PROGRAMMATION ORIENTÉE OBJET

CONTEXTE

La programmation **orientée-objet** est un **paradigme** de programmation (un “style” de conception de programmes) au même titre que

- la programmation impérative (/structurée/procédurale),
- la programmation fonctionnelle,
- la programmation logique,
- …

EXEMPLES

- **impératif:** C, Fortran, Assembleur, …
- **fonctionnel:** Haskell, F#, Reason, Scheme, …
- **objet:** Java, C#, Ruby, …
- **multi-paradigmes:** Scala, C++, OCaml, Python, …

UN UNIQUE PARADIGME OBJET ?

Non ! De **multiples** modèles objets déterminés par

- une collection de traits distincts,
- mais des emphases/variantes significatives,
- et une dimension culturelle/historique forte ...

Pas de consensus universel, mais des caractéristiques communes !

UN PEU D'HISTOIRE

(a propos du système Oberon)

"A lot of the developers and managers at Apple were gathered around watching a presentation from someone about some *wonderful* new product that would save the world. All through the presentation, he had been stating that the product was **object-oriented** while he blathered on."

Finally, someone at the back of the room piped up:

- “So, this product doesn’t support **inheritance**, right?”
- “that’s right”.
- “And it doesn’t support **polymorphism**, right?”
- “that’s right”
- “And it doesn’t support **encapsulation**, right?”
- “that’s correct”.

- “So, it doesn’t seem to me like it’s **object-oriented**”.

To which the presenter huffily responded,

- “Well, who’s to say what’s object-oriented and what’s not?”

At this point the person replied,

- **“I am. I’m Alan Kay and I invented the term.”**

(Source: “He invented the term”)

- Alan Kay, créateur du langage Smalltalk (1972).
- inspiré par le langage Simula (1960s),
  le “premier langage orienté objet”.
- Bjarne Stroustrup (créateur de C++) et James Gosling (créateur de Java) ont également reconnu l’influence majeure de Simula.

Voir: The Early History of Smalltalk

- “I made up the term ‘object-oriented’,
  and I can tell you I didn’t have C++ in mind.”
  Alan Kay, OOPSLA ‘97

  Source: The Forgotten History of OOP

CARACTÉRISTIQUES

“OOP to me means only **messaging**, **local retention and protection and hiding of state-process**, and **extreme late-binding of all things**.”

Alan Kay.

“BESTIAIRE”

Termes fréquents: envoi de messages, encapsulation, attachement dynamique, classes, instances, champs, méthodes, héritage, polymorphisme, composition, délégation, …

ENCAPSULATION

W ENCAPSULATION

- “désigne le principe de **regrouper des données brutes avec un ensemble de routines** permettant de les lire ou de les manipuler.”
- “(…) souvent accompagné du **masquage de ces données brutes** afin de s’assurer que l’utilisateur ne contourne pas l’interface qui lui est destinée.”
- “L’ensemble se considère alors comme une **boîte noire** ayant un comportement et des propriétés spécifiés.”

EXEMPLE: FRACTIONS

Spécification (boîte noire):

```
>>> Fraction(7)
7
>>> Fraction(2, 3)
2/3
>>> Fraction(1, 3) + Fraction(1, 6)
1/2
...
```

Constructeur

```
class Fraction:
    def __init__(self, num, den=1):
        self._num = num
        self._den = den
        self._simplify()
```

Méthode utilitaire

```
def _simplify(self):
    gcd = math.gcd(self._num, self._den)
    self._num = self._num / gcd
    self._den = self._den / gcd
    if self._den < 0:
        self._num = - self._num
        self._denom = - self._denom
```

Méthode d'addition

```
def __add__(self, other):
    num = self._num * other._den + \
          other._num * self._den
    den = self._den * other._den
    return Fraction(num, den)
```

Méthode de représentation

```
def __repr__(self):
    if self._den == 1:
        return f"{self._num}"
    else:
        return f"{self._num}/{self._den}"
```

- Les données des fractions sont stockées dans les **attributs** (ou **champs**) `_num` et `_den`,
- Les **méthodes** `__init__`, `simplify`, `__add__`, ... permettent de les manipuler.

