

## Préliminaire

## Introduction : classes et objets

### Programmation Orientée Objet

Jean-Christophe Routier  
Licence mention Informatique  
Université Lille 1



### Objectifs :

- présenter les concepts de base de l'approche objet de la programmation
  - adopter le "penser objet"
  - connaître et savoir mettre en œuvre les concepts fondamentaux
- préparer au cours de **Conception Orientée Objet** du S5

[portail.fil.univ-lille1.fr/ls4/poo](http://portail.fil.univ-lille1.fr/ls4/poo)

Université Lille 1 - Licence Informatique										Université Lille 1 - Licence Informatique									
Programmation Orientée Objet										Programmation Orientée Objet									
introduction	?	objets	classes	synthèse	new	référence	analyse	usage	this	introduction	?	objets	classes	synthèse	new	référence	analyse	usage	this
000000	00000	00000000	000	000000	00	000000	000	0000	000000	000000	000000	00000000	000	000000	00	000000	000	0000	000000

## à l'issue de ce module vous devriez...

## à l'issue de ce module vous devriez...

### ■ ... connaître les éléments de base de la programmation objet

- ... maîtriser le vocabulaire de la programmation objet :  
 ↳ *classe, instance, méthode, interface, attribut, constructeur, encapsulation, polymorphisme, héritage*
- ... savoir décomposer un problème simple en classes et objets
- ... savoir expliquer ce qui différencie la programmation objet des autres paradigmes
- ... savoir expliquer ce qu'est le polymorphisme, en présenter les avantages et savoir expliquer ce qu'est le "late-binding"
- ... connaître le principe ouvert-fermé, être en mesure de l'expliquer et de l'appliquer sur des exemples simples
- ... pouvoir identifier certaines situations de mauvaises conception objet et les corriger
- ... mettre en œuvre l'héritage dans des cas simples
- ... connaître le mécanisme de *lookup*

### ■ ... savoir spécifier, coder et tester un problème objet simple dans le langage JAVA

- ... connaître les principaux éléments de la syntaxe du langage java
- ... être en mesure d'écrire un programme dans le langage java
- ... savoir écrire des tests unitaires simples
- ... pouvoir expliquer clairement le rôle et la sémantique des éléments de langage suivants et savoir les utiliser :  
 ↳ *new, class, interface, public, private, this, static, final, package, import, throws, throw, implements, extends, super*
- ... comprendre le transtypage (upcast/downcast)
- ... être en mesure de choisir une structure de données appropriée et savoir utiliser les types java List, Set, Map et Iterator
- ... savoir gérer les exceptions et connaître la différence entre capture et levée d'exception
- ... savoir utiliser les "outils" liés à la plateforme java :  
 ↳ *javac, java (et classpath), javadoc, jar*

## Programmer

- 1 analyse, étude du problème à modéliser :  
spécifications, écriture de la documentation et des tests
- 2 conception : mise en place des solutions techniques
- 3 codage = écriture du code informatique conforme aux spécifications, validation par les tests

Éléments à considérer :

maintenance – évolution – réutilisation

## Paradigme de programmation

Un **paradigme** de programmation est un style fondamental de programmation qui traite de la manière dont les solutions aux problèmes doivent être formulées dans un langage de programmation.

source Wikipédia

On peut programmer la même chose avec tous les langages. Ils ont tous le même pouvoir d'expressivité (**machines de Turing**)

## Programmation objet

### paradigme objet

Un programme est un ensemble d'objets qui interagissent.

- reprend et prolonge la démarche modulaire : décomposition d'un problème en parties simples,
- (en java) la programmation des traitements reste impérative,
- plus intuitive car s'inspire du monde réel pour une modélisation "plus naturelle"
- facilite la réutilisation
- facilite la conception de "grandes" applications

Quelques langages : Java, C#, Smalltalk, Python, php5, ...

## 4 images 1 mot



Pourquoi ?

même mot ?

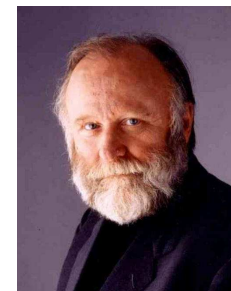
4 images 1 mot



pourquoi ?



