



Programmation Orientée Objet en Java

3IRC – 4ETI
Version 2013/2014 – Release 2015-2016

F. PERRIN

membre de UNIVERSITÉ DE LYON

LYON
CPE
ÉCOLE SUPÉRIEURE
DE CHIMIE PHYSIQUE ÉLECTRONIQUE
DE LYON

Remerciements



- Merci à Taghrid ASFOUR, Martine BREDA, Fabrice JUMEL, Xavier TROUILLOT, pour leurs supports, conseils, exemples, schémas et relecture.
- Merci à Bruno MASCRET pour ses conseils en pédagogie active et ludique.
- Merci à Kathy SIERRA et Bert BATES pour leur ouvrage *Java - Tête la Première* (O'Reilly edition) dont je me suis largement inspirée.
- Merci à tous les photographes et artistes dont j'ai empruntés les œuvres.

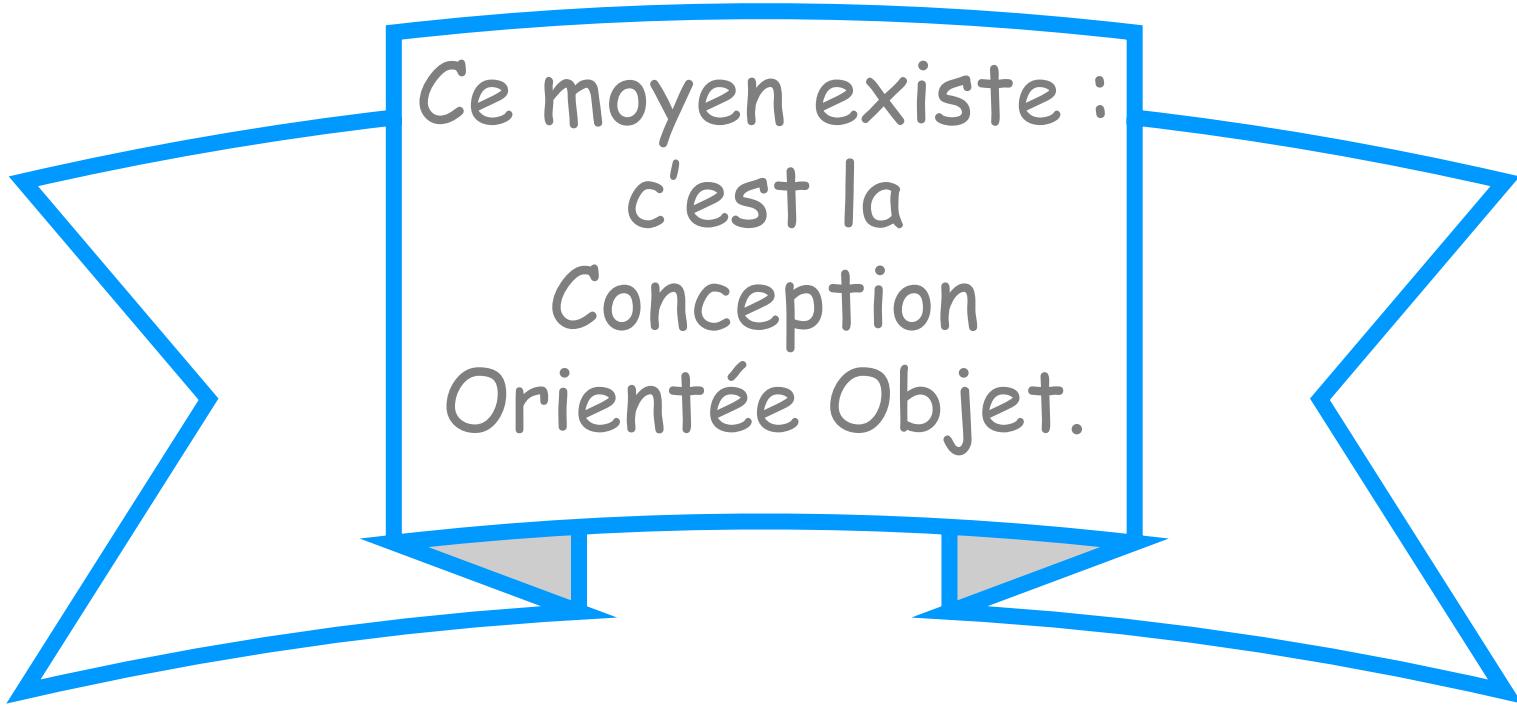


I have a dream today (1)

Ne serait-ce pas merveilleux s'il existait un moyen de programmer des logiciels souples, extensibles, facile à maintenir, réutilisables et efficaces ?
Mais ce n'est probablement qu'un rêve...



I have a dream today (2)



Ce moyen existe :
c'est la
Conception
Orientée Objet.

Qu'allez-vous trouver dans ce document ?



- Des **rêves** qui introduisent les sous-chapitres.
- Des **exemples** en Java SE accompagnés de **schémas** ou **d'images**.
- Qui illustrent des **concept**s.
- Complétés par des **interviews** et **face à face** d'éléments du langage (autre document).
- Des **exercices** (autre document) souvent ludiques : programme à trou, vrai ou faux, qui suis-je, mots croisés, etc.
- Ce qu'il faut **retenir**.



Comment allez-vous monter en compétence sans souffrir ?

- N'hésitez pas à **rêver** et à laisser parler vos **émotions** (surprise, curiosité, amusement, sensation de toute-puissance, etc.).
- **Sollicitez activement vos neurones** pour résoudre des problèmes, tirer des conclusions, etc.
- Voyez les exercices comme des **défis** et savourez la satisfaction de les avoir relevés.
- Plongez dans les TP pour **capitaliser définitivement les compétences**.



Comment sera évaluée l'acquisition de vos compétences

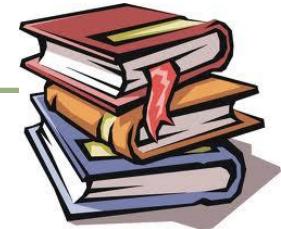
- 1 DS de 2h avant le projet (40%).
- 1 DS de 2h en fin de module pour évaluer l'ensemble des compétences acquises (40%).
- Les TP/Projet ne sont pas notés puisqu'ils doivent permettre d'acquérir des compétences. Auto-évaluation du niveau sera faite par les élèves et confirmée par les enseignants (20%).



La bibliographie que je vous recommande

Sites Web d'approche facile

- <http://jmdoudoux.developpez.com/cours/developpons/java/>
- <https://zestedesavoir.com/>



Documentation de référence du langage:

- Java SE Technical Documentation :
<http://docs.oracle.com/javase/7/docs/api/>
- Tutoriaux : <http://docs.oracle.com/javase/tutorial/>

Livres de référence :

- *Java -Tête la Première*, Kathy Sierra Bert Bates, 2005, O'Reilly.
- *Programmer en java – Java 5 à 7*, Claude Delannoy, 2012, Eyrolles.



Chapitre 1 : Génie Logiciel Conception Orientée Objet

3IRC – 4ETI
Version 2013/2014

F. PERRIN

membre de UNIVERSITÉ DE LYON



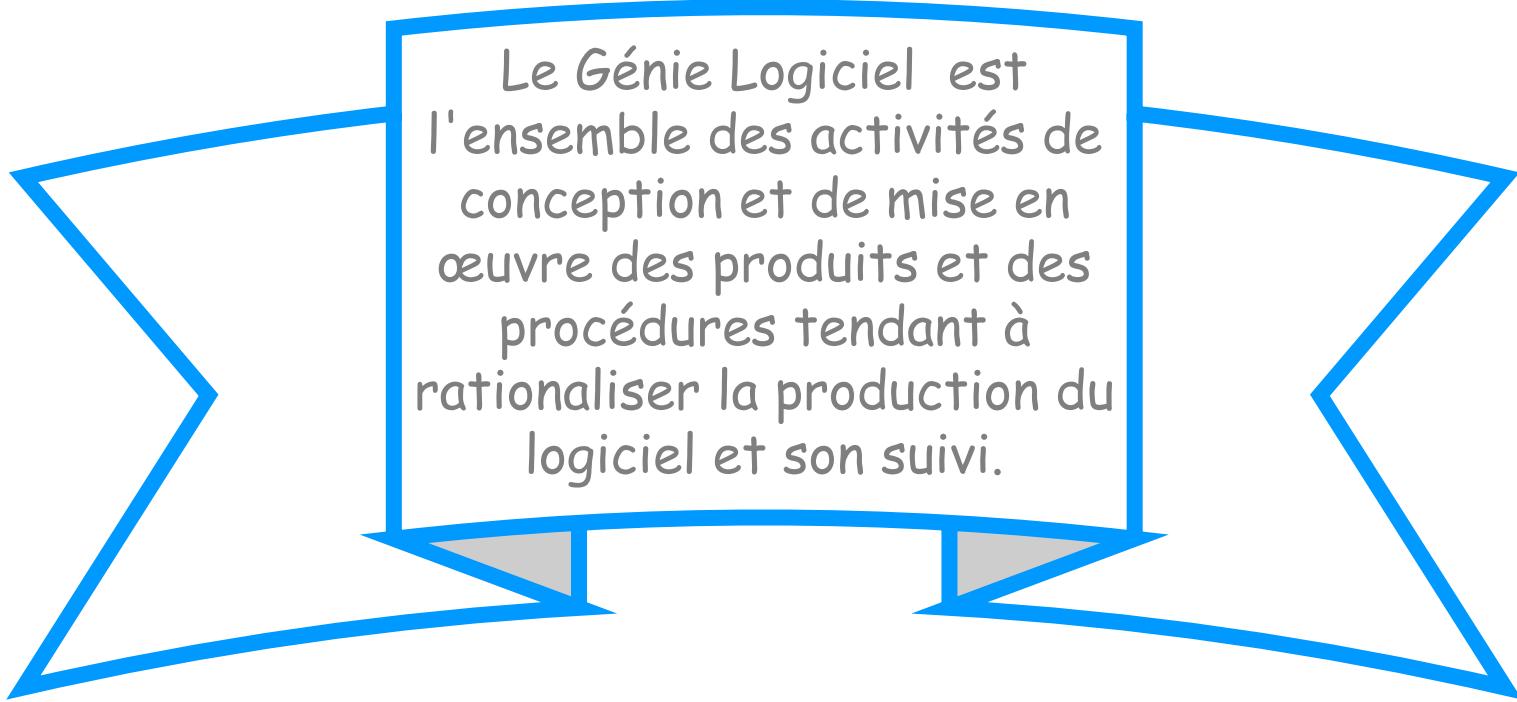
ÉCOLE SUPÉRIEURE
DE CHIMIE PHYSIQUE ÉLECTRONIQUE
DE LYON

I have a dream today (1)

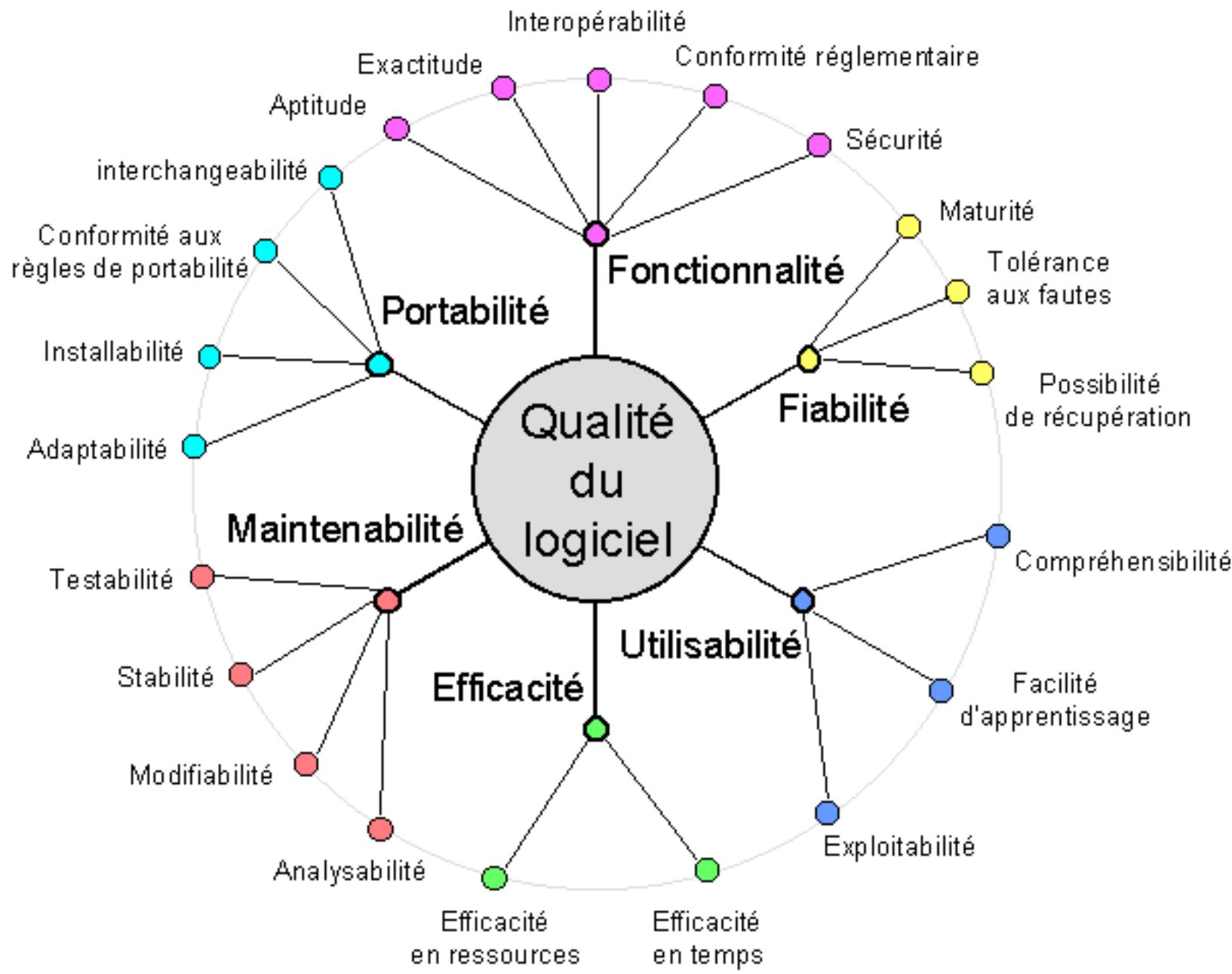
Ne serait-ce pas merveilleux s'il existait des procédures qui permettent d'arriver à ce que des logiciels de grande taille correspondent aux attentes du client, soient fiables, aient un coût d'entretien réduit et de bonnes performances tout en respectant les délais et les coûts de construction ?
Mais ce n'est probablement qu'un rêve...



I have a dream today (2)



Le Génie Logiciel est l'ensemble des activités de conception et de mise en œuvre des produits et des procédures tendant à rationaliser la production du logiciel et son suivi.

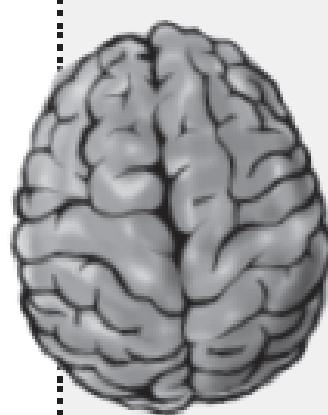


Qu'est-ce qu'un bon logiciel ?



Conquérir vos clients

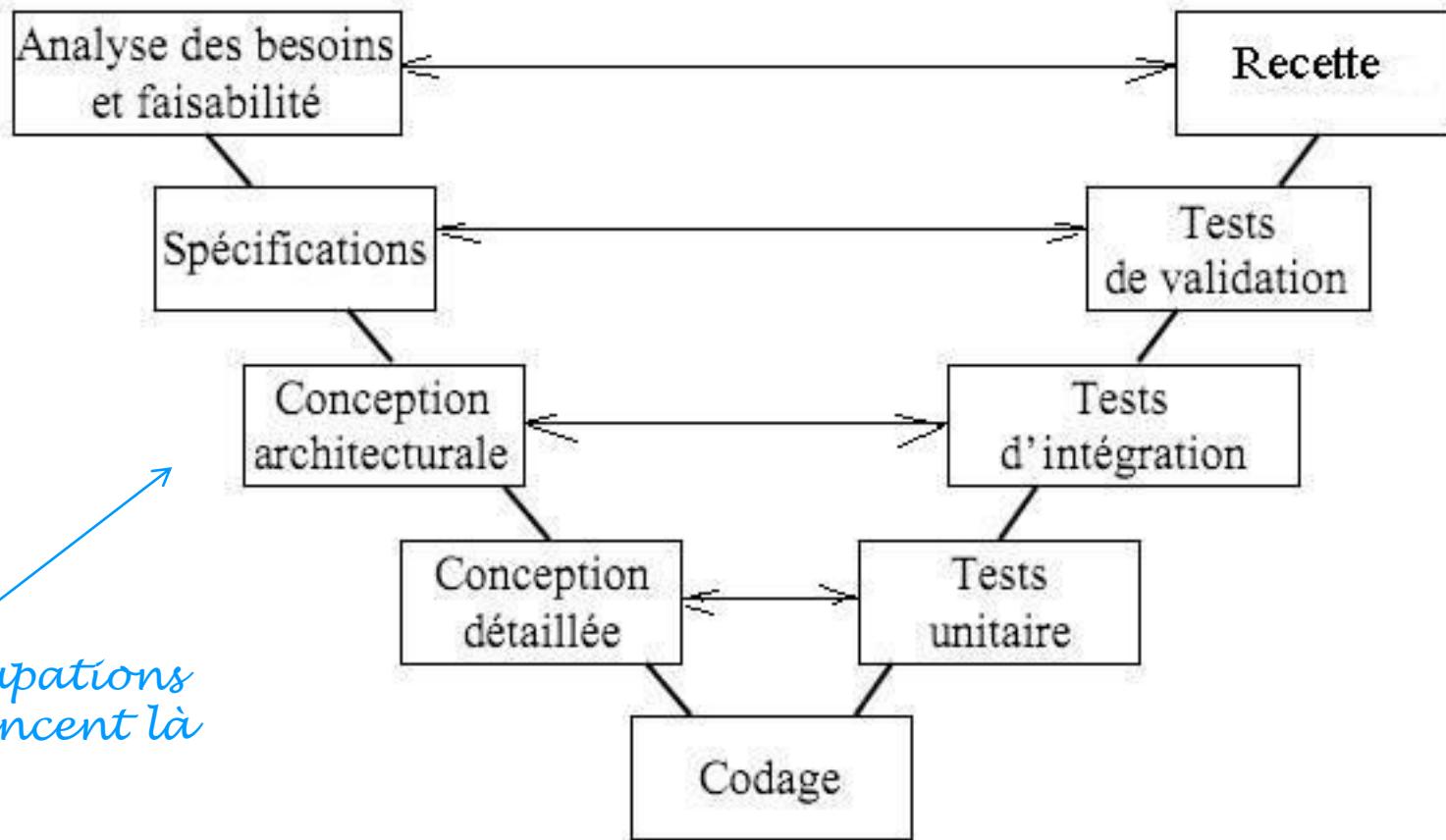
Les clients penseront que votre logiciel est formidable s'il fait ce qu'il est censé faire.



Créer un code aussi intelligent que vous.

Vous (et vos collègues) penserez que votre logiciel est formidable s'il est facile à maintenir, réutiliser et étendre.

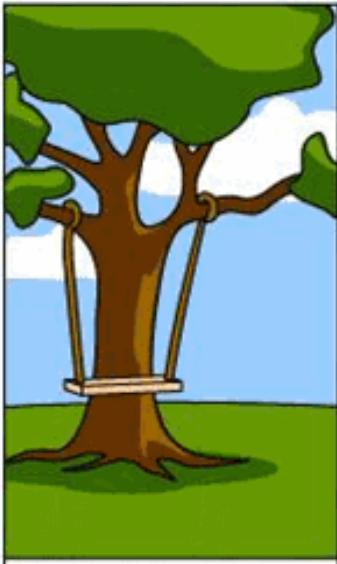
Cycle de vie d'un logiciel



Nos
préoccupations
commencent là



Comment le client
l'a expliqué.



Comment le chef de projet
l'a compris.



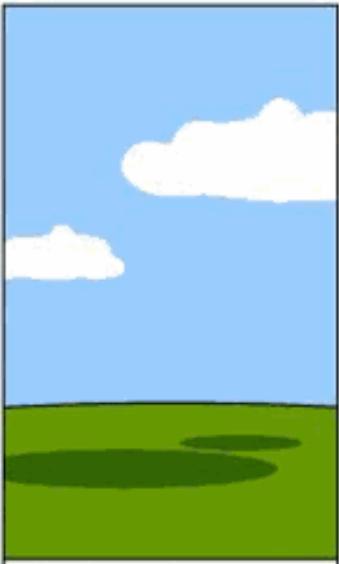
Comment l'analyste
l'a conçu.



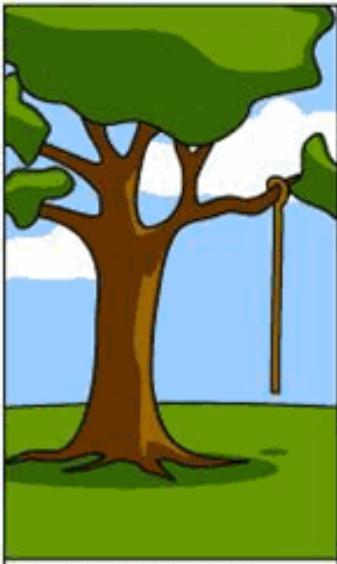
Comment le programmeur
l'a codé.



Comment le consultant
business l'a décrit.



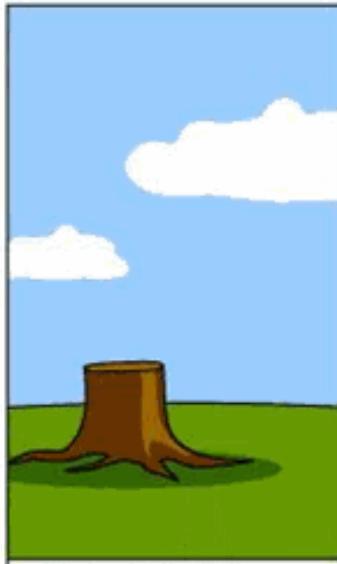
Comment le projet
a été documenté.



Ce que les opérationnels
ont installé.



Comment le client
a été facturé.



Comment le produit
a été pris en charge.



Ce dont le client
avait réellement besoin.

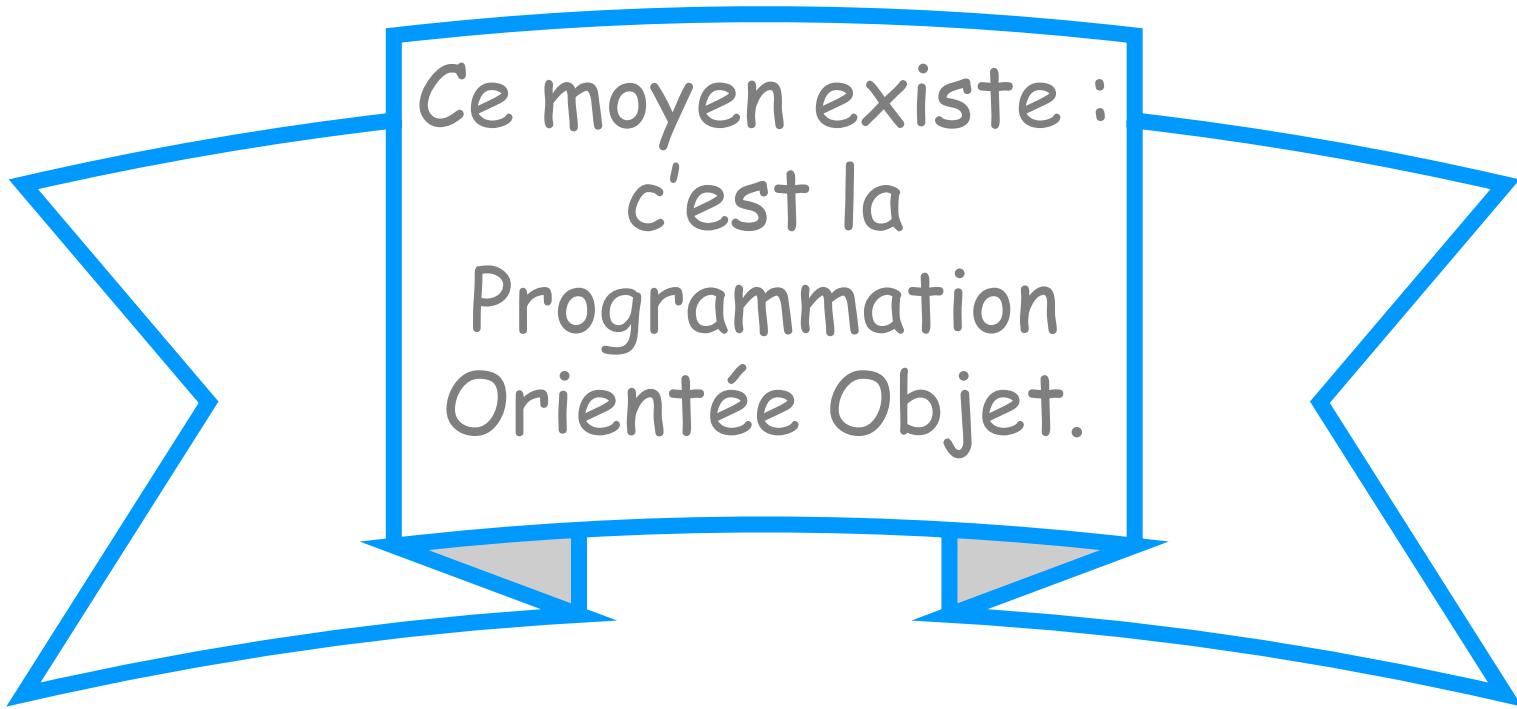
I have a dream today (1)

Ne serait-ce pas merveilleux s'il existait un moyen d'étendre un programme sans devoir modifier du code déjà testé et fonctionnel ?

