

# Introduction : classes et objets

## Programmation Orientée Objet

Jean-Christophe Routier  
Licence mention Informatique  
Université Lille 1



# préliminaire

## 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)

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*

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

### ■ ... 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

### paradigme

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.

## langage à objets (purs)

Alan Kay - SmallTalk

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

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

## tout est objet

## type

- 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

### type

un **type** de données définit

l'ensemble des valeurs possibles pour les données du type  
les opérations applicables sur ces données

toute donnée a un type, toute variable a un type

## typage dynamique

en python ou javascript le type d'une variable est défini par sa valeur et le type de la variable peut changer au cours d'une exécution

```
>>> x = 2
>>> x
2
>>> type(x)
<class 'int'>
>>> x = "timoleon"
>>> type(x)
<class 'str'>
```

```
var x = 2;
x;
/* 2 */
typeof(x);
/* number */
x = "timoleon";
typeof(x);
/* string */
```

le typage est **dynamique**

## typage statique

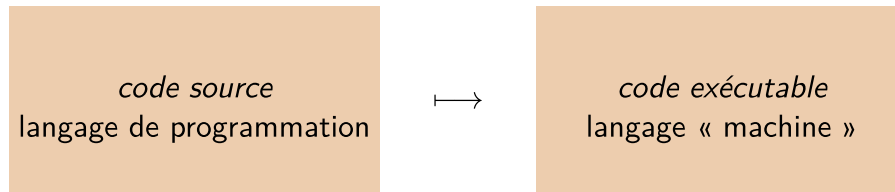
en java, comme en C, le type d'une variable est définie à sa **déclaration**  
il détermine les valeurs que peut prendre la variable et ne peut pas changer  
le compilateur vérifie le typage

```
int x = 2;
x; /* 2 */

x = "timoleon"; // interdit ! ne compile pas
```

le typage est **statique**

## compilation



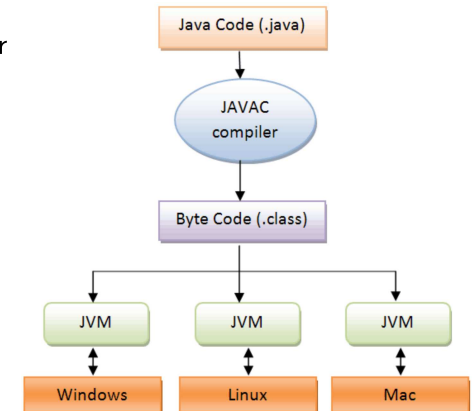
plusieurs phases

- 1 analyse lexicale
- 2 **analyse syntaxique** : vérifie la syntaxe
- 3 **analyse sémantique** : vérifie le typage
- 4 optimisation de code

en C la compilation est suivie de l'édition de liens : **early-binding**  
 pas en Java : **late-binding** voir plus tard dans le cours

## machine virtuelle

- le compilateur C génère un code en *langage machine binaire* exécutable par une machine hôte donnée
- le compilateur java génère du **bytecode java**  
 = *langage machine virtuel*
- le code java s'exécute dans une **Java Virtual Machine (JVM)**



la JVM interprète le bytecode pour l'exécuter  
 la JVM rend le code java indépendant de l'OS et de la machine hôte

« *compile once, run everywhere* »

## tout est objet

- 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

## synthèse

- 1 un objet est composé de données et peut exécuter des traitements
  - 2 un objet a un type
  - 3 un type définit
    - l'ensemble des valeurs possibles
    - les opérations applicables
- le type d'un objet définit
- les données qui composent cet objet = **les attributs**
  - les traitements que peut exécuter cet objet = **les méthodes**

# classes

langages de classes

## classe

une **classe** est un type objet

une classe définit

- la liste des **méthodes** et les traitements associés  
↪ le **comportement** des objets
- la liste des **attributs** nécessaires à la réalisation des traitements  
↪ l'**état** des objets

le **comportement** agit sur l'**état** et l'**état** influence le **comportement**

Book
author : Author
title : String
publicationYear : int
text : String
print()
getTitle() : String
getAuthor() : Author

