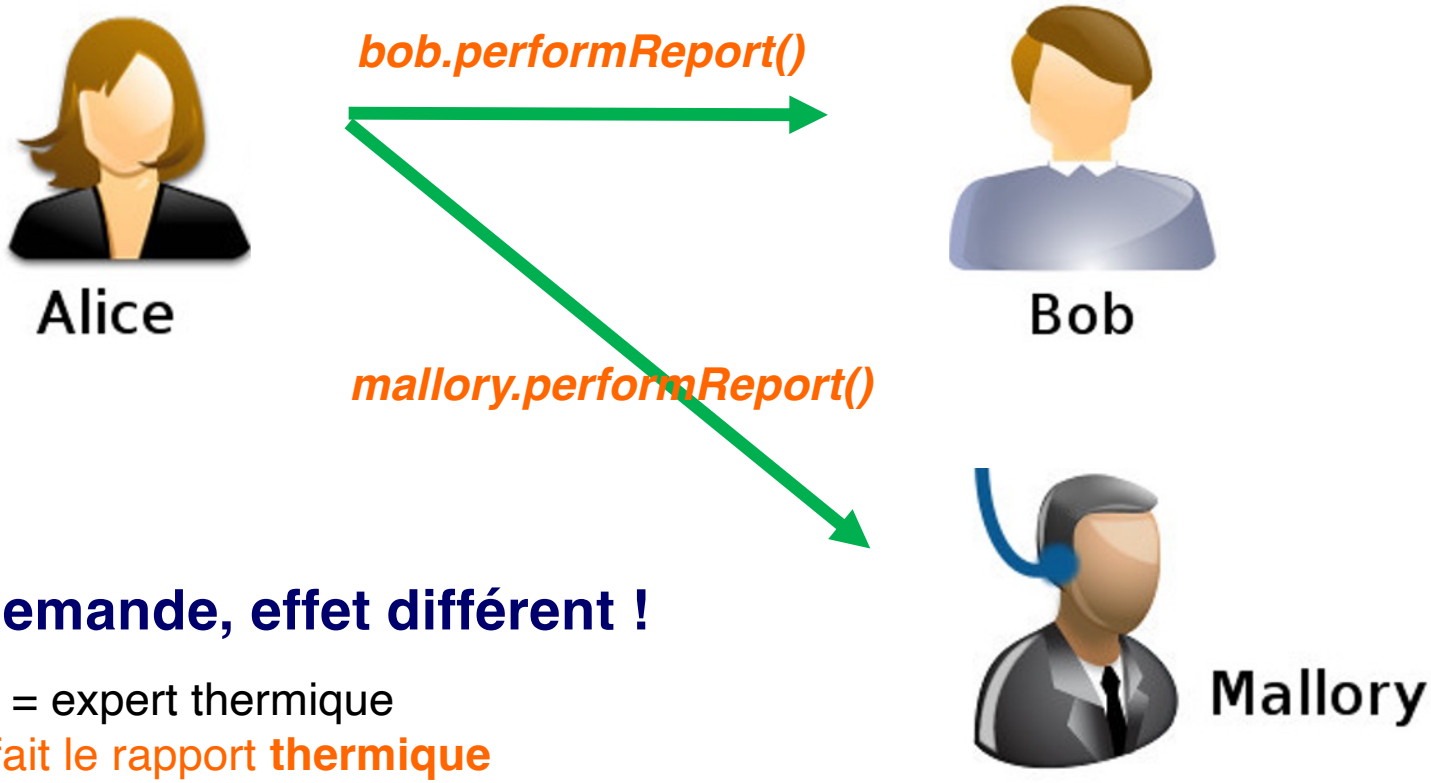
- **ne fait pas le rapport => envoie un message** — appel de méthode
- **ne "farfouille" pas => les données de Bob sont privées** — attributs, encapsulation, droits d'accès

L'objet Bob

- **est responsable => décide des actions à exécuter** — encapsulation, abstraction, polymorphisme

Paradigme objet

Polymorphisme (d'héritage)