Mais ce n'est probablement qu'un rêve...



I have a dream today (2)



Ce moyen existe :
c'est la
Programmation
Orientée Objet.

En quoi approche objet permet elle d'améliorer qualité logiciel ?

- Méthode fonctionnelle :
 - Décomposition des fonctionnalités du système.
 - Peut nécessiter restructuration importante si besoins changent (évolution données ou problématique).
- Approche orientée objet :
 - Se concentre sur l'identification des objets du système, puis fait coller les procédures à ces objets.
 - Résiste mieux quand besoins évoluent car s'appuie sur structure sous-jacente du domaine d'application plutôt que sur besoins fonctionnels liés à 1 seul pb.
 - Facilite maintenabilité, réutilisation, etc.

Programmation fonctionnelle (procédurale) vs POO (1)

Extrait de Java -Tête la Première, Kathy Sierra Bert Bates, 2005, O'Reilly.

- Soit les spécifications suivantes :
 - 3 formes affichées dans interface graphique.
 - Lorsqu'un utilisateur clique sur 1 forme :
 - Elle pivote de 360°.
 - Le système joue un fichier de son de format .aif associé à la forme.



- 2 programmeurs en concurrence.

Programmation fonctionnelle (procédurale) vs POO (2)

Extrait de Java -Tête la Première, Kathy Sierra Bert Bates, 2005, O'Reilly.



1. Programmeur procédural :

- Question : de quelles **procédures** ai-je besoin ?
- Réponse : **tourner()** et **jouerSon()**.

2. Adepte de l'Objet :

- Question : quels sont les **acteurs clés** dans ce programme ?
- Réponse : les **formes**, l'utilisateur, le son, le clic.

(Pour ces 3 derniers, je dispose déjà de code existant).

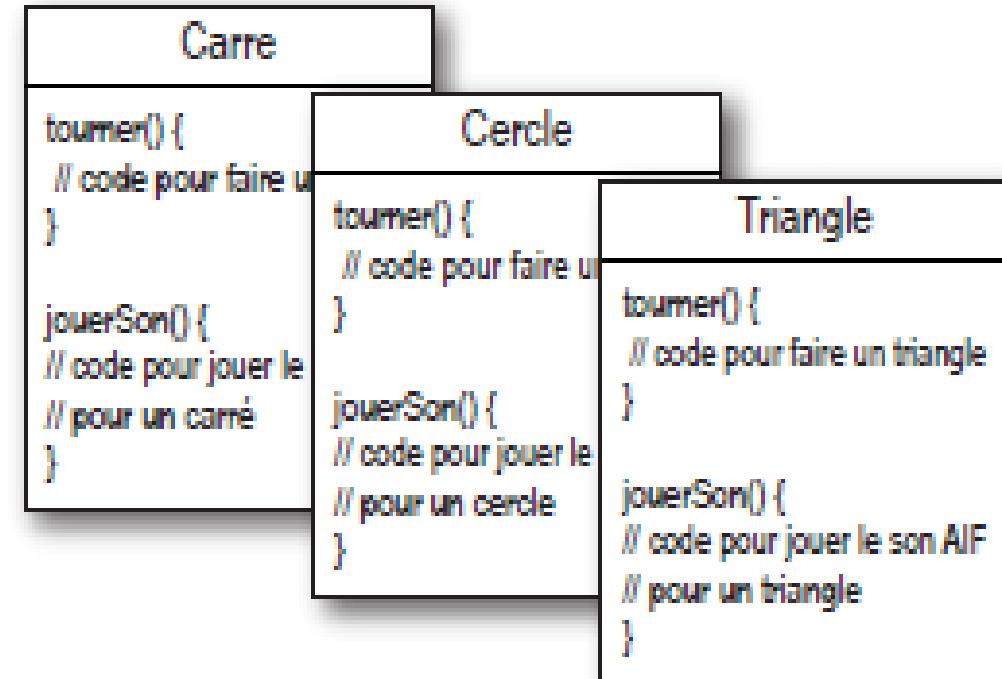
Programmation fonctionnelle (procédurale) vs POO (3)

Extrait de Java -Tête la Première, Kathy Sierra Bert Bates, 2005, O'Reilly.

1. Programmeur procédural :

```
tourner(typeForme) {  
    // faire tourner la forme de 360°  
}  
  
jouerSon(typeForme) {  
    // utiliser typeForme pour chercher  
    // quel son AIF jouer, puis le jouer  
}
```

2. Adepte de l'Objet :



A cette étape, programmeur 1. plus rapide que 2.

Programmation fonctionnelle (procédurale) vs POO (4)

Extrait de Java -Tête la Première, Kathy Sierra Bert Bates, 2005, O'Reilly.



Evolution spécification : amibe au milieu des autres formes qui après clic tourne et joue un son au format .hif.

1. Il faut **modifier** procédure existante `jouerSon()`.
2. Il faut **créer** une **nouvelle** forme.

```
jouerSon(typeForme) {  
    // si la forme n'est pas une amibe,  
    // utiliser typeForme pour chercher  
    // quel son AIF jouer, puis le jouer  
    // sinon  
    // jouer le son .hif de l'amibe  
}
```

```
Amibe  
  
tourner() {  
    // code qui fait tourner l'amibe  
}  
  
jouerSon() {  
    // code qui joue le nouveau  
    // fichier .hif pour une amibe  
}
```

A cette étape, égalité.

Programmation fonctionnelle (procédurale) vs POO (5)

Extrait de Java -Tête la Première, Kathy Sierra Bert Bates, 2005, O'Reilly.



Nouvelle évolution : l'amibe ne tourne pas comme les autres formes autour d'un axe central mais d'un axe en haut à gauche.

1. Il faut modifier la procédure existante tourner() en lui ajoutant des arguments pour le point de rotation et des tests.
2. Il faut modifier la procédure existante tourner() uniquement pour la forme Amibe en ajoutant les coordonnées du point en haut à gauche.

A cette étape, programmeur 2. plus rapide que 1.

Programmation fonctionnelle (procédurale) vs POO (6)

Extrait de Java -Tête la Première, Kathy Sierra Bert Bates, 2005, O'Reilly.



1. Programmeur procédural :

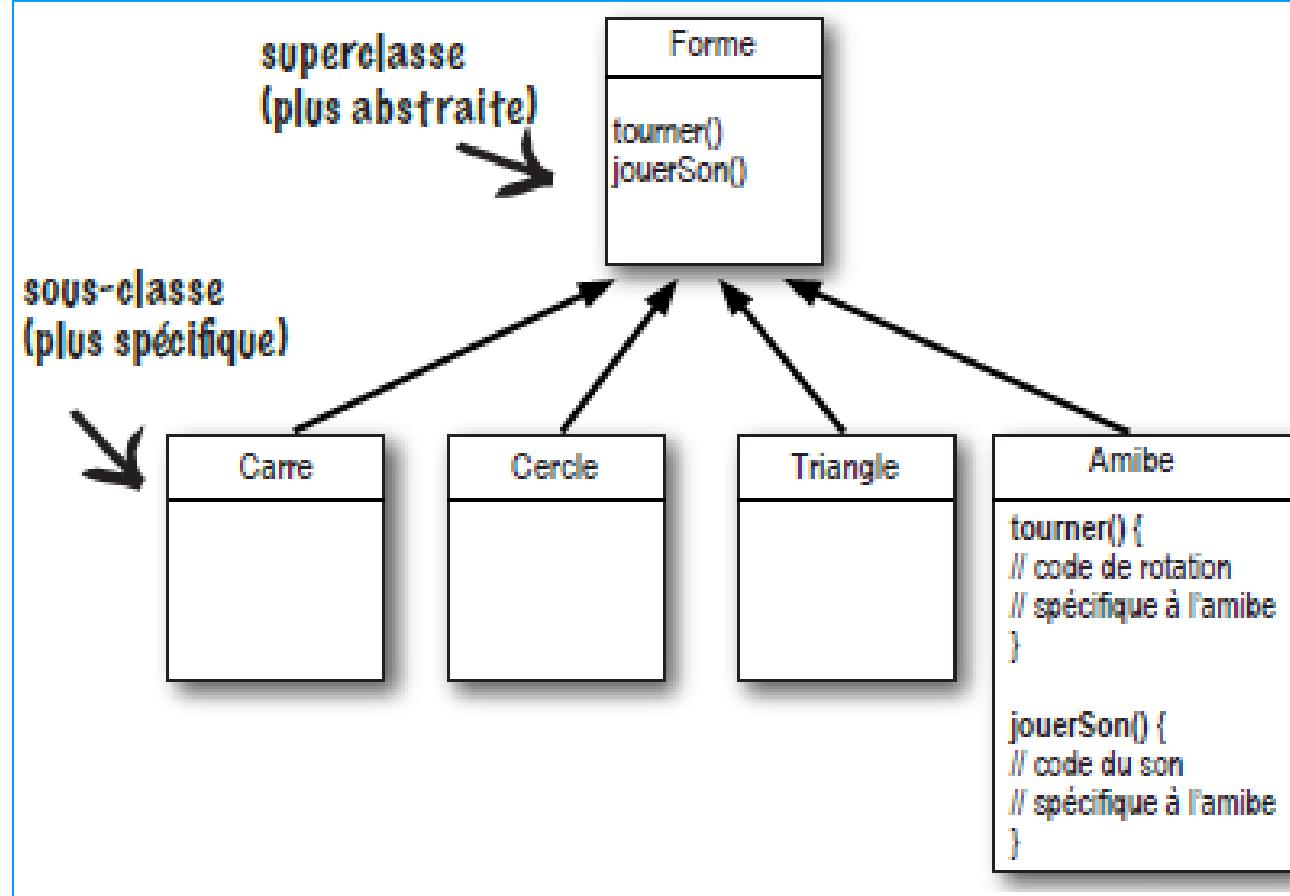
```
tourner(typeForme, xPt, yPt) {  
    // si la forme n'est pas une amibe,  
    // calculer le point central  
    // sur la base d'un rectangle,  
    // puis faire tourner  
    // sinon  
    // utiliser xPt et yPt comme  
    // décalage du point de rotation  
    // puis faire tourner  
}
```

2. Adepte de l'Objet :

```
Amibe  
  
int xPoint  
int yPoint  
tourner() {  
    // faire tourner l'amibe  
    // en utilisant x et y  
}  
jouerSon() {  
    // jouer le nouveau  
    // fichier .mid pour une amibe  
}
```

Programmation fonctionnelle (procédurale) vs POO (7)

Extrait de Java -Tête la Première, Kathy Sierra Bert Bates, 2005, O'Reilly.



Modèle objet final

I have a dream today (1)

Ne serait-ce pas merveilleux s'il existait une méthode permettant de concevoir une application Orientée Objet ?
Mais ce n'est probablement qu'un rêve...



I have a dream today (2)



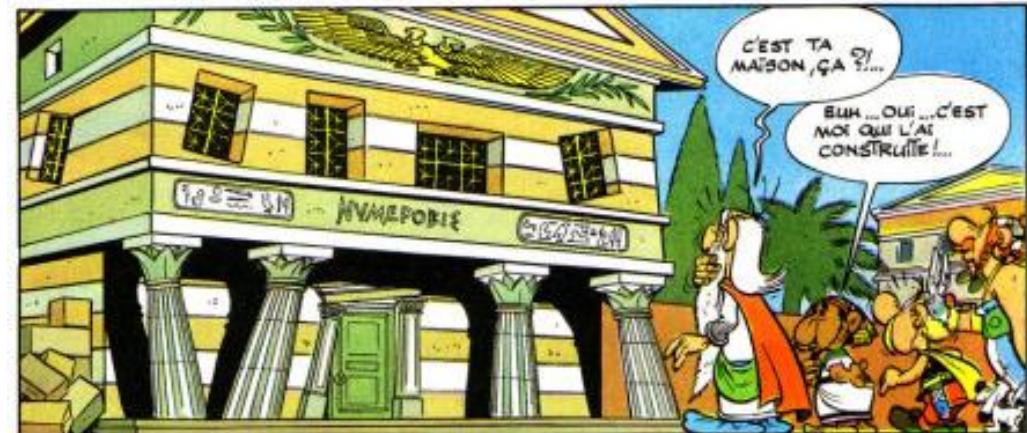
Cette méthode existe : c'est UP basée sur le langage UML.

C'est quoi finalement l'approche orientée objet ?

- Logiciel est une collection d'objets :
 - Dissociés, mais liés entre eux,
 - Comportant chacun à la fois :
 - un état (la structure de ses données)
 - et un comportement (les opérations dont il est capable).
- Alors comment écrit-on vraiment un bon logiciel ? :
 - Qui répond aux exigences du client.
 - Qui soit suffisamment souple pour être facile à maintenir.

Je suis architecte, je m'appelle Numerobis

- 3 modules complémentaires pour apprendre à écrire un bon logiciel :
 - Génie logiciel (4IRC – S7).
 - Conception Orientée Objet et Design Patterns (S8-S9 majeure Info)
 - Architecture des Systèmes d'Information (S8 majeure Info).
 - UML (intégré au projet transversal).



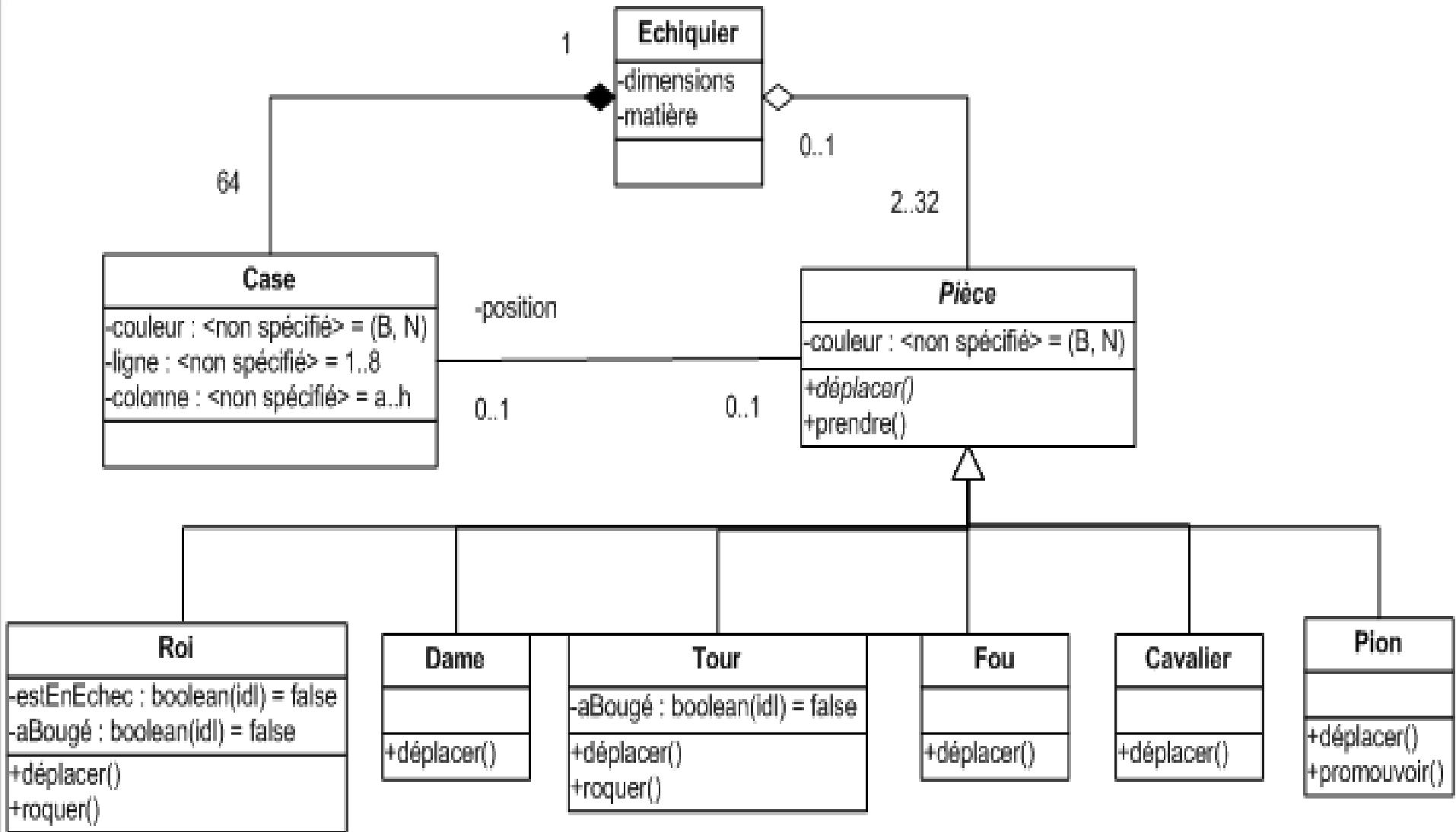
Modélisation des objets grâce à méthode UP (Unified Process) basée sur UML (Unified Modeling Language)

- Objectifs :
 - à partir des besoins des utilisateurs (à clarifier ☺), identifier les classes et les relations entre les classes (diagramme de classe) :
 - Associations, agrégation, composition.
 - Généralisation/Spécialisation.
 - Pour chaque classe, définir les attributs (d'instance ou de classe) et les opérations (méthodes ou opération de classe).

Les détails de la syntaxe du langage UML sont donnés en annexe 1 disponible sur le e-campus.

Exemple de diagramme de classes

Inspiration : UML 2 par la pratique, 7° ed., P. ROQUES, 2009, Eyrolles

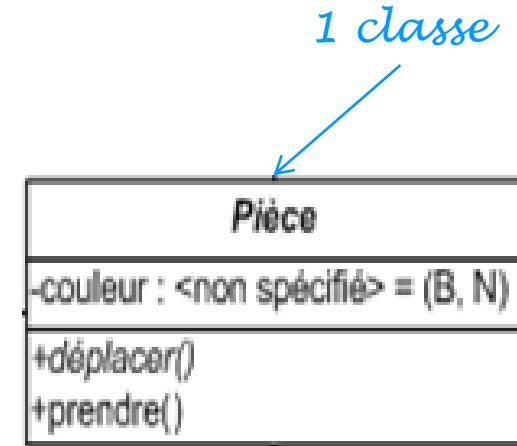


Du coup, qu'est-ce qu'une classe ?

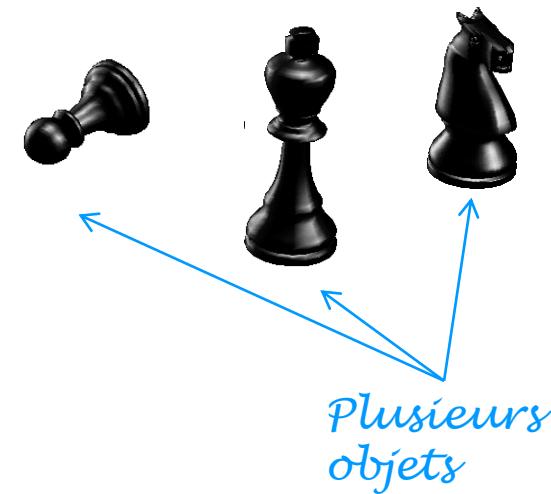
Qu'est ce qu'un objet ?

- **Classe** : c'est une **abstraction** du monde réel qui possède :

- Etat (attributs d'instance)
- Comportement (opération/méthodes)
- Ex : classe « Pièce »
 - Données : couleur.
 - Comportement : déplacer, prendre.



- **Objet** : **instance** de classe. Un objet est une occurrence unique de la classe.



Qu'est-ce qu'une méthode ?

- Les méthodes correspondent aux opérations appliquées aux objets ou par les objets définies dans une classe :
 - maTour.deplacer()
 - et non pas deplacer(@maTour).
- La question est de savoir, dans un diagramme de classe, quelle classe est **responsable** de l'opération (Cf. plus loin).
- Une méthode possède une signature (type des arguments et type de la valeur de retour).





Qu'est-ce qu'un attribut ?

- Les attributs identifiés dans une classe sont des données (ligne, colonne) de type **primitif** (entier, caractère, etc.).
- Si l'on souhaite modéliser le fait qu'une classe possède des données qui sont des objets, alors seront modélisés :
 - Un lien **d'agrégation** entre classe (Un ensemble de pièces sont disposées sur un Echiquier).
 - Un lien de **composition** (Echiquier est composé de Cases - la destruction de l'Echiquier entraîne celle des Cases).

Qu'est-ce qu'une association ?

- Une **association décrit un lien** entre les objets instances de deux classes (Une pièce est positionnée sur une Case).
- Elle exprime que les objets peuvent **communiquer** l'un avec l'autre :
 - Une pièce est positionnée sur une Case.
 - Une case « Connait » la Pièce qui l'occupe.
- Ou bien qu'un objet **délègue** à un autre objet la responsabilité d'un traitement.
- Il n'y a **pas de dépendance** entre les objets (la destruction de l'un n'entraîne pas celle de l'autre).



Qu'est-ce que l'agrégation ?

- L'**agrégation** est une association non symétrique, qui exprime un **couplage fort** et une relation de subordination. Elle représente une relation de type « ensemble / élément » (Un ensemble de pièces sont disposées sur un Echiquier).
- Une instance d'élément agrégé peut être liée à plusieurs instances d'autres classes (les Pièces sont aussi propriété d'un Joueur, etc.).
- Les cycles de vies de l'agrégat et de ses éléments agrégés peuvent être indépendants.

Qu'est-ce que la composition ?

- La **composition** est une **agrégation forte**
(Echiquier composé de cases, voiture composé de 4 roues).
- Les cycles de vies des éléments et de l'agrégat sont liés :
 - si l'agrégat est détruit (ou cloné), ses composants le sont aussi.
 - Un changement d'état d'un élément entraîne un changement d'état de l'agrégat (roue tourne ⇒ voiture se déplace).
Une instance de composant (roue) ne peut être liée qu'à un seul agrégat (voiture).





Qu'est-ce que l'héritage ?

- L'héritage est une relation de type « **Est une sorte de** » entre une super-classe (classe de base = Pièce) qui généralise des sous-classes (classes dérivées = Tour, Fou) qui spécialisent la super-classe.
- La sous-classe hérite des attributs et méthodes de sa super-classe et de toutes ses classes ancêtres (dans une hiérarchie de classe).
- Elle peut définir ses propres méthodes (`roquer()`) ou **redéfinir** celles de sa super-classe (même signature de la méthode `déplacer()`).
- Elle peut introduire de nouveaux attributs (`aBougé`).

I have a dream today (1)



*Ne serait-ce pas merveilleux s'il existait 3-4 mots clés à placer dans une conversation pour fanfaronner devant les néophytes ?
Mais ce n'est probablement qu'un rêve...*

I have a dream today (2)

Ces mots clés sont les piliers de l'approche objet.



Les piliers de l'approche objet (1)

- **Abstraction :**
 - Que fait l'objet et quelles données il manipule ?
 - Sans se soucier des détails d'implémentation (stockage données, algorithme et langage pour traitement).
- **Encapsulation :**
 - Aspects externes accessibles à d'autres objets.
 - Implémentation interne invisible.
 - Evite interdépendances : modification implémentation n'affecte pas les applications qui emploient l'objet.



Les piliers de l'approche objet (2)

■ Héritage :



Mécanisme de **généralisation/spécialisation** permettant de réutiliser des composants logiciels. Le code est également plus clair.

- Permet de définir de nouvelles classes dérivées **héritant** des données et fonctionnalités de la classe de base qu'elles pourront modifier et enrichir, sans remettre en question la classe de base.
- Ex : bouton « enregistrer », « tabulation », « gras » ont tous les données et comportements communs des « boutons » (position, visible, accès), et ont des comportements spécifiques (sauvegarde, etc.).

Les piliers de l'approche objet (2)

■ Polymorphism :

- Mécanisme permettant de manipuler des objets sans en connaître le type.
 - Permet à une **même fonction d'avoir des comportements différents** en fonction du type réel de l'objet.
 - Ex :
 - Tous les objets graphiques peuvent être visibles (bouton, boite de dialogue, menu déroulant, etc.).
 - Collection d'objets (animaux – parlent, mangent, se déplacent tous, mais de manière différente).