Michel Tournier



Frank Herbert



JRR Tolkien



Brandon Sanderson

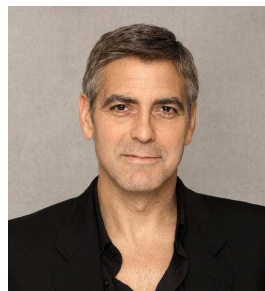
pourquoi ?

même mot ?

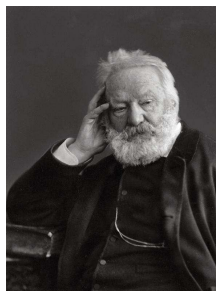
4 images 1 mot



Alan Turing

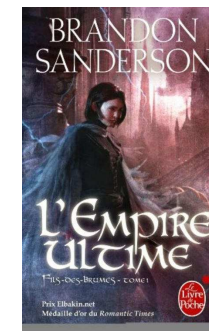
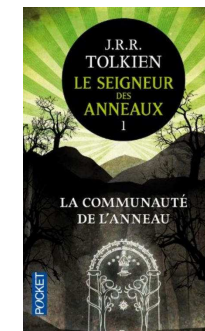
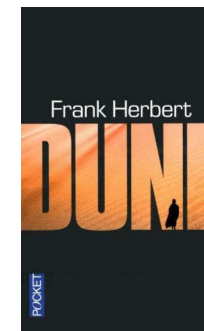
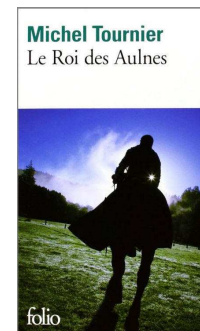


George Clooney



Victor Hugo

pourquoi ?



pourquoi ?

- chaque objet a sa propre mémoire, constituée d'autres objets
- chaque objet a un type
- un programme est un regroupement d'objets qui interagissent par envois de messages
- tous les objets d'un type donné peuvent recevoir les mêmes messages

- objet = identité + état + comportement  
attributs méthodes

Université Lille 1 - Licence Informatique	Programmation Orientée Objet	15	Université Lille 1 - Licence Informatique	Programmation Orientée Objet	16
---	------------------------------	----	---	------------------------------	----

un état

**attributs** (“data member”)

- tous les objets d'un même type ont donc **la même structure** d'attributs mais les valeurs des attributs sont « personnelles » à chaque objet, ces valeurs **sont différentes** d'un objet à l'autre

**méthodes** (“member functions”)

- tous les objets d'un même type peuvent exécuter les mêmes traitements

## classes

langages de classes

- la liste des attributs
- la liste des méthodes et la traitements associés

**classe**

Une **classe** est un type objet.

# instance

Une classe permet de **créer** des objets.  
Ces objets sont du type de cette classe.

instance

On appelle **instance** les objets créés par une classe.  
Tout objet est instance d'une classe.

définir une classe c'est

- 1 définir les méthodes que pourront invoquer ses instances
- 2 définir la structure de l'état de ses instances

# exemplesynthèse

thermomètres, chaudières et thermostats

commentaire

Le traitement de `update()` fait appel aux comportements des objets *boiler* et *thermo*.  
Un objet *thermostat* **envoie des messages** à ces objets pour interagir avec : `thermo.temperatureInCelsius`, `boiler.isOn`, ...

C'est ainsi que se crée la dynamique des programmes

**classe** = modèle

- décrit la structure de l'état (les attributs et leurs types)
- définit les envois de messages acceptés par l'objet (les méthodes)  
⇒ **interface** d'une classe

**instance** = objet conforme au modèle de la classe qui l'a créé

- son état obéit à la structure  
↔ association de valeurs aux attributs
- n'accepte que les envois de messages autorisés par la classe



- la classe définit les envois de message autorisés pour un objet  
temperatureInCelsius() n'a pas de sens pour un objet Boiler
- la classe définit le traitement exécuté suite à un envoi de message  
le message toString() ne déclenche pas les mêmes traitements pour un objet Thermometre et un objet Boiler

**classe** : *abstrait*  
la notion/le type "Chien"  
(personne n'a jamais vu "le type Chien")

**instance** : *concret*  
"ce chien noir que je vois dans la rue",  
"le chien de mon voisin"

**programmation** : définition des classes  $\implies$  abstraction  
**à l'exécution** : travail sur des objets/instances  $\implies$  concrétisation

programmer (objet) c'est écrire des classes  
= écrire des définitions de types  
= définir

- comment sont représentées les données
- comment agissent ces données

## constructeurs

à l'exécution, il faut **créer** les objets

### constructeur

Pour créer un objet il faut utiliser un **constructeur**.  
**Chaque** appel à un constructeur crée un **nouvel** objet (instance) qui obéit au modèle défini par la classe du constructeur.