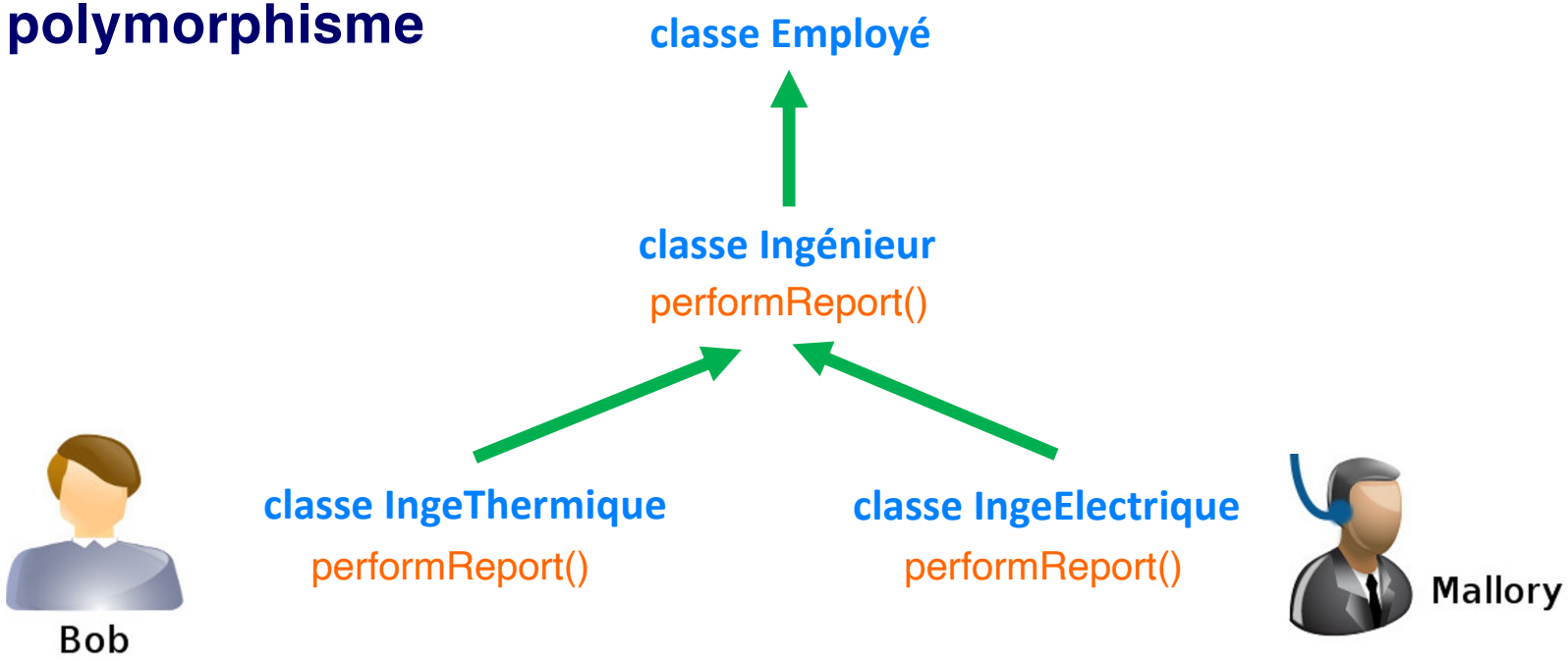


Même demande, effet différent !

- **Bob** = expert thermique
  => fait le rapport **thermique**
- **Mallory** = expert électricité
  => fait le rapport **électrique**

Paradigme objet

Héritage et polymorphisme



Même demande, effet différent !

- **Bob** = instance de **IngeThermique** => performReport() fait le rapport **thermique**
- **Mallory** = instance de **IngeElectrique** => performReport() fait le rapport **électrique**

Compléments, outils

Types

Type

- Définit ce qu'une variable **peut faire**

Typage dynamique

- type déterminé à l'**exécution**
- il peut **changer**

```
a = 1
a = 'toto'
print(a)
```

Python affiche toto

Typage statique

- type déterminé à la **compilation**
- il peut **ne peut pas** changer

```
int i = 1;
i = "toto";
```

Java, C++

ne compile pas !