Kit de survie

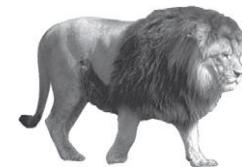


Ce qu'il faut retenir (1)

- La programmation orientée objet permet d'étendre un programme sans devoir modifier le code déjà testé et fonctionnel.
- Une classe définit la façon de créer un objet de ce type. Elle est comparable à un patron.
- Un objet peut prendre soin de lui-même. Vous n'avez pas besoin de savoir comment il le fait ni de vous en occuper.
- **Un objet sait des choses et fait des choses.**
- Les choses qu'un objet sait sur lui-même sont des variables d'instance (attributs). Elles représentent son état.
- Celles qu'il fait sont des méthodes. Elles représentent son comportement. Une classe peut hériter des variables d'instance et des méthodes d'une superclasse plus abstraite.

Ce qu'il faut retenir (2)

- Comment concevoir une hiérarchie de classes (d'Animaux par exemple) ?
 - Déterminez les caractéristiques abstraites que tous les objets partagent (attribut et comportement), puis intégrer ces caractéristiques dans une classe dont tous ces objets puissent dériver (Animaux).
 - Déterminez si une sous-classe nécessite des comportements (implémentations de méthodes) spécifiques à ce type particulier (manger(), parler(), vagabonder() différents pour Chien et Poisson).
 - Cherchez d'autres occasions d'utiliser l'abstraction, en trouvant des sous-classes qui pourraient avoir un comportement commun (félin, canin).





Chapitre 2 : Eléments du langage Java Classes et Objets

3IRC – 4ETI
Version 2013/2014 – Release 2015-2016

F. PERRIN

membre de UNIVERSITÉ DE LYON

CPE LYON
ÉCOLE SUPÉRIEURE
DE CHIMIE PHYSIQUE ÉLECTRONIQUE
DE LYON

Les maitres mots : souplesse, évolution et réutilisation

- Il existe de nombreuses classes très utiles dans l'API Java.
- Il est donc inutile de refaire le monde...
- ... mais voyons donc comment il tourne !



I have a dream today (1)

Ne serait-ce pas merveilleux s'il existait un langage qui permette de mettre en œuvre tous les concepts objets ?
Mais ce n'est probablement qu'un rêve...



I have a dream today (2)



Il s'agit du
langage Java.

Présentation du Langage Java (1)

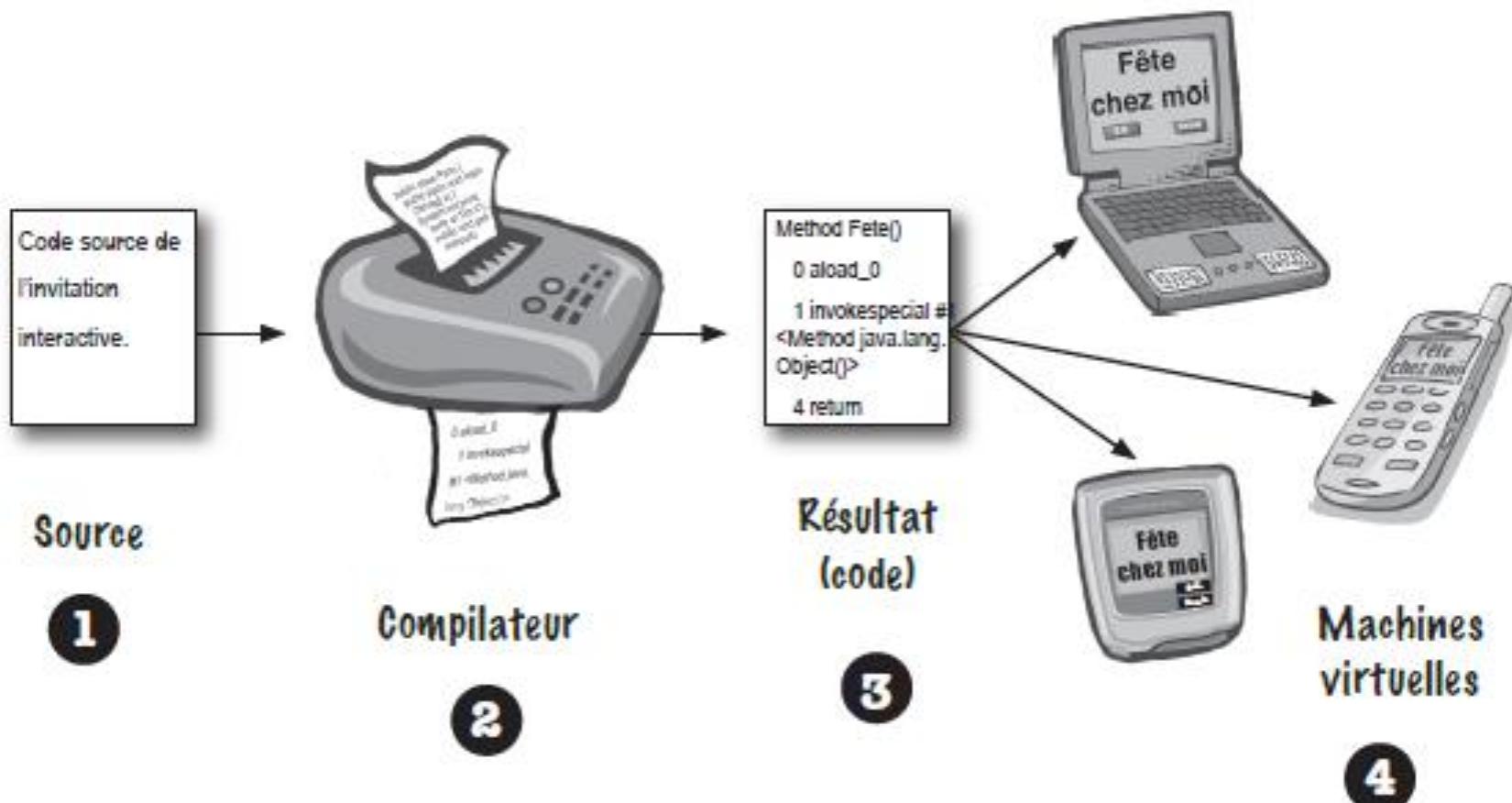


- Java est un langage développé par Sun Microsystems, racheté par Oracle.
- Java est un langage objet :
 - Toute ligne de code Java se trouve obligatoirement dans une fonction à l'intérieur d'une classe.
 - La syntaxe du langage est très proche de celle du C.
- Java permet de développer plusieurs types de pgms :
 - Application sous formes de fenêtres ou en mode console avec Java SE (Java Standard Edition).
 - Des applets (pgms java incorporés à des pages Web).
 - Des applications pour appareils mobiles (Smartphones...) avec J2ME (Java 2 Micro Edition).
 - Des sites Web dynamiques avec Java EE (Java Enterprise Edition).

Présentation du Langage Java (2)

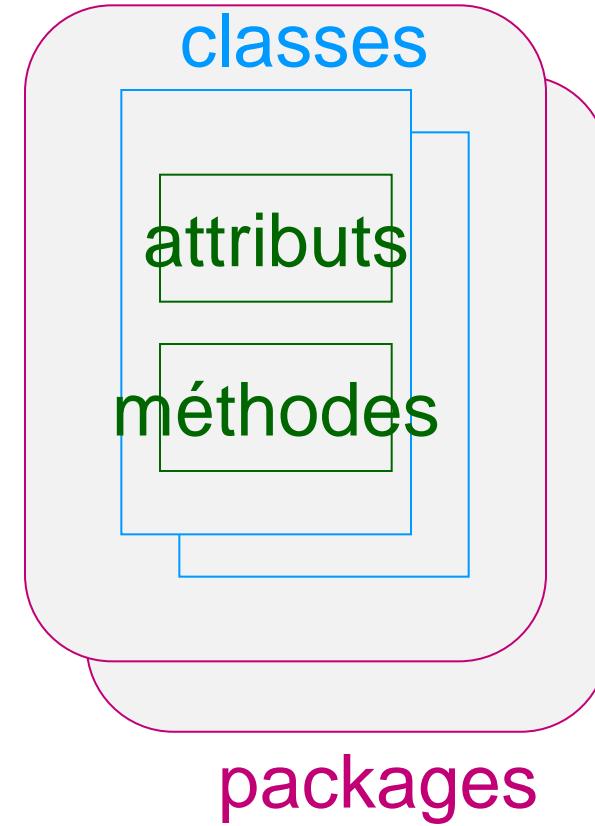
Extrait de Java -Tête la Première, Kathy Sierra Bert Bates, 2005, O'Reilly.

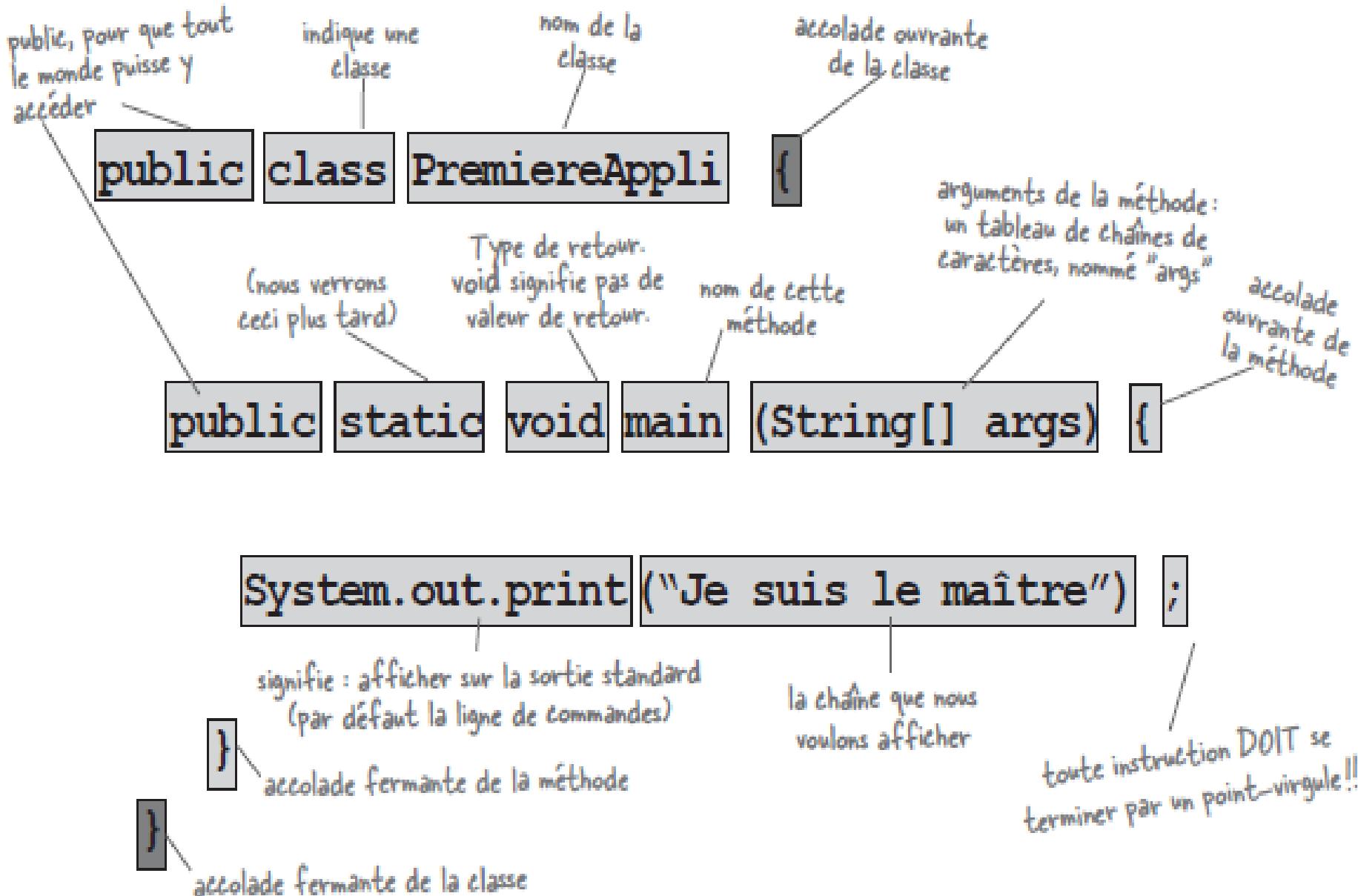
- Java est portable.
- Java est un langage compilé et interprété :



Structure d'un programme

- Un programme Java utilise un ensemble de **classes**.
- Les classes sont regroupées par **package**.
- Une classe contient des **attributs** et des **méthodes**.
- Une de ces classes doit définir une fonction `main()`, point d'entrée du programme.







Syntaxe du Java

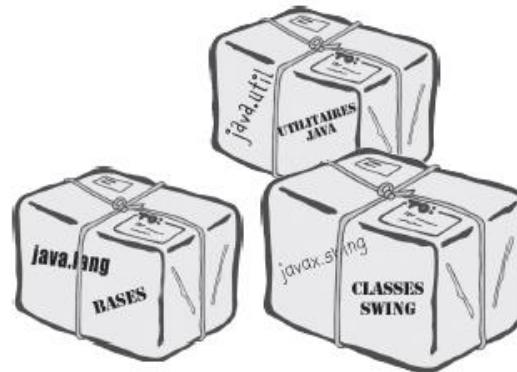
Les détails de la syntaxe du langage Java sont donnés en [Annexe 2](#) disponible sur le e-campus.

- Instructions.
- Types de données (types primitifs et tableaux).
- Structures de contrôle (boucles, tests).

Mais aussi :

- Compléments sur les packages, classes, attributs, méthodes, références.
- Exceptions.

L'API Java existante



- Les classes sont groupées en packages.
- Pour utiliser une classe d'un package autre que java.lang :
 - placer une instruction import au début du code source, ou
 - taper le nom complet partout elle est utilisée dans le code.

I have a dream today (1)

Ne serait-ce pas merveilleux s'il existait un moyen de dire à la machine virtuelle comment produire un objet d'un type donné.

Mais ce n'est probablement qu'un rêve...



I have a dream today (2)



Il s'agit de définir une classe.

Classe = patron pour construire des objets

Lorsque vous créez une classe, pensez à ce que **sait** l'objet (variable d'instance) et à ce qu'il peut **faire** (méthode)

Variables d'instance
(état)

Méthodes
(comportement)

Reveil
heureAlarme
modeAlarme
setHeureAlarme()
setAlarme()
alarmeProgramme()
pause()

savoir
faire

Membres d'une classe : Attribut d'instances et Méthode (1)

```
public class Circle {  
    private double x, y; ← Rq : Pas d'attributs public  
    private double r;  
    public Circle (double x, double y, double r) {  
        this.x = x; this.y = y; this.r = r; // Constructeur  
    }  
    public double circumference()  
    { return 2* 3.14159 * r; } ↑ passage implicite de  
    public double area() ← paramètre r  
    { return 3.14159*r*r ; }  
}
```



Instance de classe = Objet

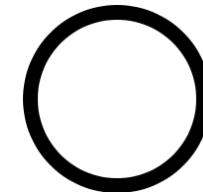
- L'accès aux attributs et aux méthodes définis dans la classe se fait à partir d'un **objet**.
- Un objet est une instance d'une classe.

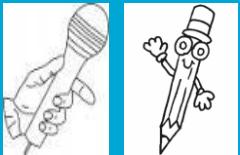
...
Circle c1, c2; // c1 et c2 sont des références
// vers des objets Circle

c1 = new Circle(); // Création d'un objet : appel
// du constructeur Circle()

c2 = new Circle (1,2,3); // Appel du constructeur
// Circle(double x, double y, double r)

double res = c1.area(); // Accès à la méthode area()





Référence vs Objet

- Crédit d'objets : **new**
 - Appelle le constructeur adéquat qui crée l'objet.
 - Retourne une **référence** vers l'objet créé.
- Variable référence : `Circle c = new Circle();`
 - `c` est une variable référence sur un objet de type `Circle`,
 - i.e. la valeur de `c` est une adresse.

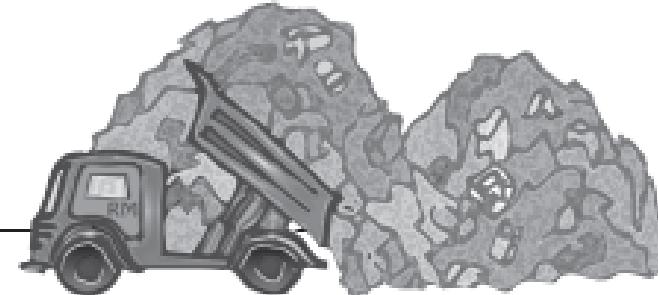
Visualisez la variable `c` comme une **télécommande** que vous utilisez pour amener l'objet (instance de `Circle`) à faire quelque chose : appeler une méthode avec la notation pointée `c.area()` ;

Vie et mort d'un objet





Comment un objet est-il déréférencé ?



Trois façons de se débarasser d'une référence :

- ① Elle sort de la portée, définitivement

```
void go() {  
    Vie z = new Vie();  
}
```

“z” meurt à la fin de la méthode.

- ② Elle est affectée à un autre objet.

```
Vie z = new Vie();  
z = new Vie();
```

Le premier objet est abandonné quand z est “reprogrammée”.

- ③ Elle est positionnée explicitement à null

```
Vie z = new Vie();  
z = null;
```

Le premier objet est abandonné quand z est “déprogrammée”.

Un objet est candidat au ramasse-miettes quand sa dernière référence vivante disparaît.

Encapsulation, accesseurs et mutateurs

Accessibilité	Visibilité attribut/méthode			
	public	protected	package	private
La même classe	yes	yes	yes	yes
Classe dans le même package	yes	yes	yes	no
Sous classe dans un autre package	yes	yes	no	no
Non sous classe, autre package	yes	no	no	no

Les accesseurs et mutateurs permettent l'encapsulation. Conventions de nommage : `getXxx` et `setXxx` ou `xxx` est le nom d'un attribut.



Attributs, variables locales, paramètres

- Les variables d'instance sont déclarées dans une classe. Elles vivent le temps de vie de l'objet qui les contient.

```
public class Canard {  
    int poids; // valeur par défaut en cas d'absence d'initialisation  
}
```

- Les variables locales sont déclarées dans une méthode et vivent le temps d'exécution de la méthode.

```
public void foo(int x) {  
    int i = x + 3;      // doit être initialisée  
    boolean b = true; // doit être initialisée  
}
```

- Les paramètres formels des méthodes sont des copies des paramètres effectifs. Ils vivent le temps d'exécution de la méthode.



Les classes Wrapper (enveloppes)

- Les types de bases java (`int`, `boolean`,..) ne sont pas des classes.
- Les classes **Wrapper** représentent un type de base et permettent de :
 - Créer une instance à partir d'un type de base et d'une chaîne.
 - Convertir un type de base en chaînes de caractères et inversement.
 - Utiliser des types de bases dans des conteneurs.
- `Boolean`, `Integer`, `Float`, `Double`, `Long`, `Character` (dans package `java.lang`).
- Attention : on ne peut pas utiliser les opérateurs arithmétiques ni logiques avec les classes enveloppes.

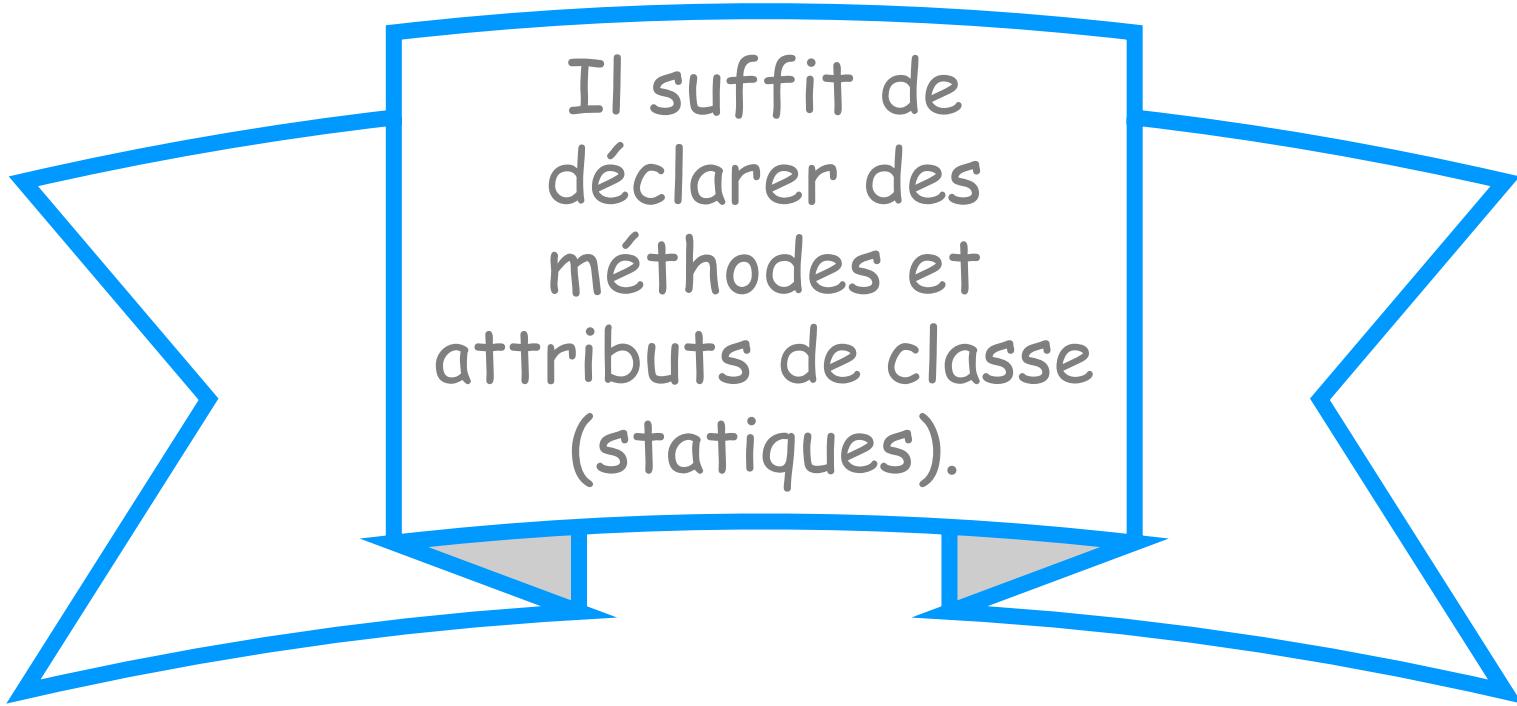


I have a dream today (1)

Ne serait-ce pas merveilleux s'il existait un moyen d'exécuter du code indépendamment de toute instance de classe (objet), par exemple pour des fonctionnalités plus utilitaires ? Mais ce n'est probablement qu'un rêve...



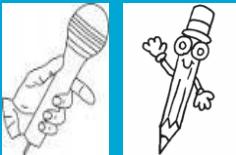
I have a dream today (2)



Il suffit de déclarer des méthodes et attributs de classe (statiques).

Attributs et fonctions membres de classes (statiques)

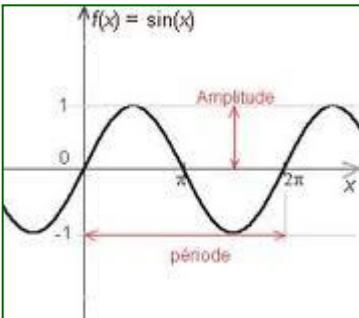
- Toute ligne de code est écrite dans une classe, mais tout n'est pas objet. Les classes utilitaires :
 - Regroupent des attributs et opérations **statiques**.
 - N'ont pas de constructeur public mais 1 constructeur **privé** empêchant l'instanciation d'objet.
- Appel des fonctions membres de classe (déclarées en **static**) :
 - nomClasse.nomFonction - Ex : double y = Math.sin(x);
- **Attributs de classe** :
 - Dans les classes utilitaires : données manipulées par les opérations statiques.
 - Ou bien attributs **partagés** par toutes les instances.



Bloc d'initialisation des variables statiques

- **Bloc statique** : pour initialiser des variables statiques.
- Exécuté lorsque classe est chargée par Machine Virtuelle.
- Ce bloc commence directement par le mot **static**.

```
public class MesMaths {  
    static private double sinus[] = new double[1000];  
    static {  
        double x, delta_x = 1;  
        for(int i = 0, x = 0.0; i < 1000; i++, x += delta_x)  
            sinus[i] = Math.sin(x);  
    }  
    private MesMaths() { // existe juste pour empêcher new}  
    public static void getSinus(int n) { ... }  
    // .....  
}
```



Kit de survie



Ce qu'il faut retenir (1)

- Tout code Java est défini dans une classe.
- Lorsqu'il s'exécute, un programme Java n'est rien d'autre qu'un ensemble d'objets qui communiquent.
- Java est un langage à typage fort.
- Il existe deux sortes de variables : les types primitifs et les références.
- Les variables doivent toujours être déclarées avec un nom et un type.
- Une variable primitive contient des bits qui représentent sa valeur (5, 'a', true, 3.1416, etc.).