- un constructeur a deux rôles
  - 1 créer les attributs de l'objet (la structure de l'état)  
 $\implies$  réserver l'espace mémoire
  - 2 donner les valeurs initiales des attributs ("initialiser l'objet")
- chaque classe doit définir comment sont initialisés les attributs  
 $\hookrightarrow$  il peut y avoir plusieurs manières de réaliser cette initialisation

en Java

référence

constructeur en Java

new + nom de la classe (+ param)

exemple :

```
new Thermometer()
new Thermometer(20)
new Author("Tolkien", "JRR", 1892)
```

- en Java, **si** une classe ne définit pas de constructeur, alors il y a un constructeur par défaut (constructeur sans paramètre)  
 ↪ il existe **seulement s'il n'y a pas** d'autre constructeur déclaré

- l'appel à un constructeur a pour résultat une référence vers l'objet créé.
- cette référence = un **pointeur vers l'identité** de l'objet.  
 Elle peut être stockée dans une variable (de type objet).

important

La référence permet d'accéder à l'objet, mais **n'est pas l'objet** lui-même.  
 Une *variable objet* contient l'information pour accéder à l'objet.

cf. télécommande d'un téléviseur

en Java

déclaration variable

Les variables sont typées. Le type d'une variable est fixé à la déclaration.

*Type variableId;*

*variableId* est une référence qui peut pointer des objets de type *Type* (si *Type* est un type objet).

affectation

L'opérateur d'affectation "=" permet d'attribuer une valeur à une variable.

*variableId = expression;*

La valeur de *expression* est affectée à *variableId*.  
 Cette valeur doit être du type de *variableId*.

- une variable **objet** non initialisée vaut **null**
- on peut déclarer et initialiser en même temps
- un objet non référencé est "perdu", on ne peut plus d'adresser à lui  
 ↪ **garbage collector**

```
Thermometer th1;           // th1 vaut null
th1 = new Thermometer(20);  // th1 référence l'objet créé
Thermometer th2 = new Thermometer(25); // déclaration et initialisation

th1 = new Thermometer(18);  // th1 référence un autre objet
                           // l'objet précédent est "perdu"
Thermometer th3 = th1;      // th3 référence le même objet que th1
th1 = new Thermometer();    // th1 référence un nouvel objet
                           // l'objet précédent est toujours référencé par th3
```

rappel : chaque appel à new crée un **nouvel** objet



envoi de message

un **envoi de message** permet d'**invoquer une méthode sur un objet** pour lui envoyer un message il faut une référence vers l'objet

```
reference.message(...)
```

le message doit être autorisé pour le type de la référence

invocation de méthode

une méthode **ne peut pas être utilisée autrement** qu'en étant invoquée sur un objet via un envoi de message à cet objet

l'objet invoquant = le **receveur** du message il fait partie du contexte d'exécution de la méthode

```
Thermometer th1 = new Thermometer(25);
th1.temperatureInCelsius();           // -> 25
th1.changeTemperature(20);
th1.temperatureInCelsius();           // -> 20
th1.temperatureInFahrenheit();        // -> 68
```

la validité du message pour le type de la référence est vérifié à la compilation

Analyse (objet) d'un problème

exemple

- *Quels sont les objets nécessaires à la résolution du problème ?*  
⇒ décomposition du problème en objets
- *A quels modèles ces objets correspondent-ils ?*  
et donc : *Quelles sont les classes ?*
- *Quelles sont les fonctionnalités/opérations dont on veut/doit pouvoir disposer pour les objets de ces classes ?*  
⇒ quel comportement ? càd quels messages doit/veut on pouvoir envoyer aux objets ?
- *Quelle est la structure de l'état des objets ?*  
structure nécessaire à la réalisation des comportements désirés.