Inférence de type

- type déterminé **automatiquement**

```
auto i = 1;
```

C++

```
var i = 1;
```

Java

Types

Typage dynamique vs. statique

- typage **dynamique** :
  - plus **pratique**, facilite l'**écriture**
- typage **statique** :
  - plus **fiable**, **moins d'erreurs** à l'exécution (moins de déboguage)

```python
def foo(val):
    if val == 0:
        print(val + 'toto')
    else:
        print(val)
```

Python

OK mais incorrect si *val* vaut 0 !

```python
foo(1)
```

OK

```python
foo(0)
```

Erreur d'exécution !

Polymorphisme

Polymorphisme (= plusieurs formes)

- même **interface** pour différents **types**

Polymorphisme paramétré

- les types sont des **paramètres** d'autres types
- ex : **Generics** de Java, **Templates** de C++

```java
var table = new ArrayList<String>();
```
Java

Polymorphisme d'héritage

- la même méthode a un **effet différent** suivant la sous-classe
  - => spécialisation, abstraction
- NB : souvent appelé "**Polymorphisme**" (sans préciser)

Interprétation vs compilation

Interpréteur

- décode et exécute le programme **au fil de l'eau**
- **plus pratique :** résultat visible immédiatement

```
>>> a = 1
>>> a = 'toto'
>>> print(a)
toto
```

interpréteur Python

Compilateur

- traduit le programme en **code machine**
- le code généré est exécutable **directement** par **la** machine
  - plus **performant,** mais **non portable**