Ce qu'il faut retenir (2)

- Une variable référence contient les bits qui représentent une façon d'accéder à un objet sur le tas.
- Une variable référence est semblable à une télécommande. Utiliser l'opérateur point (.) sur une variable référence équivaut à appuyer sur un bouton de la télécommande pour accéder à une méthode ou une variable d'instance.
- Une variable référence a la valeur null quand elle ne référence plus aucun objet.
- Un tableau est toujours un objet, même s'il est déclaré pour contenir des types primitifs.
- Une fois que vous avez déclaré un tableau, vous ne pouvez pas y placer n'importe quoi : les éléments doivent être du type que vous avez spécifié.

Ce qu'il faut retenir (3)

- Les méthodes utilisent les variables d'instance pour que des objets du même type puissent se comporter différemment.
- Une méthode peut avoir des paramètres, ce qui vous permet de lui transmettre une ou plusieurs valeurs.
- Une méthode doit déclarer un type de retour. Le type void signifie qu'elle ne retourne rien.
- Si le type est différent de void, la méthode doit retourner une valeur compatible avec le type déclaré.

Ce qu'il faut retenir (4)

- Un constructeur est le code qui s'exécute quand vous écrivez `new` suivi d'un type de classe.
- Un constructeur doit porter le même nom que la classe, et ne peut pas avoir de type de retour.
- Vous pouvez utiliser un constructeur pour initialiser l'état (les variables d'instance) de l'objet construit.
- Si vous ne placez pas de constructeur dans votre classe, le compilateur en créera un par défaut.
- Le constructeur par défaut n'a jamais d'argument.
- Si vous placez un constructeur quelconque dans votre classe, le compilateur ne fournira pas de constructeur par défaut.
- Écrivez toujours un constructeur sans argument pour faciliter la création des objets. Fournissez des valeurs par défaut.

Ce qu'il faut retenir (5)

- Des constructeurs surchargés signifient que votre classe a plusieurs constructeurs.
 - Les constructeurs surchargés doivent avoir des listes d'arguments différentes.
 - Deux constructeurs ne peuvent pas avoir la même liste d'arguments. Une liste d'arguments spécifie leur ordre et leur type.
- Les variables d'instance résident dans l'objet auquel elles appartiennent, sur le Tas.
- Si la variable d'instance est une référence à un objet, la référence et l'objet sont tous deux sur le Tas.
- Les variables d'instance ont une valeur par défaut, même si vous n'en affectez pas explicitement : 0/0.0/false pour les types primitifs et null pour les références.

Ce qu'il faut retenir (6)

- On appelle une méthode statique à l'aide d'un nom de classe et non d'une référence à un objet : Math.random() vs monObjet.go();
- Une méthode statique peut être invoquée en l'absence de toute instance de la classe de la méthode sur le tas.
- Une méthode statique convient pour une méthode utilitaire qui ne dépend pas (et ne dépendra jamais) de la valeur d'une variable d'instance donnée.
- Une méthode statique n'étant pas associée à une instance particulière — seulement à une classe — elle ne peut accéder à aucune des valeurs des variables d'instance de sa classe. Elle ne saurait pas les valeurs de quelle instance utiliser.
- Une méthode statique ne peut pas accéder à une méthode non statique, puisqu'une méthode non statique est généralement associée à l'état d'une variable d'instance.

Ce qu'il faut retenir (7)

- Si une classe ne possède que des méthodes statiques et que vous voulez empêcher de l'instancier, vous pouvez marquer le constructeur avec le mot-clé private.
- Une variable statique est une variable partagée par tous les membres d'une classe donnée. Elle n'existe qu'en un seul exemplaire dans une classe, alors que les instances possèdent chacune une copie des variables d'instance.
- Une méthode statique peut accéder à une variable statique.
- Une constante est une variable statique et finale (ne peut pas être redéfinie).



Chapitre 3 : Héritage et Polymorphisme Interfaces

3IRC – 4ETI
Version 2013/2014 – Release 2015-2016

F. PERRIN

membre de UNIVERSITÉ DE LYON

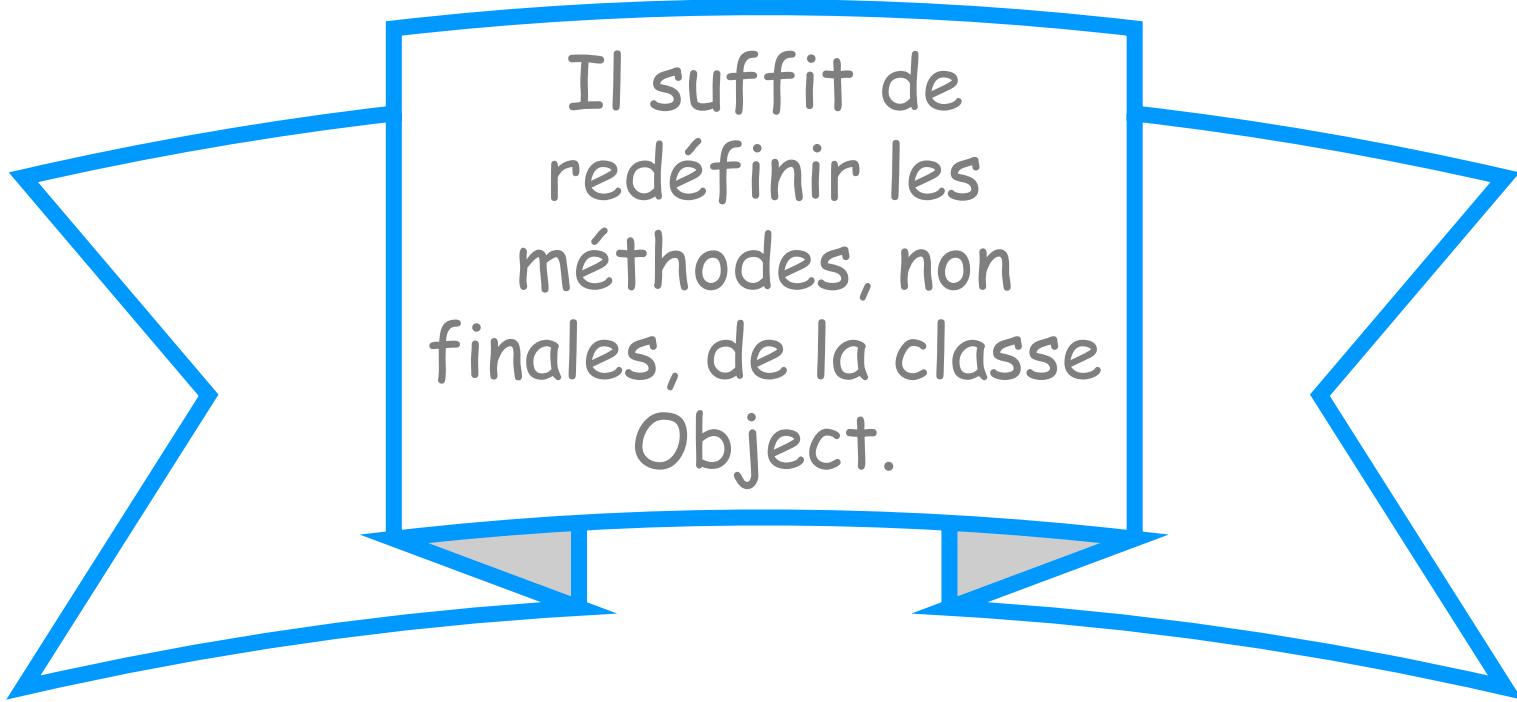
LYON
CPE
ÉCOLE SUPÉRIEURE
DE CHIMIE PHYSIQUE ÉLECTRONIQUE
DE LYON

I have a dream today (1)

Ne serait-ce pas merveilleux s'il existait un moyen de se rappeler quelles sont les méthodes qu'il faut créer a minima dans chacune de nos classes perso ?
Mais ce n'est probablement qu'un rêve...



I have a dream today (2)



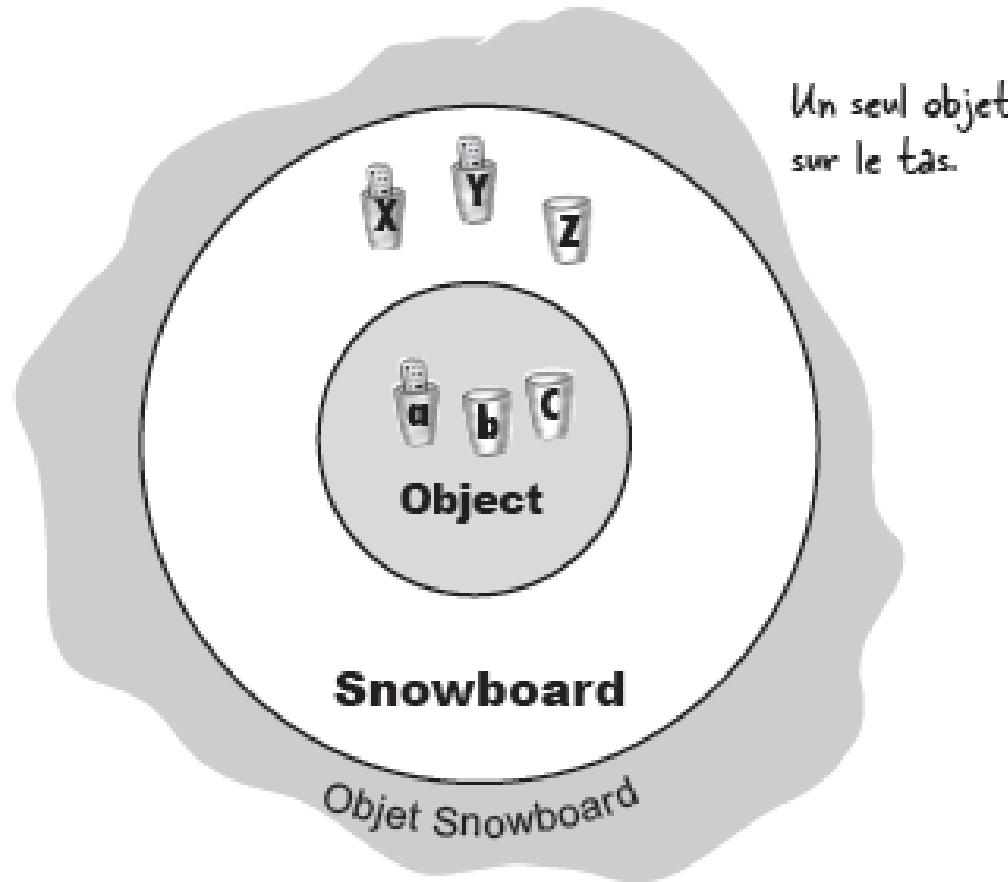
Il suffit de redéfinir les méthodes, non finales, de la classe Object.

Toutes les classes héritent de la classe Object

Object
Foo a; int b; int c;
equals() getClass() hashCode() toString()
Snowboard
Foo x Foo y int z
tourner() pivoter() sauter() tomber()

Object a des variables d'instance encapsulées par des méthodes d'accès. Ces variables d'instance sont créées quand une sous-classe quelconque est instanciée. (Ce ne sont pas les VRAIES variables d'Object, mais peu importe, puisqu'elles sont encapsulées.)

Snowboard a aussi ses propres variables d'instance : pour créer un objet Snowboard nous avons besoin de place pour les variables d'instance des deux classes.



Ici, il n'y a qu'**UN** objet sur le tas. Un objet **Snowboard**. Mais il contient à la fois des composantes **Snowboard** et des composantes **Object**. Toutes les variables d'instance des deux classes doivent être là.

Que fait la classe Object ?

```
Public class Object {  
    protected Object clone() // crée et retourne une copie de l'objet.  
    public boolean equals(Object) // teste l'égalité sémantique de  
                                // 2 objets.  
    protected void finalize() // appelée par garbage collector.  
    public final Class getClass() // retourne le descripteur de classe.  
    public int hashCode() // retourne un entier identifiant l'objet.  
    public Object()  
    public String toString() // retourne description de l'objet  
    // ...                  // sous forme chaîne caractères  
}
```

(_) /
(='.'=)
(")_("

A quoi sert la classe Object ?

- Toute classe existante dans l'API Java ou définie par un programmeur hérite de ces méthodes et redéfinit à minima les méthodes `equals()`, `hashcode()` et `toString()`.
- Elle permet la **généricité** : capacité d'effectuer le même traitement quel que soit le type d'objet manipulé :
 - `public void add(Object o);`

Comment vérifier que 2 objets sont égaux ? (1)

- Si deux objets sont égaux, l'appel de equals() sur l'un des objets DOIT retourner vrai :
si (a.equals(b)) alors (b.equals(a)).
- Si deux objets ont le même code de hachage, ils ne sont PAS nécessairement égaux.
- Mais **s'ils sont égaux**, ils DOIVENT avoir le **même** code.
- Par défaut, la méthode equals() effectue une comparaison == i.e. elle teste si les deux références pointent sur un seul objet.
- Par défaut, hashCode() génère un entier unique pour chaque objet sur le tas.

Comment vérifier que 2 objets sont égaux ? (2)

- Pour vérifier l'égalité (sémantique de 2 objets), il faut redéfinir equals() et hashCode(). Exemple :

```
public class Etudiant {  
    private String nom;  
    public boolean equals(Object o) {  
        if(!(o instanceof Etudiant)) return false;  
        Etudiant e2=(Etudiant) o; //cast  
        if( this.nom.equals(e2.nom)) return true;  
        else return false;  
    }  
    public int hashCode() {  
        return nom.hashCode();  
    }  
}
```

Comment vérifier que 2 objets sont égaux ? (3)

- Conclusion :
 - $a.equals(b) \Rightarrow b.equals(a)$
 - $a.equals(b) \Rightarrow a.hashCode() == b.hashCode()$
 - Mais $a.hashCode() == b.hashCode()$ n'implique PAS nécessairement $a.equals(b)$

I have a dream today (1)

Ne serait-ce pas merveilleux s'il existait un moyen d'imposer à nos classes d'implémenter un comportement spécifique et d'informer les classes clientes de nos classes que tel est le cas ?
Mais ce n'est probablement qu'un rêve...



I have a dream today (2)



C'est possible
grâce à la
puissance des
interfaces.

Comment comparer, cloner, etc. 2 objets ?

- Il faut implémenter les interfaces existantes Comparable et Clonable. Ex :

```
public class Etudiant implements Comparable, Clonable {  
    private String nom;  
    public int compareTo(Object o) {  
        ...  
        if ((o != null) && (o instanceof Etudiant))  
            return this.nom.compareTo(o.nom) ;  
    }  
    public Object clone() {  
        return new Etudiant(nom) ;  
    }  
}
```

Du coup c'est quoi exactement une interface en Java ?

- **Grand Gourou** : « Une interface Java définit le comportement attendu par une classe. »
- **Développeur sceptique** : « Heu ! Est-ce bien nécessaire d'implémenter une interface ? Les méthodes publiques de la classes ne définissent elles pas son comportement attendu, i.e. ce que l'on appelle son interface ? »
- **GG** : « Oui c'est vrai, mais cela permet de **découpler** les clients / des classes concrètes et ainsi de faciliter l'extension des programmes sans avoir à les modifier. »
- **DS** : « Heu ! C'est grave si je n'ai pas compris ? »
- **GG** : « Non, prends de bonnes habitudes, tu me remercieras plus tard ».

I have a dream today (1)

Ne serait-ce pas merveilleux s'il existait un moyen de centraliser le code commun dans une classe et de ne modifier que localement cette classe en cas de changement de comportement ?

Mais ce n'est probablement qu'un rêve...



I have a dream today (2)



Ce moyen existe :
c'est l'héritage.

Qu'est-ce qu'on avait dit que c'était déjà l'héritage ?

- Mécanisme représentant une relation de **Généralisation/Spécialisation** pour modéliser une relation de type « **Est une sorte de** » entre une super-classe qui généralise et des sous-classes qui spécialisent la super-classe..
- La sous-classe hérite des attributs et méthodes de sa super-classe et de toutes ses classes ancêtres (dans une hiérarchie de classe).
- La classe dérivée peut changer l'implémentation d'une ou plusieurs méthodes héritées : **redéfinition**.
- Toute classe Java est une sous classe de la classe **Object**.





Un exemple type d'implémentation de l'héritage en Java

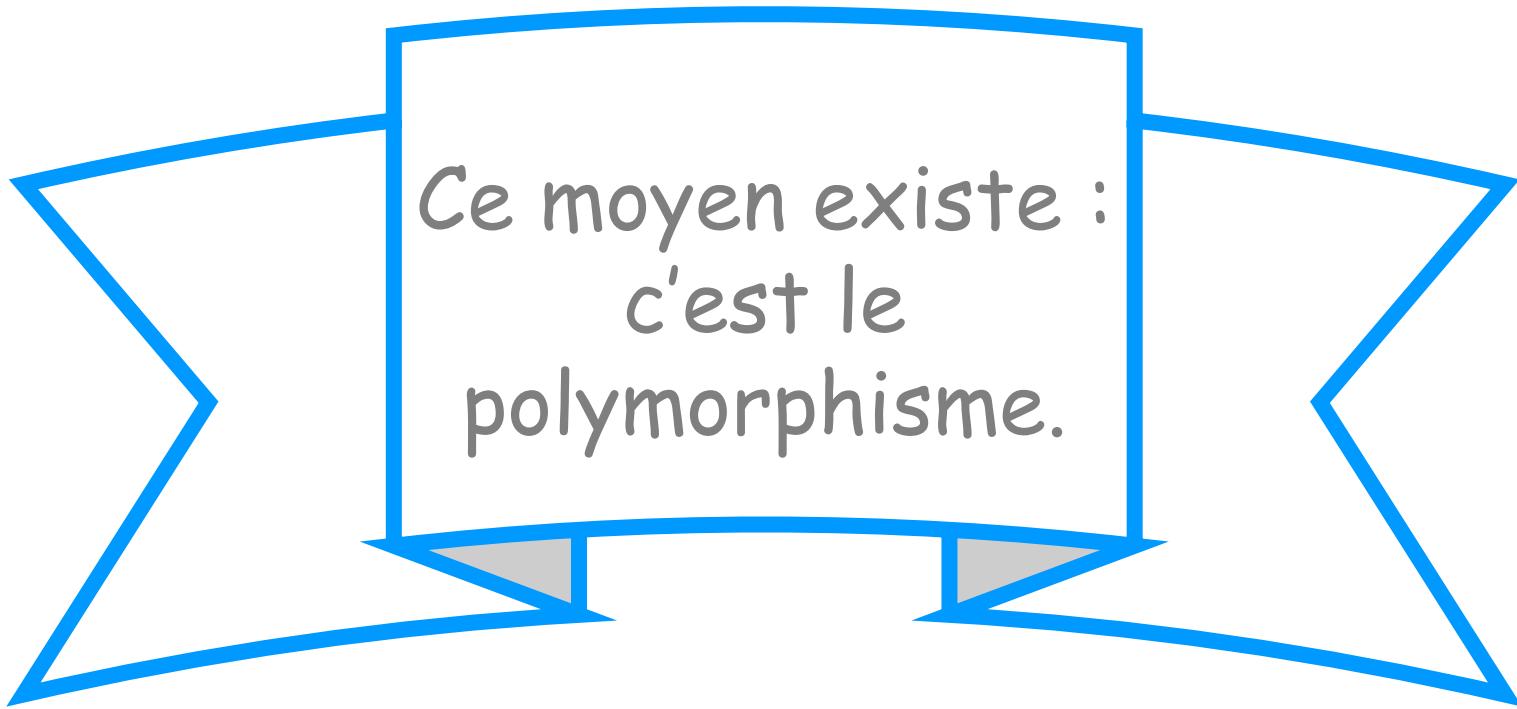
```
class abstract ClasseDeBase{  
    protected int val;  
    ClasseDeBase (int val) {  
        this.val = val; }  
    public int getVal() {  
        // ... }  
    public float calcul () {  
        // ... }  
    }  
class ClasseDerivee extends ClasseDeBase {  
    float x;  
    ClasseDerivee (int val, float x) { // attention : appel constructeur classe  
        super(val);  
        // de base doit être la 1ère instruction  
        this.x = x;  
    }  
    public float calcul() { // redéfinition méthode calcul() héritée  
        return super.calcul() * x;  
    }  
    // héritage sans redéfinition méthode getVal()  
}
```

I have a dream today (1)

Ne serait-ce pas merveilleux s'il existait un moyen d'étendre un programme sans modifier le code existant lorsque je rajoute une sous-classe.
Mais ce n'est probablement qu'un rêve...



I have a dream today (2)



Ce moyen existe :
c'est le
polymorphisme.

Un 1^{er} exemple de polymorphisme

```
public class MonstreTest {  
    public static void main(String [] args) {  
        Monstre [] ma = new Monstre[3];  
        ma[0] = new Vampire();  
        ma[1] = new Dragon();  
        ma[2] = new Monstre();  
        for(int x = 0; x < 3; x++) {  
            System.out.println (  
                ma[x].fairePeur(x));  
        }  
    }  
}
```

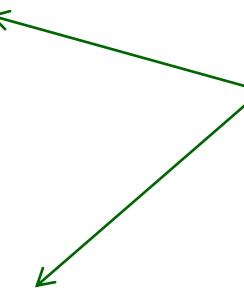
```
class Monstre {  
    String fairePeur(int d) {  
        return ("arrrgh");  
    }  
}
```

```
class Vampire extends Monstre {  
    String fairePeur(int x) {  
        return ("mordre?");  
    }  
}
```

```
class Dragon extends Monstre {  
    /// ...  
}
```

Un 2^{ème} exemple de polymorphisme

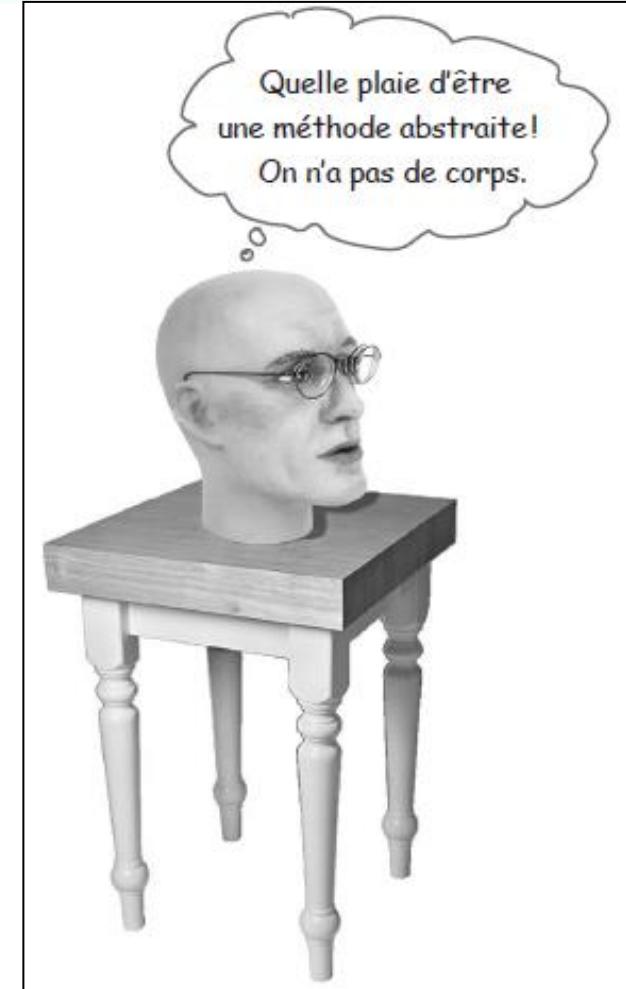
```
class Veto {  
    public void soigner(Animal a) {  
        // faire d'horribles choses à l'Animal  
        // à l'autre bout du paramètre 'a', puis  
        a.parler();  
    }  
}  
  
class ProprietaireDAnimaux {  
    public void start() {  
        Veto v = new Veto();  
        Chien c = new Chien();  
        Hippo h = new Hippo();  
        v.soigner(c);  
        v.soigner(h);  
    }  
}
```



On imagine les classes Chien et Hippo dérivées de Animal existantes et munies d'une méthode parler()

Qu'est-ce qu'une méthode abstraite ?

- Une méthode abstraite est une méthode :
 - Qui ne peut pas être définie à ce niveau de la hiérarchie de classes : elle n'a pas de corps : **parler()** dans la classe **Animal**.
 - Dont on donne la signature sans l'implémentation en la préfixant par le mot clé **abstract**.





Qu'est-ce qu'une classe abstraite ?

- Une **classe abstraite** :
 - Ne peut pas être instanciée : vous ne pouvez pas en créer des instances avec `new` : **Animal**
 - Doit être déclarée avec le mot clé **abstract**.
 - Peut ne contenir aucune méthode abstraite.
 - En revanche, si elle contient au moins 1 méthode abstraite, elle est abstraite.

```
public abstract class Animal {  
    public abstract void manger();  
    public abstract void bouger();  
    public abstract void parler();  
    // ...  
}
```

Mise en œuvre du Design Pattern Template Method (1)

- Une **classe abstraite** :
 - Peut contenir des « patron de méthode » qui définissent les étapes d'un algorithme et délèguent aux sous-classes l'implémentation de certaines parties de cet algorithme.
 - Un « patron de méthode » est une méthode d'une classe abstraite qui **fait appel à des méthodes abstraites concrétisées par ses classes dérivées**.
 - **final** est le mot clé utilisé pour faire en sorte que l'on ne puisse pas surcharger une telle méthode – **Ex : Boissons caféinées.**
 - Cela fonctionne car l'objet qui est instancié est forcément au bas de la hiérarchie de classe.