En Python:

- Le caractère privé des données ou méthodes est indiqué par une convention: l'identifiant commence par un caractère de soulignement.
  Seules les méthodes de l'objet devraient accéder au champ `_num` ou appeler la méthode `_simplify`.

- Vous pouvez décider de ne pas vous conformer à cette indication **à vos risque et périls** (“We are all consenting adults”)

Par exemple:

```
>>> f = Fraction(4, 6)
>>> f._num = 7
>>> f
???
```

En Java:

- Les mots-clés

```
public, protected, private
```

contrôlent l’accès aux attributs et méthodes des objets.

- Selon le langage, l’accès aux données peut être rendu possible – de façon controllée – par des **accesseurs** (méthodes) et/ou des **propriétés.**

En Python (lecture seule ou “getter”):

```
def get_numerator(self):
    return self._num
```

et optionnellement:

```
numerator = property(get_numerator)
```

Usage:

```
>>> f = Fraction(2, 3)
>>> f.get_numerator()
2
>>> f.numerator
2
```

ENCAPSULATION – BÉNÉFICES

- **Architecture.** Le logiciel est réalisé par assemblage de composants – plus ou moins autonomes – pour réduire la complexité de l'ensemble.
- **Abstraction.** Ce que fait un objet (son **interface**) est plus important que comment il le fait (son **implémentation**); cette "ignorance sélective" contribue à abaisser la complexité (visible) de chaque composant.

CLASSES

CLASSES → INSTANCES

EXEMPLE – LA CLASSE `Point`

- 2 champs: x et y (valeurs numériques)
- 1 méthode “spéciale”: le constructeur
- 1 méthode “normale”: `distance` (à l’origine)

PYTHON

```
class Point:
    def __init__(self, x, y):
        self.x = x; self.y = y
    def distance(self):
        return math.sqrt(self.x**2 + self.y**2)
```

```
>>> point = Point(1.0, 2.0)
>>> point.distance()
2.23606797749979
```


```
class Point
  def initialize(x, y)
    @x = x; @y = y
  end
  def distance
    Math.sqrt(@x**2 + @y**2)
  end
end
```

```
irb> point = Point.new 1.0, 2.0
=> #<Point @x=1.0, @y=2.0>
irb> point.distance
=> 2.23606797749979
```

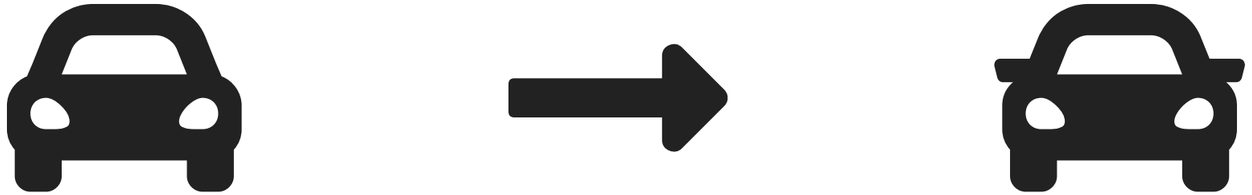
JAVASCRIPT (PROTOTYPE)

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}


Point.prototype.distance = function () {
  return Math.sqrt(this.x**2 + this.y**2);
}
```

```
> point = new Point(1.0, 2.0)
Point { x: 1, y: 2 }
> point.distance()
2.23606797749979
```

PROTOTYPES



Usage: Javascript, Lua.

Voir aussi: Prototypes in JavaScript

JS COFFEESCRIPT

```
class Point
  constructor: (@x, @y) ->


  distance: ->
    Math.sqrt(@x**2 + @y**2)
```

```
coffee> point = new Point 1.0, 2.0
Point { x: 1, y: 2 }
coffee> point.distance()
2.23606797749979
```

JAVASCRIPT (CLASSE)