- un **catalogue** regroupe des **articles**, il permet de **trouver** un article à partir de sa **référence**
- un **article** est caractérisé par un prix et une référence (une chaîne de caractères pour simplifier) que l'on **peut obtenir**, on veut pouvoir **savoir** si un article est plus cher qu'un autre article donné
- une **commande** est créée pour un **client** et un **catalogue** donnés, on peut **ajouter** des **articles** à une commande à l'aide de sa référence, on souhaite pouvoir **accéder** à la liste des articles commandés ainsi qu'au prix total de ces articles et au coût des frais de port de la commande
- un **client** peut **créer** une **commande** pour un catalogue et **commander** dans cette commande des **articles** à partir de leur référence

Catalogue
...
getItem(ref : String):Item

Item
...
getPrice() : float
getReference() : String
moreExpensiveThan(Item) : boolean

Client
...
createOrder(Catalogue) : Order
orderItem(o :Order, ref : String)

Order
...
Order(Client, Catalogue)
addItem(ref : String)
allItems() : List<Item>
getCatalogue() : Catalogue
getClient() : Client
getTotalPrice() : float
getShippingCost() : float

## Usage

*créer une commande pour un client, faire commander 2 articles par le client, obtenir le prix des articles*  
*on suppose les références*  
*Client client = new Cient(...)*  
*et*  
*Catalogue cata = new Catalogue(...)*  
*disponibles et initialisées*

```

Order order = client.createOrder(cata);
client.orderItem(order,"A0527");
client.orderItem(order,"B3879");
price = order.getTotalPrice();

```

la méthode `orderItem` de `Client` permet d'ajouter un article à une commande à partir de sa référence

quel code pour cette méthode ?

```

public void orderItem(Order order, String reference) {
    // récupérer le catalogue de la commande
    Catalogue cata = order.getCatalogue();
    // récupérer l'article dans le catalogue à partir de la référence
    Item item = cata.getItem(reference);
    // ajouter l'article à la commande
    order.addItem(item);
}

```

envoi de messages

ajouter une méthode qui fournit le coût total d'une commande

où placer la méthode ? dans quelle classe ?

v1 : dans `Client`

```

public float totalCost(Order order) {
    // cumuler les prix de tous les articles
    float total = 0;
    for(Item item : order.allItems()) {
        total = total + item.getPrice();
    }
    // ajouter les frais de port
    return total + order.getShippingCost();
}

```

usage :

```

float cost = client.totalCost(order);

```

ajouter une méthode qui fournit le coût total d'une commande  
où placer la méthode ? dans quelle classe ?

il ne semble pas naturel que ce soit au client de calculer le coût...

v2 : dans Order

```
public float totalCost(???) {
    // cumuler les prix de tous les articles
    float total = 0;
    for(Item item : ????.allItems()) {
        total = total + item.getPrice();
    }
    // ajouter les frais de port
    return total + ????.getShippingCost();
}
```

- quel paramètre ?
- à quel objet envoyer les messages ?

ajouter une méthode qui fournit le coût total d'une commande  
où placer la méthode ? dans quelle classe ?

v2 : dans Order

```
public float totalCost() {
    // cumuler les prix de tous les articles de cette commande
    float total = 0;
    for(Item item : this.allItems()) {
        total = total + item.getPrice();
    }
    // ajouter les frais de port de cette commande
    return total + this.getShippingCost();
}
```

usage :

```
float cost = someOrder.totalCost();
```

## auto-référence

il faut une référence vers l'objet qui a invoqué la méthode totalCost  
=  
le receveur du message "totalCost"

### auto-référence

en Java

**this**

**this** = référence vers l'objet qui invoque la méthode (= le receveur)

**this** est **toujours** défini dans le contexte d'exécution d'une méthode

## createOrder dans Client

*un client peut créer une commande pour un catalogue  
une commande est créée pour un client et un catalogue donnés*

```
public Order createOrder(Catalogue cata) {
    Order theOrder = new Order(this, cata);
    return theOrder;
}
```

code de méthode moreExpensiveThan de Item ?

```
public boolean moreExpensiveThan(Item otherItem) {
    // le prix de cet article est plus grand que celui de otherItem
    return this.getPrice() > otherItem.getPrice();
}
```

ou

```
public boolean moreExpensiveThan(Item otherItem) {
    return this.price > otherItem.price;
}
```

une référence objet permet aussi d'accéder aux attributs

## la classe Item

attributs ? constructeur ?

```
public class Item {
    private float price;
    private String reference;
    public float getPrice() {
        return this.price;
    }
    public float getReference() {
        return this.reference;
    }
    public boolean moreExpensiveThan(Item otherItem) {
        return this.price > otherItem.price;
    }
    public Item(float p, String ref) {
        this.price = p;
        this.reference = ref;
    }
}
```