Design Pattern Template Method (2)

Illustre Principe d’Inversion de Dépendance dit d’Hollywood
« Ne nous appelez pas, nous vous appellerons »



```
public abstract class BoissonCafeinee {  
    final void suivreRecette() {  
        faireBouillirEau();  
        preparer();  
        verserDansTasse();  
        ajouterSupplements();  
    }  
    protected abstract void preparer();  
    protected abstract void  
        ajouterSupplements();  
    private void faireBouillirEau() { //... }  
    private void verserDansTasse() { //... }  
}
```

```
public class Café extends  
    BoissonCafeinee {  
    protected void preparer()  
        { // Passage du café }  
    protected void ajouterSupplements()  
        { // Ajout lait et sucre }  
    }  
  
public class The extends  
    BoissonCafeinee {  
    protected void preparer()  
        { // infusion du thé }  
    protected void ajouterSupplements()  
        { // Ajout citron }  
    }
```

On est toujours capable de mettre en œuvre le polymorphisme

```
public class TestBoisson {  
    public static void main(String[] args) {  
        BoissonCafeinee the = new The();  
        BoissonCafeinee cafe = new Cafe();  
        System.out.println("\nPréparation du thé...");  
        the.suivreRecette();  
        System.out.println("\nPréparation du café...");  
        cafe.suivreRecette();  
    }  
}
```

// ou bien

```
BoissonCafeinee[] tab = new BoissonCafeinee[10];  
tab[0] = new The();  
tab[1] = new Cafe(); // ...  
for (int i = 0; i < 10; i++)  
    tab[i].suivreRecette();  
}
```



Comment fonctionne le polymorphisme ? (1)

- Choix signature(s) méthode(s) potentielle(s) est effectué au moment de la **compilation** en fonction type **déclaré** de l'objet.
- Choix méthode à utiliser au moment de l'exécution :
 - Par liaison dynamique en fonction du type **réel** de l'objet.
 - Choix effectué parmi les méthodes **candidates** ayant la signature identifiée à la compilation.



Comment fonctionne le polymorphisme ? (2)

- Soient les définitions de classes suivantes :

```
class A {           public void f(float x) { ... }           }
class B extends A {
    public void f(float x) { ... }      // redéfinition de f de A
    public void f(int n) { ... }        // surdéfinition de f de A et de B
}
```

- Avec les déclarations :

A a = new A(); **B b = new B();** **int n = 10 ;**

- Soient les instructions :

a=b;
a.f(n);

- La méthode appelée est :

void f(float x) { ... } de la classe **B** (seule signature candidate).

I have a dream today (1)

Ne serait-ce pas merveilleux s'il existait un moyen d'affecter plusieurs comportements très différents à des objets, tout en bénéficiant de la puissance du polymorphisme ?
Mais ce n'est probablement qu'un rêve...



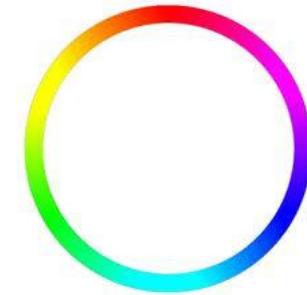
I have a dream today (2)



Rappelez vous de
la puissance des
interfaces.

Exemple simple d'utilisation des interfaces

```
public class Circle {  
    protected double x, y, r;  
    public double circumference() {return 2* 3.14159 * r;}  
    public double area() {return 3.14159*r*r ; }  
}  
public interface Drawable {  
    public void setColor(Color c);  
    public void setPosition(double x, double y);  
    public void draw(DrawWindow dw);  
}  
public class DrawableCircle extends Circle implements Drawable {  
    private Color c;  
    public void setColor(Color c) { this.c =c;}  
    public void setPosition(double x, double y) {this.x = x; this.y= y;}  
    public void draw(DrawWindow dw) { dw.drawcircle(x,y,r,c) }  
}
```



Kit de survie



Exemple de synthèse (1)

- Imaginons une hiérarchie de classes d'animaux qui bougent, mangent, parlent, etc. utilisée dans un didacticiel sur les animaux.
- Imaginons une application pour une animalerie qui voudrait utiliser les classes Chien, Chat, etc. et leur ajouter un comportement d'animal de compagnie : etreAmical, jouer, etc.
- Comment intégrer ces méthodes : 4 moyens, 1 seul correct ...



Exemple de synthèse (2)

- Méthodes concrètes dans classe animal :
 - Pour : économique.
 - Contre : tous les animaux en héritent : pas une bonne idée pour le requin...
- Méthodes abstraites dans classe animal, surchargées dans sous-classes :
 - Pour : on est sûr qu'un hippopotame ne sera pas amical.
 - Contre : toutes les sous-classes doivent surcharger les méthodes même à vide... et annonceraient aux clients qu'elles sont capables de ce comportement alors que c'est faux.

Exemple de synthèse (3)

- Méthodes concrètes seulement dans classes qui doivent avoir ce comportement :
 - Pour : pas de risque de se méprendre sur les intentions d'un animal.
 - Contre : nécessité d'un protocole avec les clients qui devraient connaître les fonctionnalités de chaque classe de la hiérarchie sans possibilité de polymorphisme.
- Définir une interface Compagnon qui serait seulement implémentée par les classes qui ont ce comportement :
 - Pour : polymorphisme possible sur hiérarchie de classes dérivées de Animal.
 - Conséquence : transtypage nécessaire pour invoquer les méthodes de Compagnon.

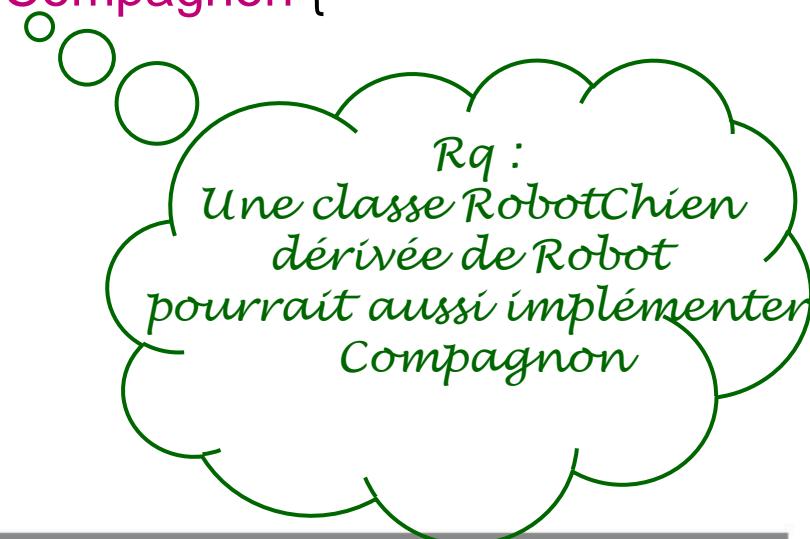
Exemple de synthèse (4)

```
interface Animal {  
    void manger();  
    void bouger();  
    void parler();      // ...  
}  
interface Compagnon {  
    void jouer();        void etreAmical();      // ...  
}  
abstract class AbsAnimal implements Animal{  
    // attributs communs à tous les animaux  
    // + getter() et setter()  
}  
abstract class Canin extends AbsAnimal {  
    // implémentation de certaines des méthodes de Animal  
    // lorsque ça a du sens pour tous les Canins  
}
```

Exemple de synthèse (5)

```
class Loup extends Canin { // implémente Animal par héritage
    public void bouger() { }
    public void manger() { }
    public void parler() { }
}
```

```
class Chien extends Canin implements Compagnon {
    // Méthode interface Compagnon
    public void etreAmical() { }
    public void jouer() { }
    //...
    // Méthode Interface animal
    public void parler() { }
}
```



Exemple de synthèse (6)

```
public class TestAnimal {  
    public static void main (String args[]) {  
        Animal[] tabAnimal = new Animal[10];  
        Compagnon compagnon;  
        tabAnimal[0]= new Loup();  
        tabAnimal[1]= new Chien();  
        for (int i = 0; i < 2; i++) {  
            tabAnimal[i].manger(); // etc.  
            try { // si l'animal est aussi un Compagnon  
                compagnon = (Compagnon) tabAnimal[i];  
                compagnon.jouer(); // etc.  
            }  
            catch (ClassCastException e) { }  
        }  
    }  
}
```

Comment aurait-il fallu procéder si la classe Chien existait déjà ?



Réponse dans module Conceptualisation Orientée Objet et Design Patterns



Kit de survie



Ce qu'il faut retenir (1)

- Une sous-classe étend 1superclasse et 1 seule.
- Une sous-classe hérite de toutes les méthodes publiques de la superclasse, mais elle n'hérite pas des privées. Elle hérite également de l'ensemble de ses variables d'instance (bien que les variables privées ne soient pas accessibles directement).
- Les méthodes héritées peuvent être redéfinies ; les variables d'instance ne peuvent pas l'être (elles peuvent être redéfinies dans la sous-classe, mais ce n'est pas la même chose et on en a très rarement besoin).
- On invoque la version de la superclasse d'une méthode dans une sous-classe qui l'a redéfini en utilisant le mot-clé super. (super.nomMethode(); - super() pour appel constructeur)

Ce qu'il faut retenir (2)

- Effectuez le test EST-UN pour vérifier que votre hiérarchie d'héritage est valide. Si X étend Y, alors X EST-UN Y doit avoir un sens.
- Si B étend A et si C étend B, B EST-UN A, C EST-UN B, et C EST-UN A également.
- La relation EST-UN ne fonctionne que dans une seule direction. Un Chien est un Animal, mais tous les Animaux ne sont pas des Chien.
- Quand une méthode est redéfinie dans une sous-classe, et que cette méthode est appelée sur une instance de la sous-classe, c'est la méthode redéfinie qui est appelée. (La moins haut placée gagne.)

Ce qu'il faut retenir (3)

- Quand vous ne voulez pas qu'une classe soit instanciée utilisez le mot-clé abstract.
- Une classe abstraite peut avoir des méthodes abstraites et des méthodes non abstraites.
- Une classe qui a ne serait-ce qu'une seule méthode abstraite doit être déclarée abstraite.
- Une méthode abstraite n'a pas de corps, et sa déclaration se termine par un point-virgule (pas d'accolades).
- Toute méthode abstraite doit être implémentée par la première classe concrète de la hiérarchie d'héritage.

Ce qu'il faut retenir (4)

- Comment savoir s'il faut créer une classe, une sous-classe, une classe abstraite ou une interface ?
 - Créez une classe qui n'étend rien (à part Object) quand elle ne réussit pas le test EST-UN pour tout autre type.
 - Utilisez une interface pour définir un rôle que d'autres classes puissent jouer, indépendamment de leur place dans la structure d'héritage.
 - Utilisez une classe abstraite quand vous voulez définir un patron pour un groupe de sous-classes, et/ou que vous avez au moins un peu de code commun. Rendez la classe abstraite pour garantir que personne ne pourra créer d'objets de ce type.
 - Créez une sous-classe uniquement si vous voulez obtenir une version plus spécifique d'une classe et redéfinir ou ajouter des comportements (attention à ne pas nuire au polymorphisme en cas d'ajout).

Ce qu'il faut retenir (5)

- Une interface permet de définir un Type Abstrait de Données et d'indiquer qu'un objet « a un (comportement) » et non pas juste « est un (sousType) ».
- Il est donc possible d'avoir une référence sur un objet implémentant une interface :
 - `Drawable monCercle = new DrawableCircle();`
- Une classe peut implémenter une ou plusieurs interfaces (Voiture : « Drawable » et « Reparable »).
- Des objets complètement différents répondent au même message, s'ils implémentent, souvent différemment, la même interface (DrawableCircle et Voiture implémentent Drawable).



Ce qu'il faut retenir (6)

- Une interface peu contenir des constantes (`static` et `final` non obligatoire) qui doivent être initialisées.
- Une interface fournit signature méthodes : les classes qui implémentent une interface sont responsables de cette implémentation.
- Une classe qui implémente une interface doit implémenter toutes les méthodes de cette interface sinon elle est abstraite.
- Toutes les méthodes d'une interface sont donc souvent implémentée grâce à plusieurs classes dans une hiérarchie de classes (Cf. Framework de collections Java chapitre suivant).
- Une interface peut hériter d'une ou plusieurs interfaces contrairement aux classes : Héritage multiple.



Chapitre 4 : Structures de stockage et de données Framework de Collections Java

3IRC – 4ETI
Version 2013/2014

F. PERRIN

membre de UNIVERSITÉ DE LYON

CPE
ÉCOLE SUPÉRIEURE
DE CHIMIE PHYSIQUE ÉLECTRONIQUE
DE LYON

I have a dream today (1)

Ne serait-ce pas merveilleux s'il existait un moyen de stocker et de ranger plusieurs éléments du même type et de pouvoir y accéder facilement et/ou rapidement ?

Mais ce n'est probablement qu'un rêve...



I have a dream today (2)



Nous connaissons
déjà les tableaux
et les listes
chainées.

Structures de stockage

- Pour stocker une collection de données en mémoire on peut utiliser :
 - Des **tableaux** :
 - Valeurs contigües.
 - Utilisation : accès directs plus fréquents que ajouts/suppressions.
 - Des **listes chainées** :
 - Allocation dynamique.
 - Valeurs non contigües.
 - Utilisation : ajouts/suppressions plus fréquents que accès directs.



La persistance de la mémoire
Salvador Dalí - 1931



Les tableaux

- L'attribut `length` renvoie le nb d'éléments du tableau.
- L'indice des éléments va de 0 à `length-1`.
- Déclaration et allocation :

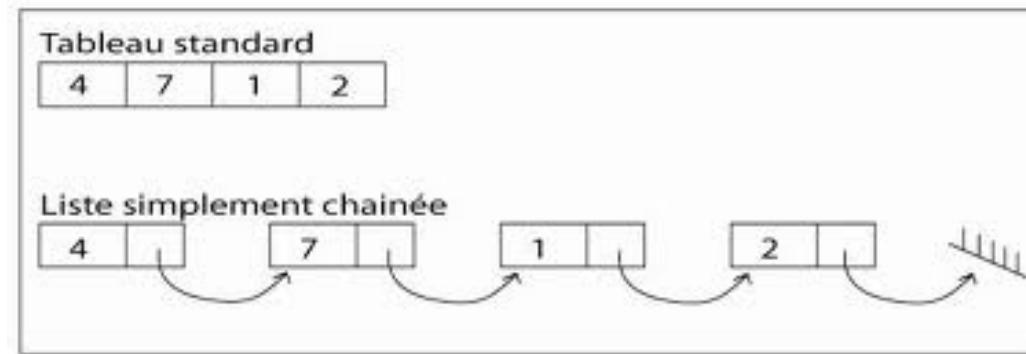
```
int [] tab = {0, 1, 2};           // allocation statique
char data[] = {'a', 'b', 'c'};
double tab2[] = new double[10];   // allocation dynamique
```

- Ces 2 parcours sont équivalents :

```
for (int i=0; i< tab2.length; i++)
    somme += tab2[i];
for (double elem : tab2)      // boucle « for each »
    somme += elem;
```

Les listes chainées (1)

- Elles utilisent une structure de maillon qui contient l'information.
- Chaque maillon contient une référence vers le maillon suivant et/ou précédent selon les besoins de parcours.

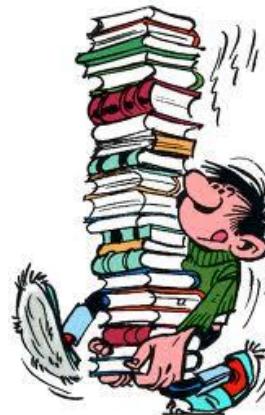


Les listes chainées (2)

```
private static class Node
{
    Object object;
    Node previous, next;
    Node(Object object, Node previous, Node next) {
        this.object = object;
        this.previous = previous;
        this.next = next;
    }
    Node(Object object) {
        this.object = object;
        this.previous = this.next = this;
    }
}
```

Les structures de données (TAD : Type Abstrait de Données)

- Elles définissent l'**organisation** des données :
 - **Comportement** attendu (LIFO, FIFO, liste, ensemble, arbre, graphe, etc.).
 - **Opérations** attendues (empiler, dépiler, etc.).
 - **Pré-conditions** pour effectuer ces opérations (impossible de dépiler si pas d'éléments).



I have a dream today (1)

Ne serait-ce pas merveilleux s'il existait un tableau qui rétrécirait quand on supprime quelque chose. Et qu'on n'aurait pas besoin de parcourir pour vérifier chaque élément, mais auquel on pourrait simplement demander s'il contient ce qu'on cherche. Et on pourrait en extraire des valeurs sans savoir exactement où elles sont.

Mais ce n'est probablement qu'un rêve...



I have a dream today (2)



Un tel objet
existe :
c'est une
ArrayList.





ArrayList vs tableau

```
ArrayList<String> maListe = new  
    ArrayList<String>();
```

```
String a = new String("bouh!");
```

```
maListe.add(a);
```

```
String b = new  
    String("Grenouille");
```

```
maListe.add(b);
```

```
int laTaille = maListe.size();
```

```
Object o = maListe.get(1);
```

```
maListe.remove(1);
```

```
boolean qqch =  
    maListe.contains(b);
```



```
String [] maListe = new  
    String[2];
```

```
String a = new String("bouh!");
```

```
maListe[0] = a;
```

```
String b = new  
    String("Grenouille");
```

```
maListe[1] = b;
```

```
int laTaille = maListe.length;
```

```
String o = maListe[1];
```

```
maListe[1] = null;
```

```
boolean qqch = false;  
for (String item : maListe) {  
    if (b.equals(item[i])) {  
        qqch = true;  
        break;  
    }
```

Qu'est-ce qu'une ArrayList ?

- Une ArrayList est une **liste**, implémentée dans un **tableau**, c'est-à-dire un ensemble ordonné d'éléments que l'on peut parcourir en **séquence**.
- Une ArrayList se redimensionne **dynamiquement**, quelle que soit la taille nécessaire. Elle grandit quand on insère des objets et rétrécit quand on en supprime.
- On déclare le type du tableau en utilisant un paramètre de type, qui est un nom de type entre crochets angulaires. Exemple : `ArrayList<Button>` signifie que l'ArrayList ne pourra contenir que des objets de type Button (ou de sous-classes de Button).
- Bien qu'une ArrayList contienne des objets et non des types primitifs, le compilateur « emballera » (et « déballera » ensuite) un type primitif dans un objet, et placera cet objet dans l'ArrayList à sa place.

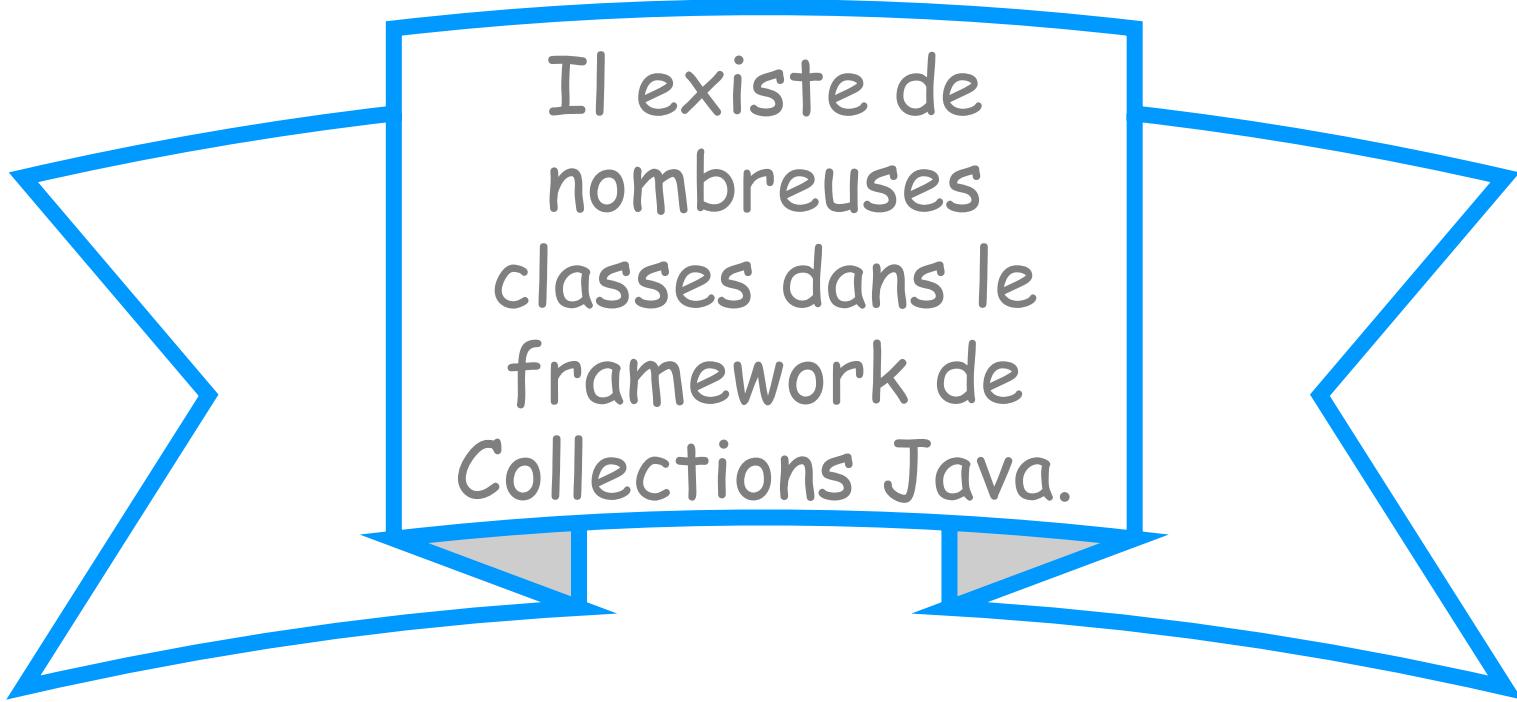
I have a dream today (1)

Ne serait-ce pas merveilleux s'il existait en Java des classes pour implémenter toutes les structures de données soit sous forme de tableau soit de liste chainée ?

Mais ce n'est probablement qu'un rêve...



I have a dream today (2)

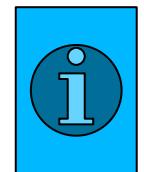
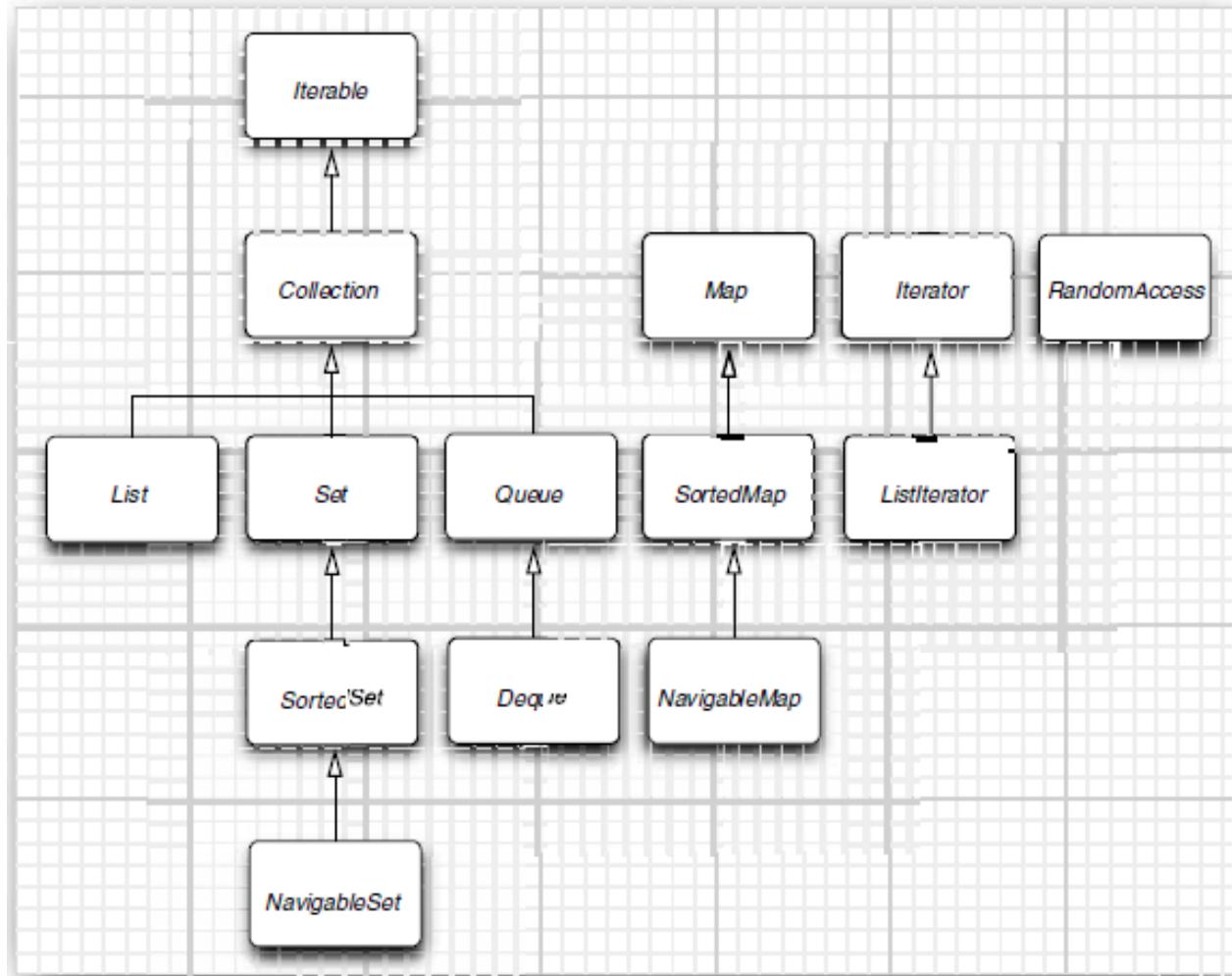


Il existe de nombreuses classes dans le framework de Collections Java.