```
class Point {
  constructor(x, y) {
    this.x = x; this.y = y;
  }
  distance() {
    return Math.sqrt(this.x**2 + this.y**2);
  }
}
```

```
> point = new Point(1.0, 2.0)
Point { x: 1, y: 2 }
> point.distance()
2.23606797749979
```

JAVA

```
public class Point {
  double x, y;
  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
  public double distance() {
    return Math.sqrt(x*x + y*y);
```

AVEC LES OUTILS JAVA

```
class Main {
  public static void main(String[] args) {
    Point point = new Point(1.0, 2.0);
    System.out.println(point);
    double distance = point.distance();
    System.out.println(distance);
  }
}
```

```
$ javac *.java
$ java Main
Point@6fffcba5
2.23606797749979
```

AVEC LE “CODE PAD” DE BLUEJ

```
> Point point = new Point(1.0, 2.0);
> point
<object reference> (Point)
> point.distance()
2.23606797749979 (double)
```

 / 

AVEC JYTHON

```
>>> import Point
>>> point = Point(1.0, 2.0)
>>> point.distance()
2.23606797749979
```

ENVOI DE MESSAGES

Assemblage / Architecture

Les “objets” communiquent par envoi de messages.

OBJET = “ACTEUR”

“(…) considère les **acteurs** comme les seules fonctions primitives nécessaires pour la **programmation concurrente**.
Les acteurs communiquent par échange de messages.
En réponse à un message, un acteur peut effectuer un traitement local, créer d’autres acteurs, ou envoyer d’autres messages.”

W Modèle Acteur

“Actors systems research was based on the assumption that massively parallel, distributed, computer systems could become prevalent, and therefore a convenient and efficient way to structure a computation was as a large number of **self contained processes**, called **actors**, communicating by sending messages to each other.”

Smalltalk wiki

“I realised that Erlang was the only true OO language
– **the big thing about OO is message passing** –
Java/C++ are not OO.”

Joe Armstrong 

See also Why OO Sucks

S’inscrivent dans cette philosophie:

Smalltalk, Erlang, Ruby, Elixir, etc.


```
> 1 + 2
=> 3
```

L’opérateur + calcul la somme des valeurs 1 et 2.


```
> 1.+(2)
=> 3
```

Le calcul est délégué à la méthode + de l’objet 1.

Ruby

```
> 1.send(:+, 2)
=> 3
```

L’addition est la réponse à un message
– contenant le symbole + et l’objet 2 –
adressé à l’objet 1.

RÉFÉRENCES

- Ruby is a Message-Oriented Language
- Do your understand Ruby’s Objects, Message and Blocks?

PAR EXTENSION …

On peut interpréter:

 Java

```
myDictionary.put(key, value);
```

comme

- l’envoi du message `"put"`,
- contenant les données `key` et `value` (*payload*),
- à l’objet `myDictionary`.

HÉRITAGE ET POLYMORPHISME

UN CONCEPT FONDAMENTAL ?

“Unfortunately, **inheritance** – though an incredibly powerful technique – has turned out to be very difficult for novices (and even professionals) to deal with.”

Alan Kay

(Smalltalk-72 n’a pas d’héritage)

What does Alan Kay think about inheritance in object-oriented programming?

LA CLASSE `list`

```
>>> l = [1, 2, 3]
>>> l
[1, 2, 3]
>>> type(l)
<class 'list'>
>>> sum(l)
6
```

MA CLASSE `List` (USAGE)

```
>>> l = List([1, 2, 3])
>>> l
<[1, 2, 3]>
>>> type(l)
<class 'List'>
>>> sum(l)
6
```

- la représentation de ma liste a changé,
- ainsi que son type, `List` et non `list`,
- mais pas le reste des fonctionnalités.

- en héritant de la class `list`, on peut réutiliser ses fonctionnalités,
- on peut également enrichir ou modifier (surcharger) ses comportements.

MA CLASSE `List` (IMPLEMENTATION)