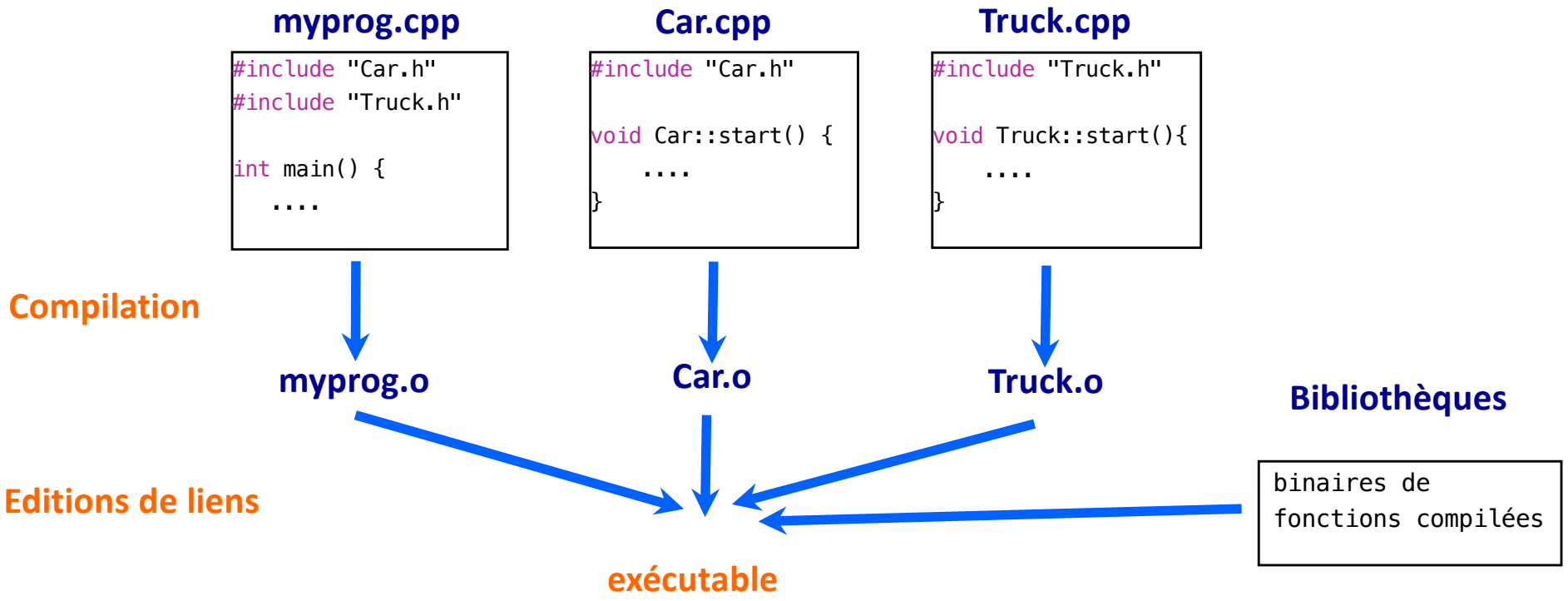
Bytecode

- 1) le programme est **compilé** en **bytecode**
- 2) le bytecode est **interprété** par une **machine virtuelle** (ex : **JVM** pour Java)

Compilation JIT (juste-à-temps)

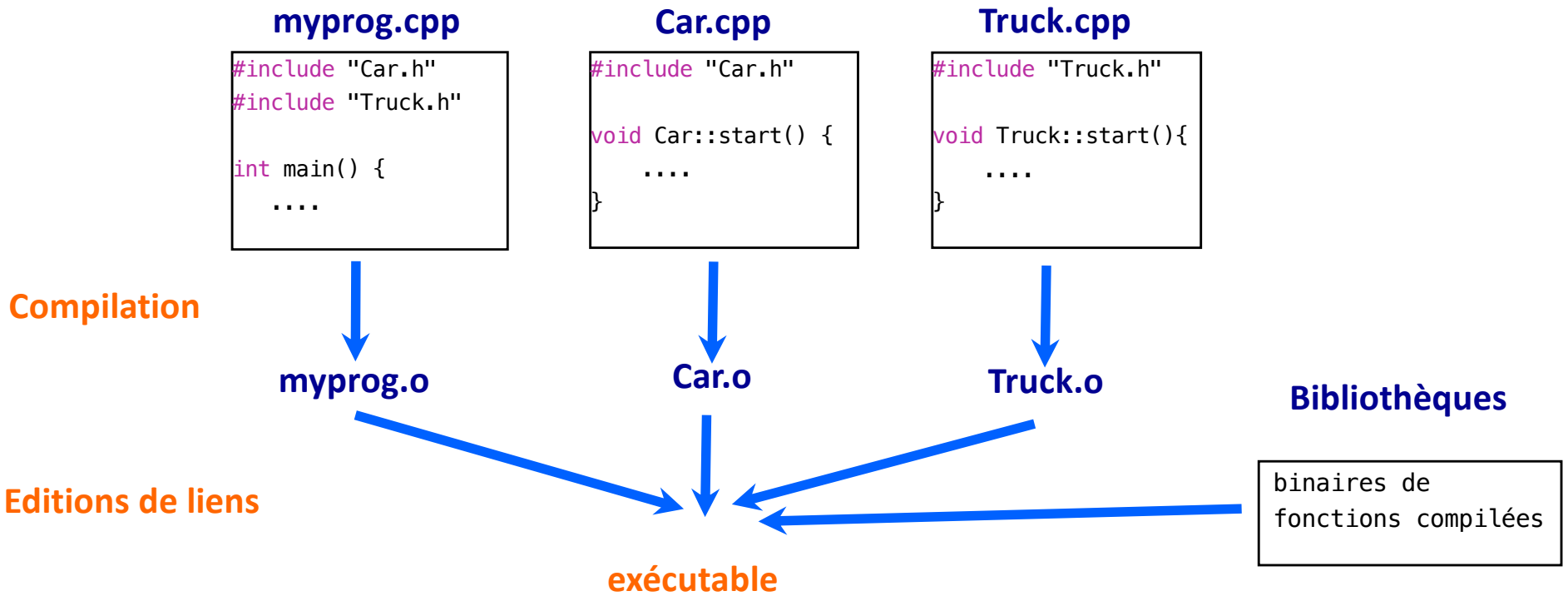
- une partie du bytecode est **compilé** en code machine à l'**exécution**