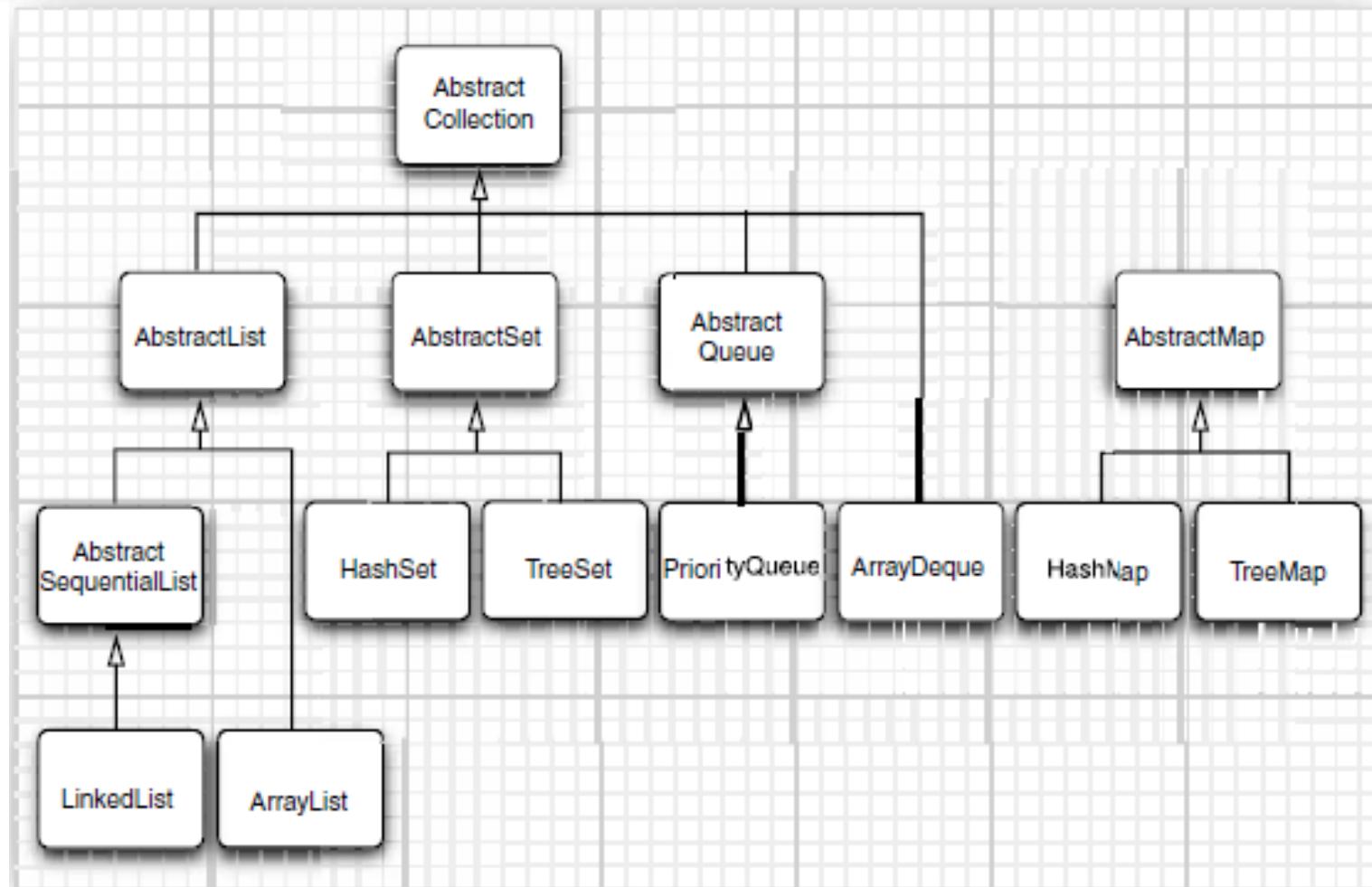
Notion de collection en Java

- Une **collection** est un **conteneur d'objets** (en réalité, de **références** vers des objets en Java ☺).
- Sur lequel on peut effectuer des opérations telles que :
 - Manipulation d'éléments : consultations (recherche), mises à jour, suppressions, ajouts.
 - Manipulation de la collection : parcours, tri, mélange, copie, recherche min ou max, etc.
- Le choix d'une collection se fait en fonction :
 - du mode **d'organisation** (séquence, ensemble, etc.)
 - **d'accès** (séquentiel, direct indexé, direct par hachage) et de stockage.

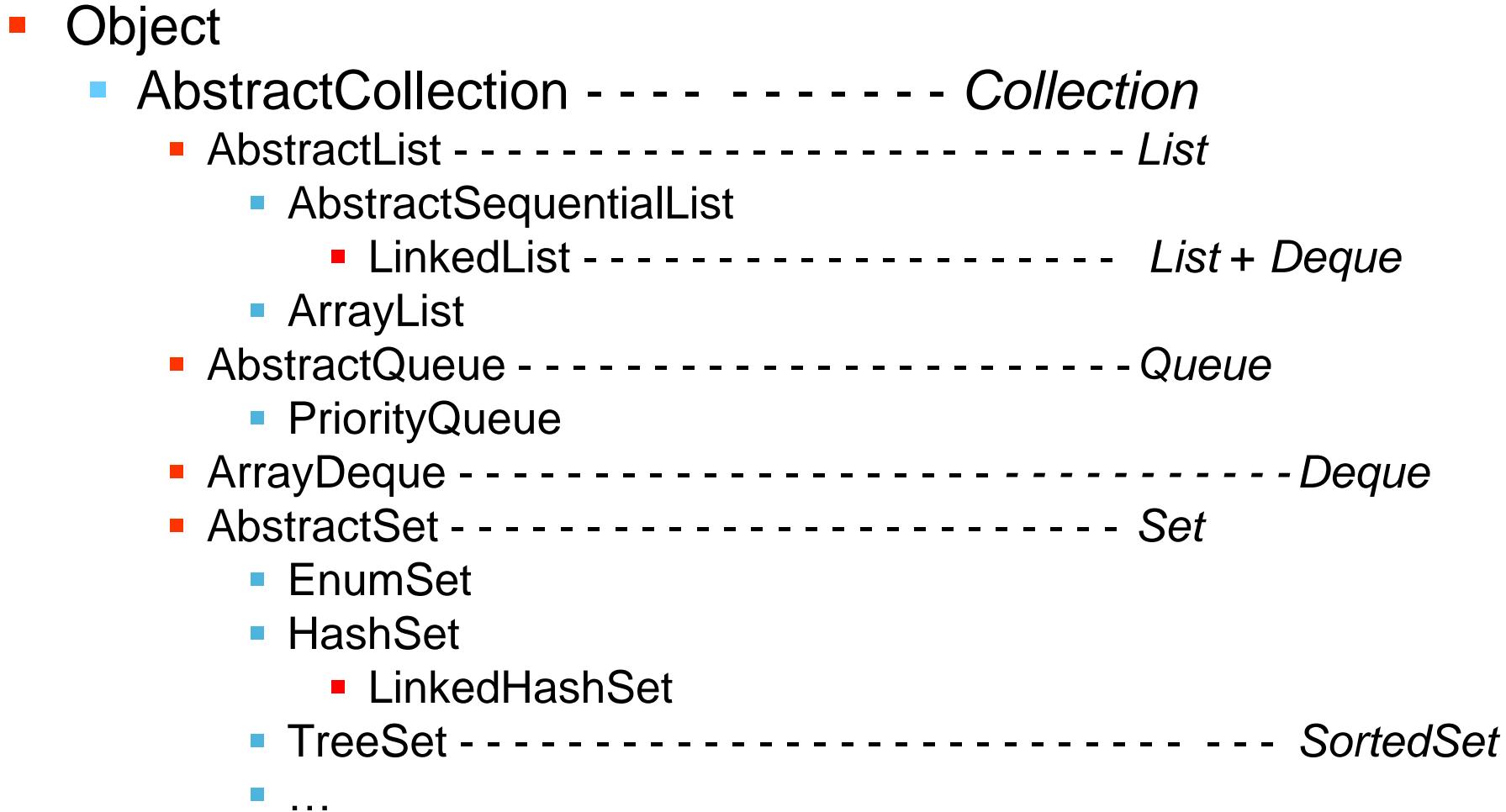
Hiérarchie des interfaces du Framework de collections Java



Hiérarchie des classes du Framework de collections Java (1)



Hiérarchie des classes du Framework de collections Java (2)



Hiérarchie des classes du Framework de collections Java (3)

- Object
 - AbstractMap - - - - - *Map*
 - EnumMap
 - HashMap
 - LinkedHashMap
 - IdentityHashMap
 - TreeMap - - - - - *SortedMap*
 - WeakHashMap
 - Arrays (algorithmes de manipulation des tableaux [])
 - BitSet
 - Collections (algorithmes de manipulation des collections)



Les méthodes De l'interface Collection

```
public boolean add(Object Object) ;  
public boolean addAll(Collection collection) ;  
public void clear() ;  
public boolean contains(Object Object) ;  
public boolean containsAll(Collection c) ;  
public boolean equals(Object Object) ;  
public int hashCode() ;  
public boolean isEmpty() ;  
public Iterator iterator() ;  
public boolean remove(Object Object) ;  
public boolean removeAll(Collection collection) ;  
public boolean retainAll(Collection collection) ;  
public int size() ;  
public Object[] toArray() ;  
public Object[] toArray(Object[] objects) ;
```

Comportement des classes du Framework de collections

- À chaque niveau de la hiérarchie, toutes les classes abstraites :
 - Donnent une définition d'autant de méthodes qu'elles en sont capables par rapport aux interfaces qu'elles tentent d'implémenter.
 - Héritent de méthodes à travers la hiérarchie en les redéfinissant ou non selon si elles souhaitent surcharger leur fonctionnalités.
 - Peuvent invoquer des méthodes qui sont implémentées **plus bas** dans la hiérarchie (DP Template Method).
- Dans toutes les classes instanciables :
 - Toutes les méthodes sont implémentées,
 - ou l'ont été dans les classes de plus haut niveau.

Comportement des classes du Framework de collections : exemple

- La classe `AbstractList` étend la classe `AbstractCollection` qui étend la classe `Object`. Elle peut utiliser ou surcharger les méthodes héritées.

- La méthode `equals()` qui teste l'égalité sémantique de 2 listes pourraient être définie ainsi :

```
public boolean equals(Object object) { // DP Template Method
    if (object == this)
        return true;
    if ( !(object instanceof AbstractList) ) return false;
    AbstractList abstractList = (AbstractList) object;
    if ( abstractList.size() != size() ) return false;
    return containsAll(abstractList);
}
```

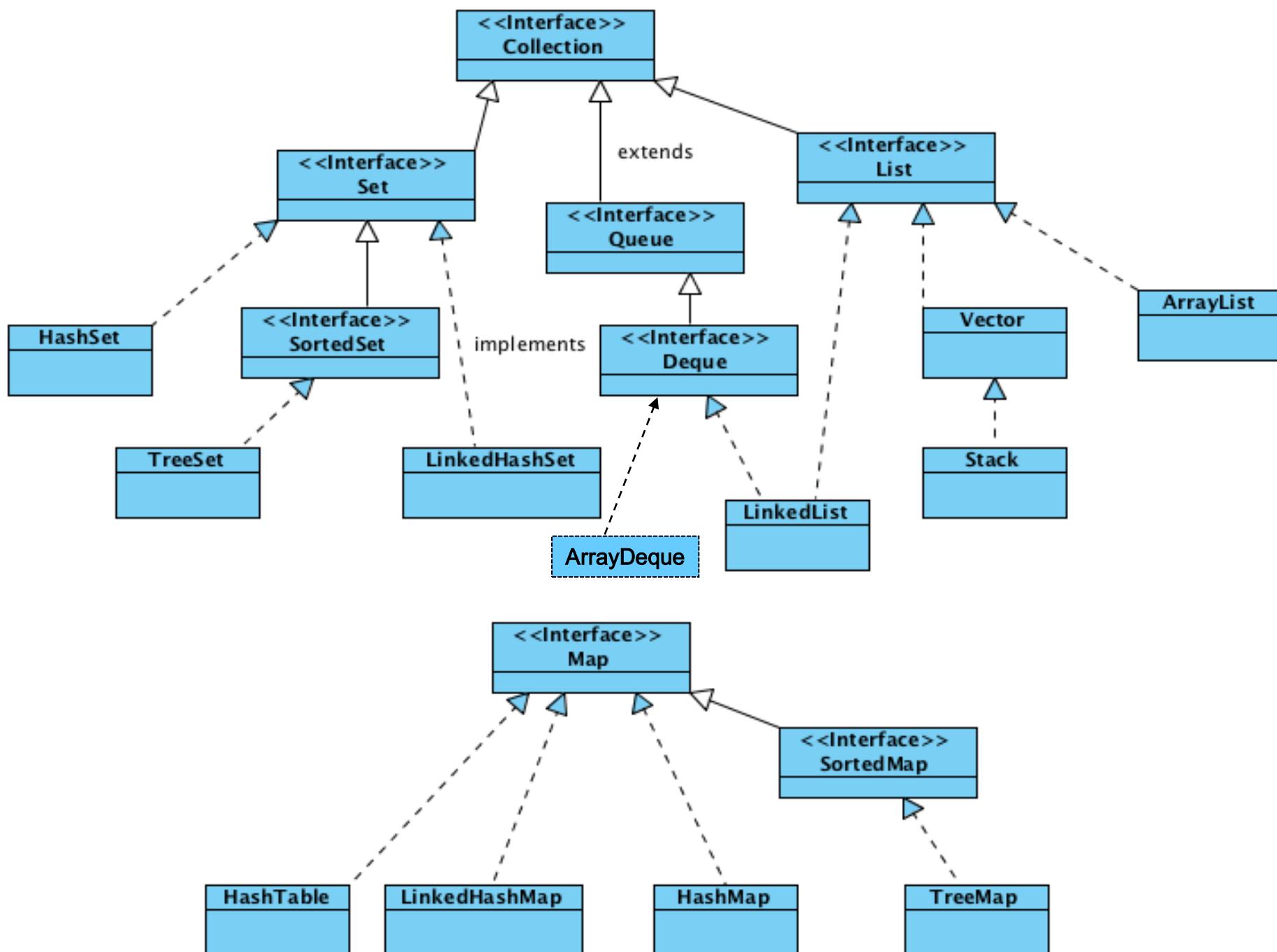
I have a dream today (1)



*Ne serait-ce pas merveilleux s'il existait un moyen facile de choisir la bonne collection pour fanfaronner devant les néophytes ?
Mais ce n'est probablement qu'un rêve...*

I have a dream today (2)

Ce n'est pas magique ☺ : il faut connaître l'API Java.



Comment choisir la bonne collection pour organiser et stocker ses données ?

- Les principales questions à se poser pour choisir une collection :
 - Les doublons sont ils autorisés ?
 - L'ordre est il important ?
 - Quel type d'ordre (insertion, comparaison) ?
 - Quels types d'accès (séquentiel, direct par indexation ou par hachage) ?
 - Quelle est la complexité des accès ?



Le TAD « LISTE » : quand la séquence compte (1)

- Une liste :
 - Admet des doublons, reconnaît les indices.
 - Est munie d'une relation d'ordre, souvent l'ordre d'insertion ou l'ordre induit par un comparateur après un tri.
- Le TAD LISTE dont le comportement est défini par l'interface List est implémenté :
 - Sous forme de liste doublement chainée par la classe [LinkedList](#).
 - Sous forme de vecteur dynamique (tableau) par les classes [ArrayList](#) et [Vector](#) (obsolète).

ArrayList

```
public class TestList {  
    public static void main (String[] args) {  
        Ouvrage l1 = new Ouvrage("La vie des chats");  
        // ... création d'autres objets Ouvrage  
        List<Ouvrage> list = new ArrayList<Ouvrage>();  
        list.add(l1);      // ...  
        list.get(2);  
        list.indexOf(l3);  
        list.remove(l2);  
        System.out.println(list);  
    }  
}
```

LinkedList

```
public class TestList {  
    public static void main (String[] args) {  
        Ouvrage l1 = new Ouvrage("La vie des chats");  
        // ...  
        List<Ouvrage> list = new LinkedList<Ouvrage>();  
        list.add(l1);      // ...  
        list.get(2);       // Déconseillé sur une LinkedList  
list.indexOf(3); // Déconseillé sur une LinkedList  
        list.remove(l2);  
        System.out.println(list);  
    }  
}
```

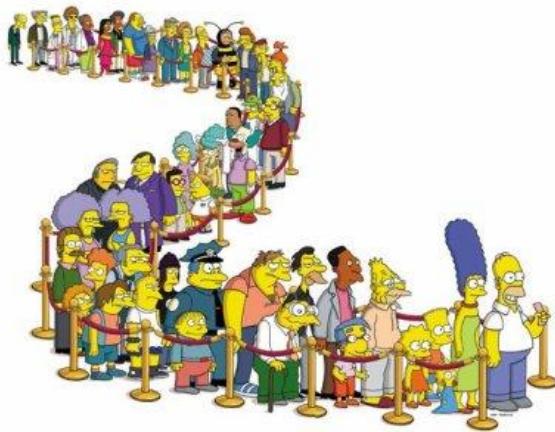
Le TAD « LISTE » : quand la séquence compte (2)

■ Comment choisir ?

- Les vecteurs (ArrayList) sont bien adaptés à l'accès direct à condition que les additions et les suppressions en milieu de liste restent limitées (récupération espace).
- Les listes chainées (LinkedList), sont bien adaptés :
 - aux accès séquentiels et pas aux accès directs qui sont plus coûteux.
 - Aux insertions/suppressions en milieu de liste sous réservent qu'elles se fassent à travers un itérateur (Cf. + loin).

Les TAD « FIFO » et « LIFO »

- L'interface **Queue** permet de définir le comportement attendu des files d'attentes (FIFO).
- L'interface **Deque** (depuis Java 6), destinée à gérer des files d'attente à double entrée permet également de gérer les piles (LIFO).
 - Elles sont implémentées :
 - Sous forme de liste chainée par la classe **LinkedList**.
 - Sous forme de vecteur dynamique (tableau) par la classe **ArrayDeque** (plus rapide que Stack (obsolète) quand LIFO et que LinkedList quand FIFO).



Le TAD « File à priorité »

- Les files à priorité, qui représentent des TAS sont implémentées par la classe **PriorityQueue** :
 - À la construction, elle permet de choisir une relation d'ordre.
 - Le type des éléments doit implémenter l'interface Comparable ou être doté d'un comparateur approprié.
 - Les éléments de la Queue sont alors ordonnés et le prélèvement d'un élément porte alors sur le « premier » au sens de cette relation (on parle du « plus prioritaire »).

Le TAD « ENSEMBLE » : quand l'unicité compte

- Un ensemble est une collection non ordonnée d'éléments qui n'admet pas les doublons.
- Cependant, pour éviter des tests d'appartenance en $O(n)$, une relation d'ordre est introduite dans les **Set** Java implémentés par **HashSet** et **TreeSet**.

HashSet

- Quand vous placez un objet dans un HashSet,
 - Il utilise le code de hachage de l'objet pour déterminer où il va le placer dans le Set.
 - Mais il compare également ce code de hachage à celui de tous les autres objets présents dans le HashSet.
 - Si aucun code ne correspond, le HashSet présume que ce nouvel objet n'est pas un doublon, sinon, il ne l'ajoute pas.

TreeSet

- TreeSet est similaire à HashSet au sens où il empêche les doublons mais il conserve également la liste triée.
 - si vous créez un TreeSet en utilisant le constructeur sans arguments, le TreeSet utilise la méthode compareTo() de chaque objet pour trier.
 - Mais vous avez la possibilité de transmettre un Comparateur au constructeur du TreeSet, pour que celui-ci l'utilise à la place.
 - Si vous n'avez pas besoin du tri, vous continuez à payer pour, avec une légère perte de performance.

compareTo() vs compare() (1)

```
class Livre implements Comparable<Object> {  
    String titre;  
    public Livre(String t) {  
        titre = t;  
    }  
    public int compareTo(Object l) {  
        Livre livre = (Livre) l;  
        return (this.titre.compareTo(livre.titre));  
    }  
}  
class ComparateurLivres implements Comparator<Livre>{  
    public int compare(Livre un, Livre deux) {  
        return ((un.titre).compareTo(deux.titre));  
    }  
}
```

compareTo() vs compare() (2)

```
public class TestTree {  
    public static void main (String[] args) {  
        Set<Livre> tree = new TreeSet<Livre>();  
        tree.add(new Livre("La vie des chats"));  
        // ...  
        tree = new TreeSet<Livre>(new ComparateurLivres());  
        tree.add(new Livre("Collections, mode d'emploi"));  
        // ...  
    }  
}
```



Constructeur
s'appuie sur
compareTo()
de classe
Livre



Constructeur
s'appuie sur
compare() de classe
ComparateurLivres

Le TAD « Tables associatives »

- Une table associative (interface **MAP**) permet de conserver et de retrouver la **valeur associée à une clé** donnée.
- Exemples :
 - Dictionnaire : mot = clé, définition = valeur.
 - Annuaire inversé : Tel = clé, nom+adresse = valeur.
- Implémentations :
 - Sous forme de table de hachage : classe **HashMap**,
 - Sous forme d'arbre binaire : classe **TreeMap**.
- Rq : les **HashSet** et **TreeSet** sont des cas particuliers de Map dont la valeur est vide.



Table associative implémentée par un HashMap

```
import java.util.*;  
public class TestMap {  
    public static void main(String[] args) {  
        Map<String, Integer> scores = new HashMap<String, Integer>();  
        scores.put("Kathy", 42);  
        scores.put("Bert", 343);  
        scores.put("Skyler", 420);  
        System.out.println(scores);  
        System.out.println(scores.get("Bert")); // affiche 343  
    }  
}
```

I have a dream today (1)

Ne serait-ce pas merveilleux s'il existait des algorithmes déjà implémentées, de manipulation des collections ?