```
class List(list):
    def __repr__(self):
        return "<" + super().__repr__() + ">"
```

POLYMORPHISME

```
def display(item):
    print("L'objet item est:" + repr(item))
```

(`repr` appelle la méthode `__repr__` de `item`)

- le code de `display` ne permet pas de dire quelle implémentation de `__repr__` va être utilisée (**attachement dynamique/tardif**).
- le “contrat moral” est d’utiliser comment argument un objet **représentable**.
- tous les types d’objets respectant cette contrainte peuvent être utilisés comme argument (**polymorphisme**).

- En l’absence de méthode `__repr__` spécifique dans votre classe, Python va se tourner vers les classes dont elle hérite (`object` par défaut).

```
>>> class NoRepr:
...     pass
>>> nr = NoRepr()
>>> nr
<__main__.NoRepr object at 0x7f0a620cb588>
```

```
>>> class List(list):
...     pass
>>> l = List()
>>> l
[]
```

“DUCK TYPING”

(CC BY-SA 3.0, Link)

- L’argument doit passer le test du canard:

  “If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.”

- S’il échoue, une exception se produit (elle peut être gérée par le programme).

EN JAVA

- Pas de “contrat moral” ou “duck typing”,
- Les obligations sont vérifiées par le compilateur,
- Suppose l’usage de **classes** ou d’**interfaces**.

Hériter de – ou **étendre** – `LinkedList`, **une classe**:

```
import java.util.LinkedList;
public class MyList extends LinkedList<Integer> {
  public String toString() {
      return "<" + super.toString() + ">";
  }
}
```

Permet de réutiliser son implémentation.

```
class Main {
  public static void main(String[] arg) {
    MyList list = new MyList();
    list.add(1);
    list.add(2);
    System.out.println(list);
  }
}
```

EXÉCUTION

```
$ java Main
<[1, 2]>
```

“REFACTORING”

```
class Main {
  public static void main(String[] arg) {
    MyList list = new MyList();
    Main.addOneTwo(list);
  }
...
```

```
...
  public static void addOneTwo(MyList list) {
    list.add(1);
    list.add(2);
    System.out.println(list);
  }
}
```

Mais la fonction `addOneTwo` ne peut être utilisée qu'avec les instances de `MyList` (ou qui en dérivent).

Son usage est donc (trop) limité …

ALTERNATIVE – INTERFACES

- La classe `LinkedList` **implémente** de nombreuses interfaces (ou “contrats” vérifiés par le compilateur):

  `Serializable`, `Cloneable`, ..., `Deque`, **`List`**, `Queue`

- En implémentant `List<E>`, la classe `LinkedList<E>` garantit qu’elle implémente la méthode `add`:

```
boolean add(E e)
```

```
import java.util.List;
...
  public static void addOneTwo(List<Integer> list)
{
    list.add(1);
    list.add(2);
    System.out.println(list);
  }
```

POLYMORPHISME

Toutes les classes implémentant `List` sont désormais susceptibles d'utiliser `addOneTwo`:

`MyList`, `LinkedList<Integer>`, `Vector<Integer>`, etc.

BÉNÉFICES DE L'HÉRITAGE

- aggrégation de données et de code,
- réutilisation (sans *modification*) de code existant,
- flexibilité (polymorphisme & attachement tardif).

ALTERNATIVES À L'HÉRITAGE: COMPOSITION

```
class List:
    def __init__(self, items):
        self.l = list(items)
```

C'est **avoir** une liste (et non pas **être** une liste).

DÉLEGATION

On peut “être une liste” sans hériter de `list`:

```
class List:
    def __init__(self, items):
        self.l = list(items)
    def __repr__(self):
        return self.l.__repr__()
    def __iter__(self):
        return self.l.__iter__()
    ...
```