Compilation séparée



- les fichiers sources sont compilés **indépendamment**
- l'**éditeur de liens (linker)** :
  - resout les **symboles** des fichiers générés
  - les lie aux **bibliothèques**
  - génère un **exécutable**

Librairies statiques et dynamiques



- **Bibliothèques statiques**
  - leur code binaire est **inclus** dans l'exécutable
- **Bibliothèques dynamiques**
  - les liens sont faits à **l'exécution**
  - **avantage :** programmes (beaucoup) moins gros / plus performants

Performance

Critères

- 1. **temps** d'exécution
- 2. consommation **énergétique**
- 3. occupation **mémoire**
- 4. (éventuellement) temps de **compilation**

Performance

	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

	Time
(c) C	1.00
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	1.89
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(c) Fortran	4.20
(v) F#	6.30
(i) JavaScript	6.52
(i) Dart	6.67
(v) Racket	11.27
(i) Hack	26.99
(i) PHP	27.64
(v) Erlang	36.71
(i) Jruby	43.44
(i) TypeScript	46.20
(i) Ruby	59.34
(i) Perl	65.79
(i) Python	71.90
(i) Lua	82.91

	Mb
(c) Pascal	1.00
(c) Go	1.05
(c) C	1.17
(c) Fortran	1.24
(c) C++	1.34
(c) Ada	1.47
(c) Rust	1.54
(v) Lisp	1.92
(c) Haskell	2.45
(i) PHP	2.57
(c) Swift	2.71
(i) Python	2.80
(c) Ocaml	2.82
(v) C#	2.85
(i) Hack	3.34
(v) Racket	3.52
(i) Ruby	3.97
(c) Chapel	4.00
(v) F#	4.25
(i) JavaScript	4.59
(i) TypeScript	4.69
(v) Java	6.01
(i) Perl	6.62
(i) Lua	6.72
(v) Erlang	7.20
(i) Dart	8.64
(i) Jruby	19.84

Ranking Programming Languages by Energy Efficiency

Rui Pereiraa et al.

in *Science of Computer Programming*, Elsevier

Performance et optimisation

Remarque

- on peut **combiner** plusieurs langages pour combiner leurs **avantages** !
  - ex : du **Python** qui appelle des routines **C / C++**

Mode développement et déploiement

- les compilateurs (et les IDEs) offrent **plusieurs modes**
- impactent **énormément** les performances !
  - *cf options –g et –O (ou similaire)*

Profileurs

- affichent le **temps passé** (et autres infos) dans chaque fonction
- indispensables si on souhaite optimiser !

Popularité

TIOBE
the software quality company

About us | Knowledge | News | Coding Standards | TIOBE Index | Contact
Products | Quality Models | Markets | Schedule a demo

Août 2023

Aug 2023	Aug 2022	Change	Programming Language	Ratings	Change
1	1		Python	13.33%	-2.30%
2	2		C	11.41%	-3.35%
3	4	↑	C++	10.63%	+0.49%
4	3	↓	Java	10.33%	-2.14%
5	5		C#	7.04%	+1.64%
6	8	↑	JavaScript	3.29%	+0.89%
7	6	↓	Visual Basic	2.63%	-2.26%
8	9	↑	SQL	1.53%	-0.14%
9	7	↓	Assembly language	1.34%	-1.41%
10	10		PHP	1.27%	-0.09%
11	21	⇈	Scratch	1.22%	+0.63%
12	15	↑	Go	1.16%	+0.20%
13	17	⇈	MATLAB	1.05%	+0.17%
14	18	⇈	Fortran	1.03%	+0.24%
15	31	⇈	COBOL	0.96%	+0.59%
16	16		R	0.92%	+0.01%
17	19	↑	Ruby	0.91%	+0.18%
18	11	⇊	Swift	0.90%	-0.35%
19	22	↑	Rust	0.89%	+0.32%

https://www.tiobe.com/tiobe-index/