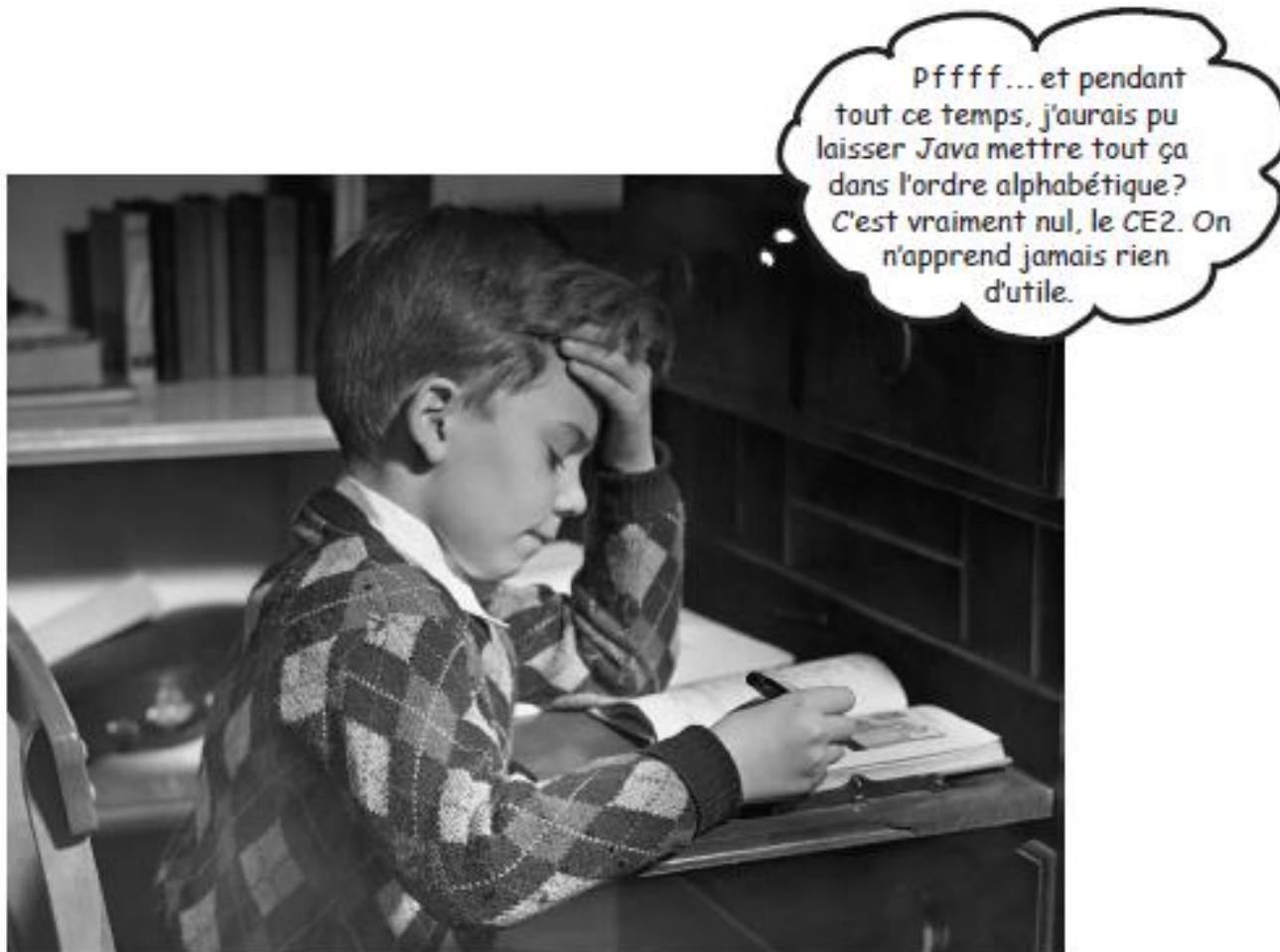
Mais ce n'est probablement qu'un rêve...



I have a dream today (2)



De tels algos
sont définis dans
la classe
Collections.





Algorithmes applicables sur les collections

- La classe **Collections** fournit, sous forme de méthodes statiques, des méthodes utilitaires générales applicables aux collections, notamment :
 - recherche de maximum ou de minimum,
 - tri et mélange aléatoire,
 - recherche binaire,
 - Copie, etc.
- Ces méthodes disposent d'arguments d'un type interface **Collection** ou **List**.
- Elles se basent sur l'ordre induit par **compareTo()** ou par un **comparateur**.
- Rq : classe **Arrays** fournit services équivalents sur tableaux.



I have a dream today (1)

Ne serait-ce pas merveilleux s'il
Existait un mécanisme permettant
de parcourir une séquence sans en
exposer ni l'organisation ni
la représentation ?
Mais ce n'est probablement qu'un
rêve...



I have a dream today (2)



Un tel objet
existe :
c'est un
itérateur.

Comment parcourir une collection ?

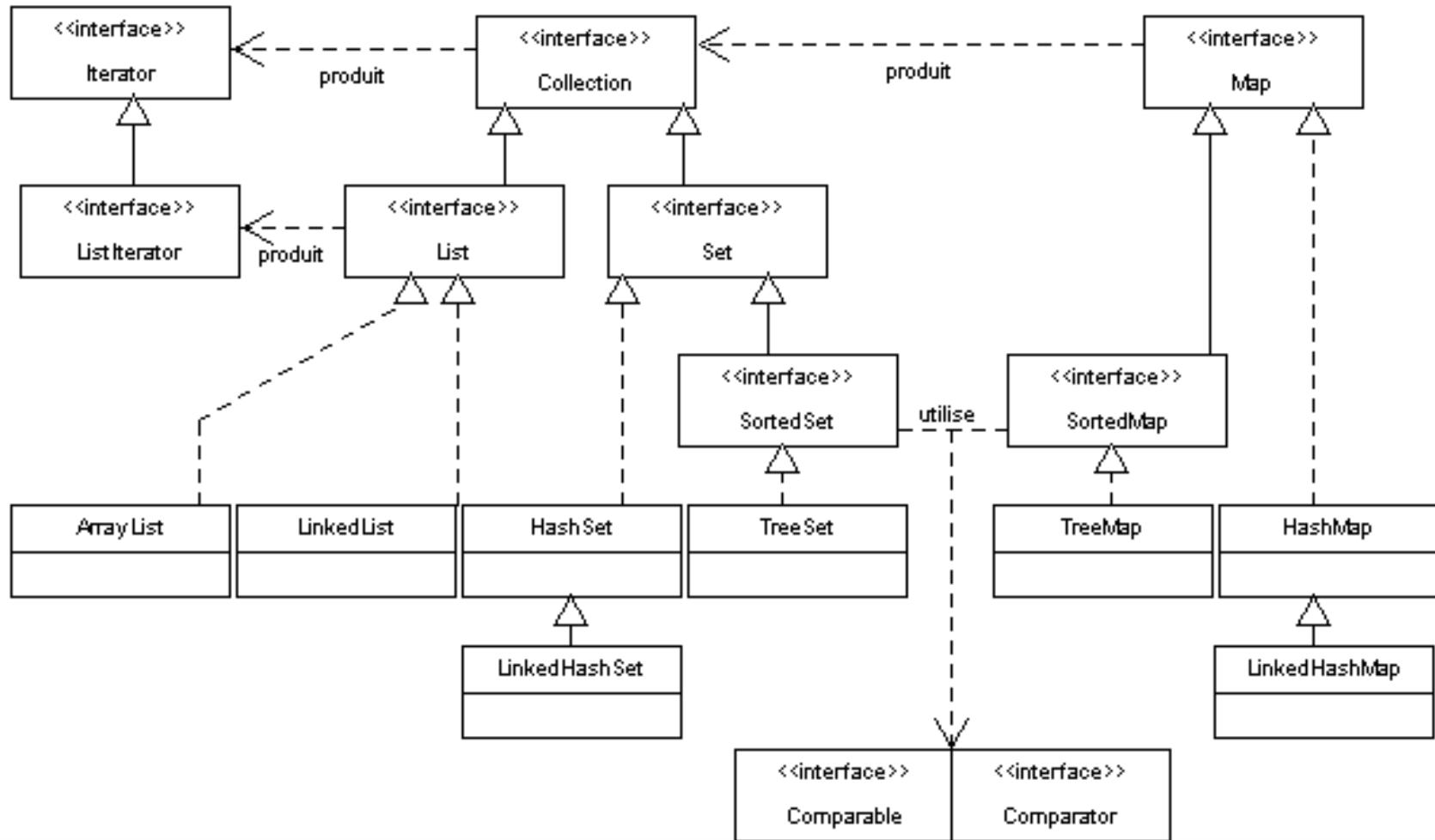
- La classe `AbstractCollection` est capable d'implémenter les méthodes `containsAll()`, `toString()`, etc. sans pourtant connaître l'organisation (liste, ensemble) ni le mode de stockage (tableau, chaînage dynamique) de la collection.
- Il est cependant inconcevable d'imaginer tester le type de l'objet pour utiliser la bonne méthode avec un `switch` :

```
public String toString() {  
    Si objet instance de Vector ... return concaténation des elementAt(i)  
    Si objet instance de ArrayList... return concaténation des get(i)  
    etc.  
}
```

Qu'est-ce qu'un itérateur ?

- Java met en œuvre le Design Pattern **Iterator** :
 - Un **itérateur** est un objet qui parcourt une structure composite et en vérifie chaque élément en une seule fois.
 - L'interface **Iterator** spécifie le comportement d'un itérateur unidirectionnel, l'interface **ListIterator** le comportement d'un itérateur bidirectionnel dans des listes doublement chainées.
 - Dans chaque classe instanciable (`ArrayList...`), il a suffit de définir la méthode `iterator()` (ou `listIterator()`).

Mise en œuvre du DP Iterator dans le langage Java



Interface Iterator

- Interface **Iterator** :

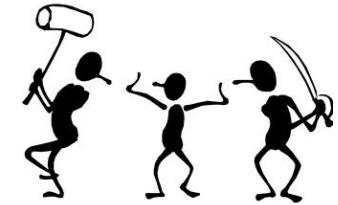
```
Public interface Iterator {  
    public boolean hasNext();  
    public object next(); // retourne l'élément courant et se  
                        // positionne sur le suivant  
    public void remove(); // supprime l'élément courant  
}
```

- On distingue la **position** courante
de **l'élément** courant.



Comment utiliser un itérateur pour parcourir une collection ?

- Soit les collections définies ainsi :
 - Collection collection = new ArrayList();
 - List<String> liste = new LinkedList<String>();
- Utilisation d'un itérateur :
 - `for (Iterator ite = collection.iterator(); ite.hasNext();)
System.out.println(ite.next().toString());`
 - `for (ListIterator ite = liste.listIterator(); ite.hasNext();)
System.out.println(ite.next());`



Comment la boucle for-each s'appuie-t-elle sur un itérateur ?

- Soit les collections définies ainsi :
 - Collection collection = new ArrayList();
 - List<String> liste = new LinkedList<String>();
- Utilisation de la boucle **for-each** : utiliser boucle *for* pour itérer n'importe quelle classe qui implémente l'interface **Iterable**.
 - **for (Object valeur : collection)**
System.out.println(valeur);
 - **for (String valeur : liste)**
System.out.println(valeur);

Comment parcourir les éléments d'une Map ?

- Une table associative (Map) **n'implémente pas l'interface Iterable** et ne peut donc pas être parcourue directement par un itérateur.
- Il existe une méthode **entrySet()** retournant une vue de la table associative sous la forme d'un Set.
- C'est cet ensemble qui sera parcouru par l'itérateur (les classes TreeSet et HashSet implémentent interface Iterable).
- La boucle for doit déclarer un objet **Map.Entry** représentant une paire clé/valeur.

Comment parcourir les éléments d'une Map avec une boucle for ?

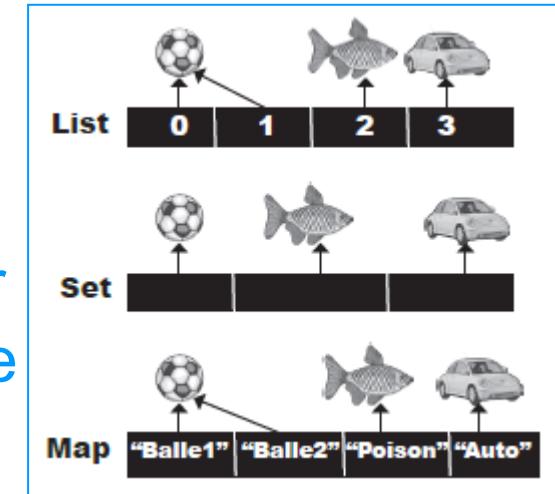
- Map<String, Integer> tableAssociation;
tableAssociation = new TreeMap<String, Integer>();
Set< Map.Entry<String, Integer> > set =
tableAssociation.entrySet();
// remplissage tableAssociation ...
- System.out.println(set); ou bien
- for (Map.Entry<String, Integer> e : set)
{ System.out.println(e.getKey() + "=" + e.getValue()); }
- set référence une vue sur la Map : si la Map évolue,
l'affichage de l'ensemble évolue aussi.

Kit de survie



Ce qu'il faut retenir (1)

- Une List sait où se trouve quelque chose dans la liste grâce à son indice. Vous pouvez avoir plusieurs éléments référençant le même objet.
- Dans un Set vous ne pouvez pas avoir plus d'un élément référençant le même objet.
- Une Map connaît la valeur associée à une clé donnée. Vous pouvez avoir deux clés qui référencent la même valeur, mais vous ne pouvez pas avoir de clés dupliquées.



Ce qu'il faut retenir (2)

- Pour placer quelque chose dans une collection, utilisez add().
- Pour supprimer quelque chose d'une collection utilisez remove().
- Pour savoir où se trouve quelque chose (et s'il s'y trouve) dans une collection, utilisez indexOf().
- Pour savoir si une collection est vide, utilisez isEmpty().
- Pour connaître la taille (le nombre d'éléments) d'une collection, utilisez la méthode size().
- Pour parcourir une collection, utilisez un itérateur : iterator() ou une boucle for-each.
- En fonction de la collection choisie, il existe de nombreuses méthodes spécifiques.

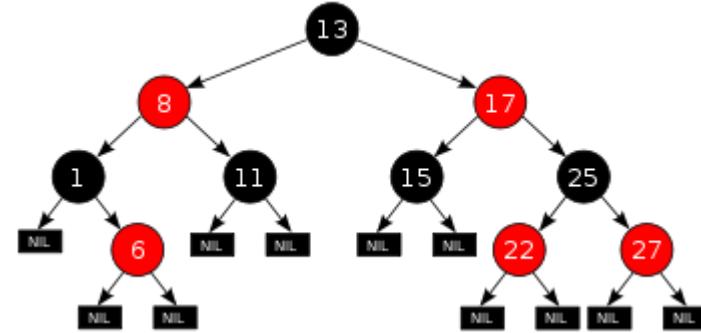
Ce qu'il faut retenir (2)

- **LinkedList :**
 - Est implémentée par une liste chainée bidirectionnelle et parcourue par un ListIterator.
 - Parcours/recherche ième élément : $O(n)$.
 - Insertion/suppression, avec Itérateur : $O(1)$.
 - Insertion/suppression après recherche bonne place : $O(n)$.
- **ArrayList :**
 - Est implémentée dans un tableau Recherche ième élément, insertion à la fin : $O(1)$.
 - Parcours/insertion/suppression à la ième position : $O(n)$.

Ce qu'il faut retenir (3)



- HashSet :
 - Recourt à une technique de hachage pour ranger (et accéder) les (aux) éléments (basée sur méthode hashCode()).
 - Toutes opérations en O(1).
- TreeSet :
 - Utilise un arbre binaire bicolore pour ordonner complètement les éléments, qui s'appuie sur méthode compareTo() ou sur un comparateur.
 - Toutes opérations en O(Log N).



Ce qu'il faut retenir (4)

- **HashMap :**
 - Ordonnancement à partir du code de hachage des objets formant la clé.
 - Toutes opérations en O(1).
- **TreeMap :**
 - Ordonnancement à partir de la relation d'ordre induite par compareTo() ou par un comparateur fixé à la construction.
 - Toutes opérations en O(Log N).
 - Accès moins rapides mais TreeMap (comme TreeSet) toujours ordonnés selon leur clé.



Chapitre 5 : Programmes à interfaces graphiques Programmation évènementielle

3IRC – 4ETI
Version 2013/2014

F. PERRIN

membre de UNIVERSITÉ DE LYON

CPE
ÉCOLE SUPÉRIEURE
DE CHIMIE PHYSIQUE ÉLECTRONIQUE
DE LYON

Quelques composants graphiques supplémentaires sont donnés en Annexe 3 disponible sur le e-campus.



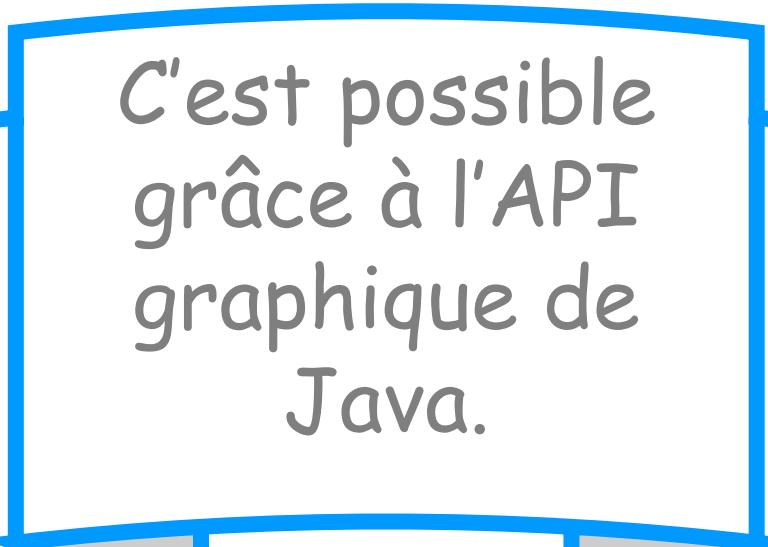
I have a dream today (1)

Ne serait-ce pas merveilleux s'il existait un moyen d'écrire des Interfaces Homme Machine (GUI) souples et conviviales et que le programme réagisse aux actions de l'utilisateur?

Mais ce n'est probablement qu'un rêve...



I have a dream today (2)



C'est possible
grâce à l'API
graphique de
Java.

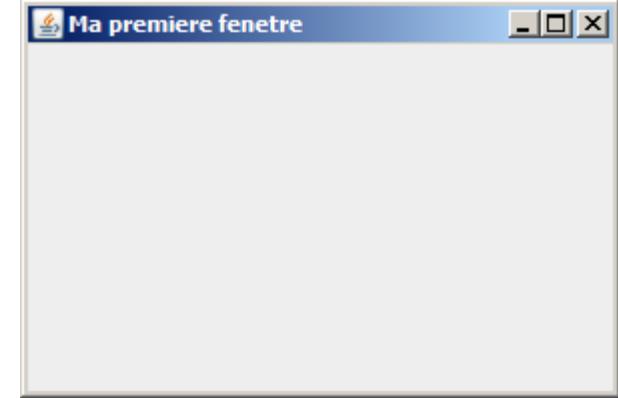


Principe de l'événementiel

- Tout commence par une **fenêtre** JFrame.
- Une fois que vous avez une JFrame, vous pouvez y placer des **widgets**.
- Il existe de nombreux composants Swing dans le package javax.swing. Les plus courants sont JButton, JRadioButton, JCheckBox, JLabel, JList, JScrollPane, JSlider, JTextArea, JTextField et JTable.
- Ensuite, il faut **écouter** les évènements associés aux l'actions de l'utilisateur et y **répondre**.
- Et enfin, il faut **afficher** la fenêtre et ainsi l'utilisateur pourra commencer à **agir** sur les différents widgets.

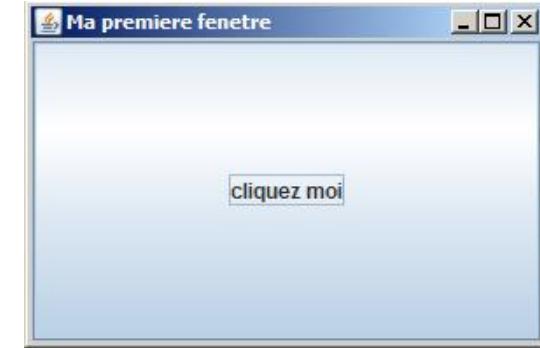
Création fenêtre graphique

```
import javax.swing.*;  
public class SimpleIhm {  
    JFrame cadre;  
    public static void main (String[] args) {  
        SimpleIhm ihm = new SimpleIhm();  
        ihm.go();  
    }  
    public void go() {  
        cadre = new JFrame();  
        cadre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        cadre.setSize(300,200);  
        cadre.setTitle ("Ma premiere fenetre") ;  
        cadre.setVisible(true);  
    }  
}
```



Ajout d'un bouton dans une fenêtre

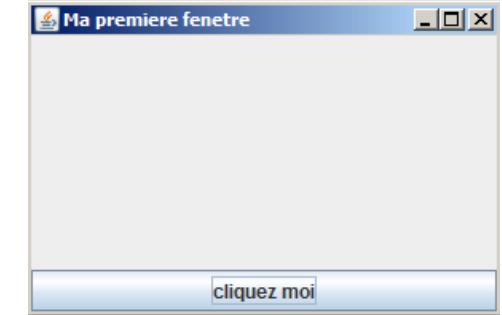
```
public class SimpleJhm {  
    JFrame cadre;  
    JButton bouton;  
    public static void main (String[] args) {  
        // ...  
    }  
    public void go() {  
        cadre = new JFrame();  
        // ...  
        bouton = new JButton("cliquez moi");  
        cadre.getContentPane().add(bouton);  
        // ...  
    }  
}
```



Ouah! Ce bouton est gigantesque.

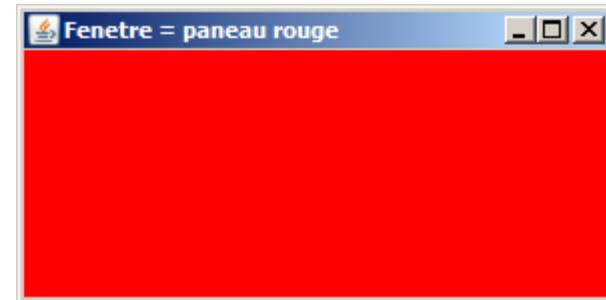
Ajout d'un bouton dans une fenêtre en utilisant un Gestionnaires d'agencement.

```
public class SimpleJhm
    JFrame cadre;
    JButton bouton;
    public static void main (String[] args) {
        // ...
    }
    public void go() {
        cadre = new JFrame();
        // ...
        bouton = new JButton("cliquez moi");
        cadre.getContentPane().add(bouton, "South");
        // ...
    }
}
```



Ajout d'un panneau rouge dans une fenêtre

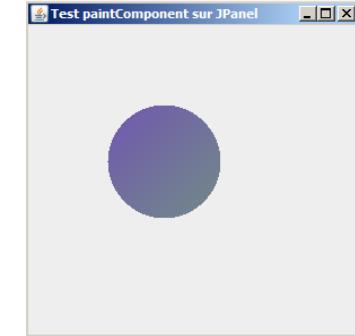
```
public class SimpleJhm2C {  
    JFrame cadre;  
    JPanel panneau;  
    public static void main (String[] args) {  
        // ...  
    }  
    public void go() {  
        panneau = new JPanel ();  
        cadre = new JFrame();  
        cadre.getContentPane().add(panneau);  
        panneau.setBackground(Color.red) ; // fond rouge  
        // ...  
    }  
}
```





Dessin dans un JPanel

```
public class SimpleJhm2C {  
    JFrame cadre;  
    JPanel panneau;  
    public static void main (String[] args) { // ...  
    }  
    public void go() {  
        panneau = new MonPanneau(); cadre = new JFrame();  
        cadre.getContentPane().add(panneau);  
    }  
}  
class MonPanneau extends JPanel {  
    public void paintComponent(Graphics g) {  
        super.paintComponent(g);  
        g.fillOval(70,70,100,100);  
    }  
}
```

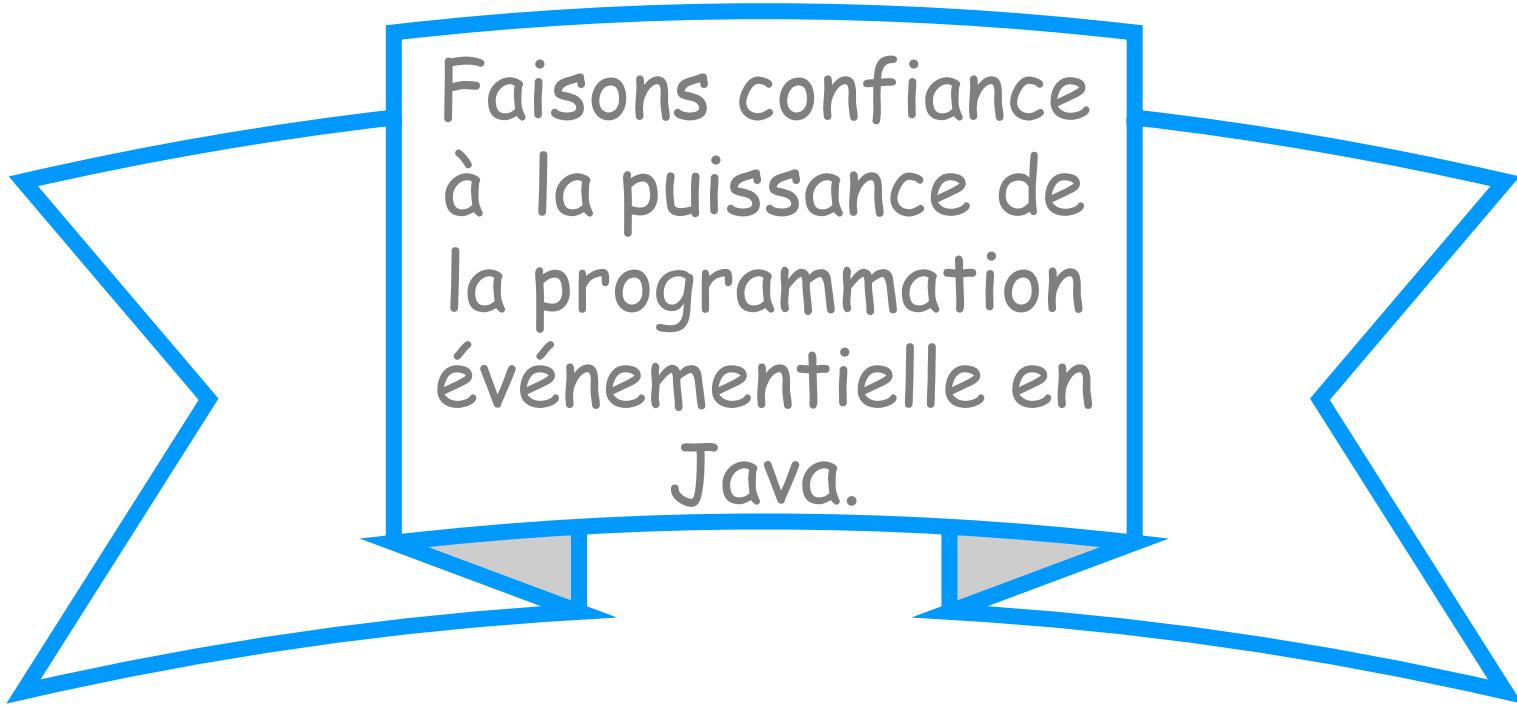


I have a dream today (1)

Ne serait-ce pas merveilleux s'il existait un moyen pour que le programme réagisse aux actions de l'utilisateur sur l'IHM et non pas s'exécute en séquentiel du début à la fin de la fonction main() ?
Mais ce n'est probablement qu'un rêve...