Author
name : String
firstname : String
birthYear : int
deathYear : int
getName() : String
getDeathYear() : int
setDeathYear(death : int)
toString() : String



## instance

une classe permet de **créer** des objets  
ces objets sont du type de cette classe

## instance

on appelle **instance** un objet créé par une classe  
tout objet est instance d'une classe

## méthodes et attributs

### méthode

une **méthode** est une fonction qui appartient à une classe  
« *member function* »

une méthode ne peut être utilisée que par les instances de la classe

### attribut

un **attribut** est une variable qui appartient à une classe  
« *data member* »

chaque attribut a un type

## définir une classe c'est

### 1 définir les méthodes que pourront exécuter ses instances

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

chaque traitement réalisé par le programme est défini dans une méthode

### 2 définir les attributs de ses instances

tous les objets d'un même type ont donc **la même structure d'attributs**

mais les valeurs des attributs sont « personnelles » à chaque objet, elles **sont différentes** d'un objet à l'autre

les données du programme sont mémorisées dans les attributs des objets

```
public class Book {
    // les attributs de la classe Book
    private Author author;
    private String title;
    private int publicationYear;
    private String text;
    // constructeur
    public Book(Author someAuthor, String title, int pubYear, String text) {
        this.author = someAuthor;
        this.title = title;
        this.publicationYear = pubYear;
        this.text = text;
    }
    // les méthodes de la classe Book
    public void print() {
        System.out.println(this.text);
    }
    public Author getAuthor() {
        return this.author;
    }
    public void getTitle() {
        return this.title;
    }
}
```

Université Lille 1 - Licence Informatique									Université Lille 1 - Licence Informatique								
Programmation Orientée Objet									Programmation Orientée Objet								
introduction ○○○○○○	langages ○○○○○○○○○○	classes ○○○○○○○	synthèse ●○○○○○	new ○○	référence ○○○○○	analyse ○○○	usage ○○○○	this ○○○○○○	introduction ○○○○○○	langages ○○○○○○○○○○	classes ○○○○○○○	synthèse ●○○○○○	new ○○	référence ○○○○○	analyse ○○○	usage ○○○○	this ○○○○○○
exemple									synthèse								

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 possibles (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 messages autorisés par la classe  
= n'exécute que les méthodes définies 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 Thermometer 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"  
"Idéfix"

**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 sont manipulées 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

**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

invocation de méthode

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

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

- *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.

exemple

- 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, 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(item : Item)
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 Client(...)
```

et

```
Catalogue cata = new Catalogue(...)
```

disponibles et initialisées

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

Université Lille 1 - Licence Informatique	Programmation Orientée Objet							37	Université Lille 1 - Licence Informatique	Programmation Orientée Objet							38
introduction	langages	classes	synthèse	new	référence	analyse	usage	this	introduction	langages	classes	synthèse	new	référence	analyse	usage	this
000000	0000000000	000000	00000	00	00000	000	00●00	000000	000000	0000000000	000000	00000	00	00000	000	00●0	000000

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

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

quel code pour cette méthode ?

où placer la méthode ? dans quelle classe ?

```
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);
}
```

envois de messages

v1 : dans Client

```
public float totalCost(Order order) {
    // récupérer le prix de tous les articles
    float total = order.getTotalPrice();
    // ajouter les frais de port à celui des articles
    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(???) {
    // récupérer le prix de tous les articles
    float total = ????.getTotalPrice();
    // ajouter les frais de port
    return total + ????.getShippingCost();
}
```

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

## 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

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

où placer la méthode ? dans quelle classe ?

v2 : dans Order

```
public float totalCost() {
    // récupérer le prix de tous les articles de cette commande
    float total = this.getTotalPrice();
    // ajouter les frais de port de cette commande
    return total + this.getShippingCost();
}
```

usage :

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

## getTotalPrice dans Order

```
public float getTotalPrice() {
    float total = 0;
    // cumuler les prix de tous les articles
    for(Item item : this.allItems()) {
        total = total + item.getPrice();
    }
    return total;
}
```

## 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;
    }
}
```