I have a dream today (2)



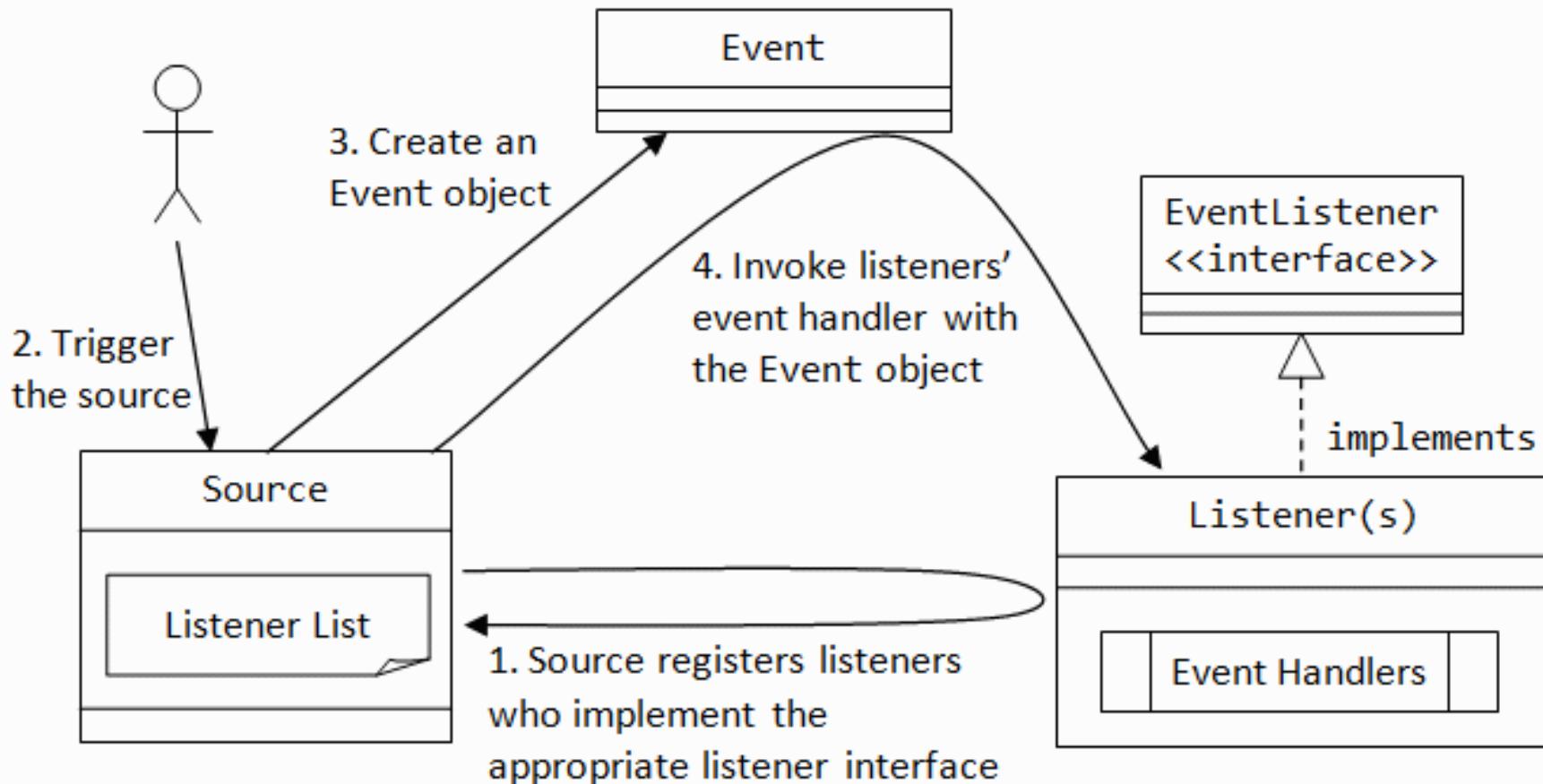
Faisons confiance
à la puissance de
la programmation
événementielle en
Java.

Quels sont les concepts de la programmation événementielle

- **Événement** : clic de souris, sélection menu...
- **Catégorie d'événement** (ou type d'événement) : MouseEvent, ActionEvent ...
- **Source événement** : objet qui lui a donné naissance : fenêtre, bouton, item de menu...
- **Écouteur** : pour traiter un événement, on associe à la source un objet (appelé un écouteur) dont la classe implémente une interface correspondant à une catégorie d'événement.



Mise en œuvre du DP Observer pour la gestion des événements dans Java



Comment écouter un bouton ? (1)

- Un bouton ne peut déclencher qu'un événement correspondant à l'action de l'utilisateur sur ce bouton.
- Cet événement est l'unique événement de la catégorie **Action** ; il faudra donc :
 - Créer un écouteur objet d'une classe qui implémente **ActionListener** ; cette dernière ne contient qu'une seule méthode **actionPerformed()**
 - Associer cet écouteur au bouton par la méthode **addActionListener()**.



Comment écouter un bouton ? (2)

```
public class SimpleIhm1B implements ActionListener {  
    // ...  
    public void go() {  
        // ...  
        bouton = new JButton("cliquez moi");  
        bouton.addActionListener(this); ←  
        cadre.getContentPane().add(bouton, "South");  
        // ...  
    }  
    public void actionPerformed(ActionEvent event) {  
        bouton.setText("J'ai été cliqué!");  
    }  
}
```

Le bouton est
écouté par
l'objet
SimpleIHM1B



Gestion des événements en général

- A une **catégorie d'événements Xxx**, on associera toujours un objet **écouteur des événements** (de type **XxxEvent**), par une méthode nommée **addXxxListener()**.
- Chaque fois qu'une catégorie disposera de plusieurs méthodes, on pourra :
 - Redéfinir toutes les méthodes de l'interface **XxxListener** (certaines seront vides), ou
 - Faire appel à une classe dérivée d'une classe **XxxAdapter** et ne fournir que les méthodes utiles.
- L'objet écouteur pourra être n'importe quel objet, et en particulier l'objet source lui-même.
- Un programme ne considère que les événements qui l'intéressent, les autres subissent un traitement par défaut.

Comment écouter 2 composants bouton avec 1 action distincte pour chacun ? (1)

```
public class SimpleIhm4C implements ActionListener{
```

```
    // ...
```

```
    private JButton monBouton1, monBouton2 ;
```

```
    JLabel etiquette; JPanel panneau;
```

```
    public void go() {
```

```
        // ...
```

```
        monBouton1.addActionListener(this);
```

```
        monBouton2.addActionListener(this);
```

```
}
```

```
    public void actionPerformed(ActionEvent event) {
```

```
        if (event.getSource() == bouton1) {
```

```
            cadre.repaint();
```

```
        } else {
```

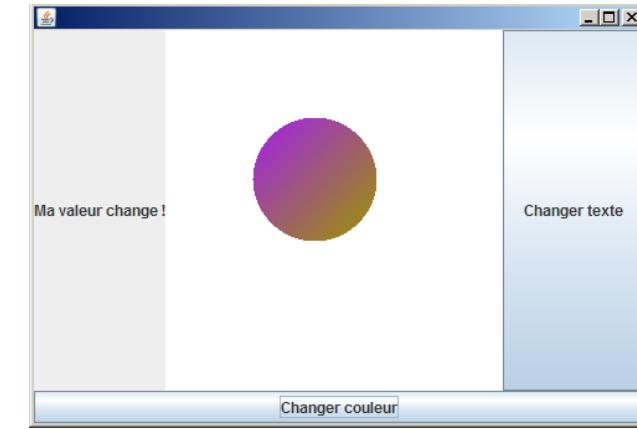
```
            etiquette.setText("Ma valeur change !");
```

```
}
```

Pas très objet !

```
}
```

```
}
```



Comment écouter 2 composants bouton avec 1 action distincte pour chacun ? (2)

```
public class SimpleIhm4C implements ActionListener {  
    // ...  
    public void go() {  
        // ...  
        monBouton1.addActionListener(new boutonCouleurListener());  
        monBouton2.addActionListener(new LabelButtonListener());  
    }  
}  
class boutonCouleurListener implements ActionListener {  
    public void actionPerformed(ActionEvent event) {  
        cadre.repaint();  
    }  
}  
class LabelButtonListener implements ActionListener {  
    public void actionPerformed(ActionEvent event) {  
        etiquette.setText("Ma valeur change !!");  
    }  
}
```

KO : pas de référence à variable 'cadre'

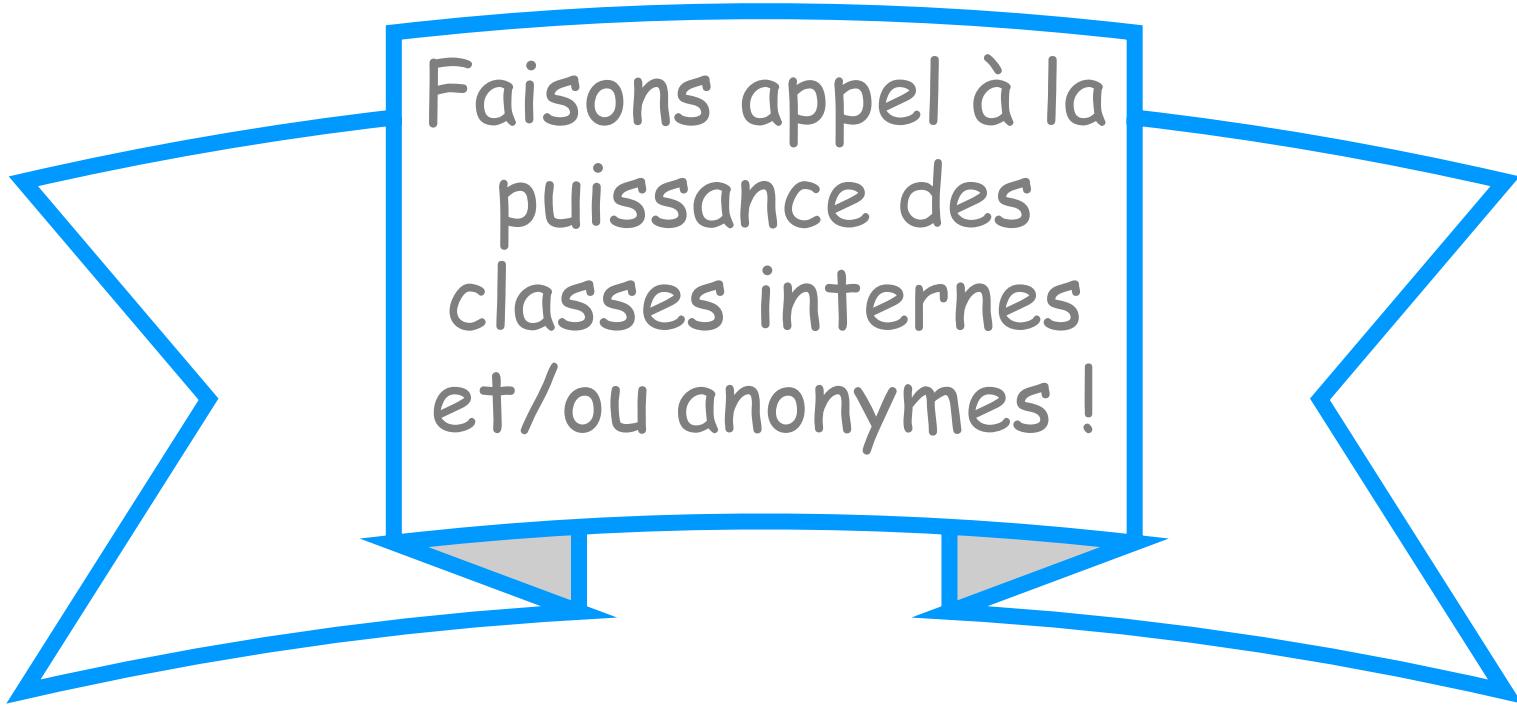
KO : pas de référence à variable 'etiquette'

I have a dream today (1)

Ne serait-ce pas merveilleux si on pouvait avoir deux classes écouteurs différentes, mais qu'elles puissent accéder aux variables d'instance de la classe principale, presque comme si elles leur appartenaient. Comme ça, on aurait le meilleur des deux mondes.
Oui, ce serait la solution rêvée.
Mais ce n'est probablement qu'un rêve.



I have a dream today (2)



Faisons appel à la puissance des classes internes et/ou anonymes !



Comment écouter 2 composants bouton avec 1 action distincte pour chacun ? – meilleure approche –

```
public class SimpleIhm4C
{
    // ...
    public void go() {
        // ...
        monBouton1.addActionListener (new ActionListener() {
            public void actionPerformed (ActionEvent ev) {
                cadre.repaint();
            }
        });
        monBouton2.addActionListener (new LabelButtonListener ());
    }

    private class LabelButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            etiquette.setText("Ma valeur change !!");
        }
    }
}
```

Classe interne anonyme

Classe interne



Kit de survie



Ce qu'il faut retenir (1)

- Pour créer une IHM, commencez par une fenêtre, généralement un cadre de type JFrame :
`JFrame cadre = new JFrame();`
- Pour afficher le JFrame, vous devez définir sa taille et le rendre visible :
`cadre.setSize(300,300);`
`cadre.setVisible(true);`
- Vous pouvez ajouter des widgets (boutons, champs de texte, etc.) au JFrame avec :
`frame.getContentPane().add(bouton);`
- Contrairement à beaucoup d'autres composants, on ne peut pas ajouter d'éléments directement à un JFrame. On les insère dans son panneau de contenu `frame.getContentPane()`.
- Pour les positionner correctement on utilise un Layout Manager.

Ce qu'il faut retenir (2)

- Les gestionnaires d'agencement (Layout Manager) contrôlent la taille et le placement de composants imbriqués dans d'autres composants.
- Quand vous ajoutez un composant à un autre composant, le composant ajouté est contrôlé par le gestionnaire d'agencement du composant d'arrière-plan.
- Un gestionnaire d'agencement demande aux composants leur taille préférée avant de prendre une décision sur le placement. Selon sa politique, il peut respecter tout ou partie de leurs désirs ou les ignorer.
- BorderLayout est le gestionnaire par défaut des JFrame et FlowLayout le gestionnaire par défaut des JPanel.
- Si vous voulez un autre gestionnaire d'agencement, vous devez appeler setLayout().

Ce qu'il faut retenir (3)

- Vous pouvez dessiner des graphismes en 2D directement sur un widget.
- Vous pouvez directement placer un fichier .gif ou .jpeg sur un widget.
- Pour créer vos propres graphismes (y compris les .gif et les .jpeg), créez une sous-classe de JPanel et redéfinissez la méthode paintComponent().
- La méthode paintComponent() est appelée par le système.
VOUS NE L'APPELEZ JAMAIS VOUS-MÊME directement.
- L'argument de paintComponent() est un objet Graphics qui vous fournit la surface sur laquelle dessiner. Vous ne pouvez pas construire cet objet vous-même.

Ce qu'il faut retenir (4)

- Les méthodes usuelles à appeler sur un objet Graphics (le paramètre de paintComponent ()) sont :
`g.setColor(Color.blue); g.fillRect(20,50,100,120);`
- Pour un .jpg,
 - construisez une Image comme suit :
`Image image = new ImageIcon("catzilla.jpg").getImage();`
 - et affichez l'image avec : `g.drawImage(image,3,4,this);`
- L'objet référencé par le paramètre de paintComponent() est en réalité une instance de la classe Graphics2D. Cette classe possède de nombreuses méthodes, notamment :
`fill3DRect(), draw3DRect(), rotate(), scale(), shear(), transform()`
- Pour pouvoir invoquer les méthodes de la classe Graphics2D, vous devez convertir le paramètre et transformer l'objet Graphics en objet Graphics2D : `Graphics2D g2d = (Graphics2D) g;`

Ce qu'il faut retenir (5)

- Pour savoir quand l'utilisateur clique sur un bouton (ou effectue tout autre action dans l'interface graphique), vous devez écouter un événement.
- Pour pouvoir écouter un événement, vous devez faire part de votre intérêt à sa source. Une source d'événements est l'élément (bouton, case à cocher, etc.) qui déclenche l'événement.
- L'interface écouteur donne à la source d'événements un moyen de vous rappeler, parce qu'elle définit les méthodes que la source appelle quand un événement se produit.
- Pour s'enregistrer auprès d'une source, appelez sa méthode d'enregistrement. Celle-ci prend toujours la forme :
`add<Type d'événement>Listener.`

Par exemple, pour les ActionEvents, d'un bouton, appelez :
`bouton.addActionListener(this);`

Ce qu'il faut retenir (6)

- Implémentez l'interface écouteur en écrivant toutes ses méthodes de gestion des événements. Placez le code dans la méthode de rappel de l'écouteur.

Pour les ActionEvents, la méthode est actionPerformed() - Ex :

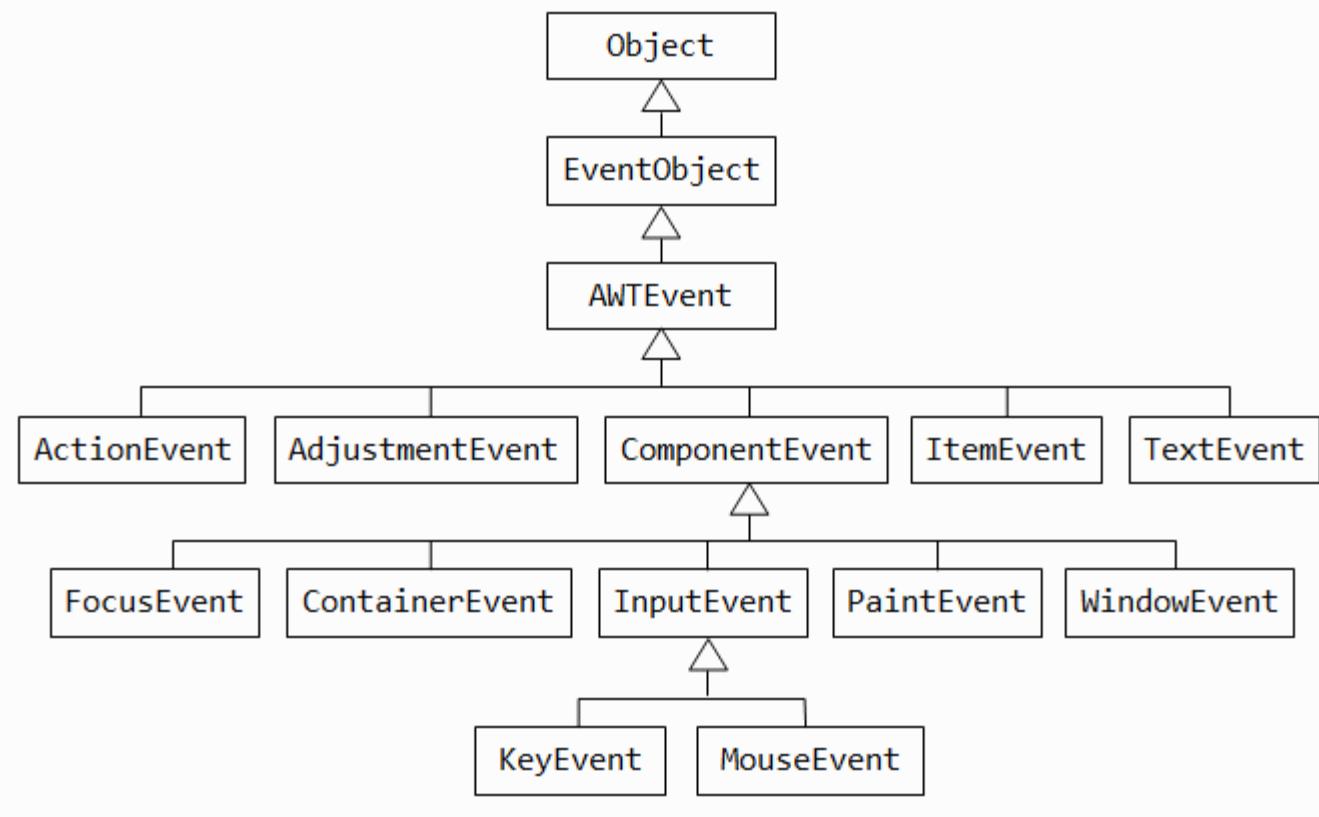
```
public void actionPerformed(ActionEvent event) {  
    bouton.setText("c'est cliqué !");  
}
```

- L'objet event transmis à la méthode de gestion de l'événement contient des informations sur celui-ci, notamment sa source.



Hiérarchie des catégories d'évènements

- http://www3.ntu.edu.sg/home/ehchua/programming/java/J4a_GUI.html -



Dans package `java.awt.event`



Conclusion :

Et s'il n'y avait que peu de choses à retenir
pour faire une bonne conception objet ?

3IRC – 4ETI
Version 2013/2014

F. PERRIN

membre de UNIVERSITÉ DE LYON

CPE LYON
ÉCOLE SUPÉRIEURE
DE CHIMIE PHYSIQUE ÉLECTRONIQUE
DE LYON

I have a dream today (1)



Ne serait-ce pas merveilleux si c'était la fin du poly ? S'il n'y avait plus d'explications, ni d'exercices, ni d'interview, ni rien ?

Mais ce n'est probablement qu'un rêve...

I have a dream today (2)

*Résumons juste ce que nous
avons appris.*



Notre objectif était
d'apprendre à concevoir
et développer des
applications souples,
évolutives, facile à
maintenir.



Gardez bien à l'esprit
que pour réaliser une
bonne conception, il faut
respecter quelques
principes :



Pour être faciles à maintenir, vos classes ne doivent avoir qu'une seule responsabilité et ne pas se disperser en de multiples traitements.

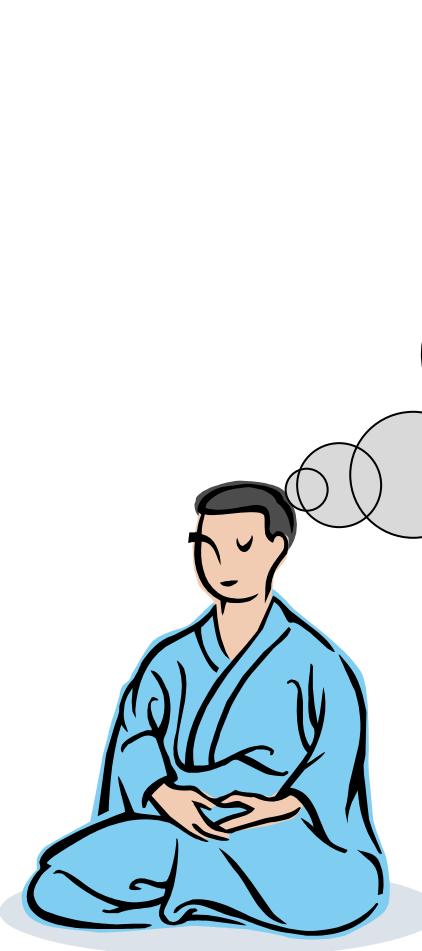


Evitez en particulier les objets « Dieu » qui ont de nombreuses responsabilités, alors que les autres ne contiennent que des données et accesseurs.

⇒ redistribuez les responsabilités.



Pour être évolutive sans nuire à la maintenabilité, votre application doit être ouverte aux extensions et fermée aux modifications.



Par exemple, définissez des classes polymorphes et ajoutez une nouvelle classe dérivée plutôt que tester le type d'un objet pour savoir ce qu'il sait faire.



N'hésitez pas non plus à déléguer des responsabilités à d'autres classes plutôt que dériver et modifier à l'infini.





CONTACT

Domaine Scientifique de la Doua
43, bd du 11 Novembre 1918 - Bâtiment Hubert Curien
B.P. 2077 - 69616 Villeurbanne cedex - France

Tél. : (33) 04 72 43 17 00
Fax : (33) 04 72 43 16 84

www.cpe.fr

Tél. : (33) 04 72 44 79 13
francoise.perrin@cpe.fr

membre de  UNIVERSITÉ DE LYON


LYON
CPE
ÉCOLE SUPÉRIEURE
DE CHIMIE PHYSIQUE ÉLECTRONIQUE
DE LYON

Diagramme simplifié du sujet de TP

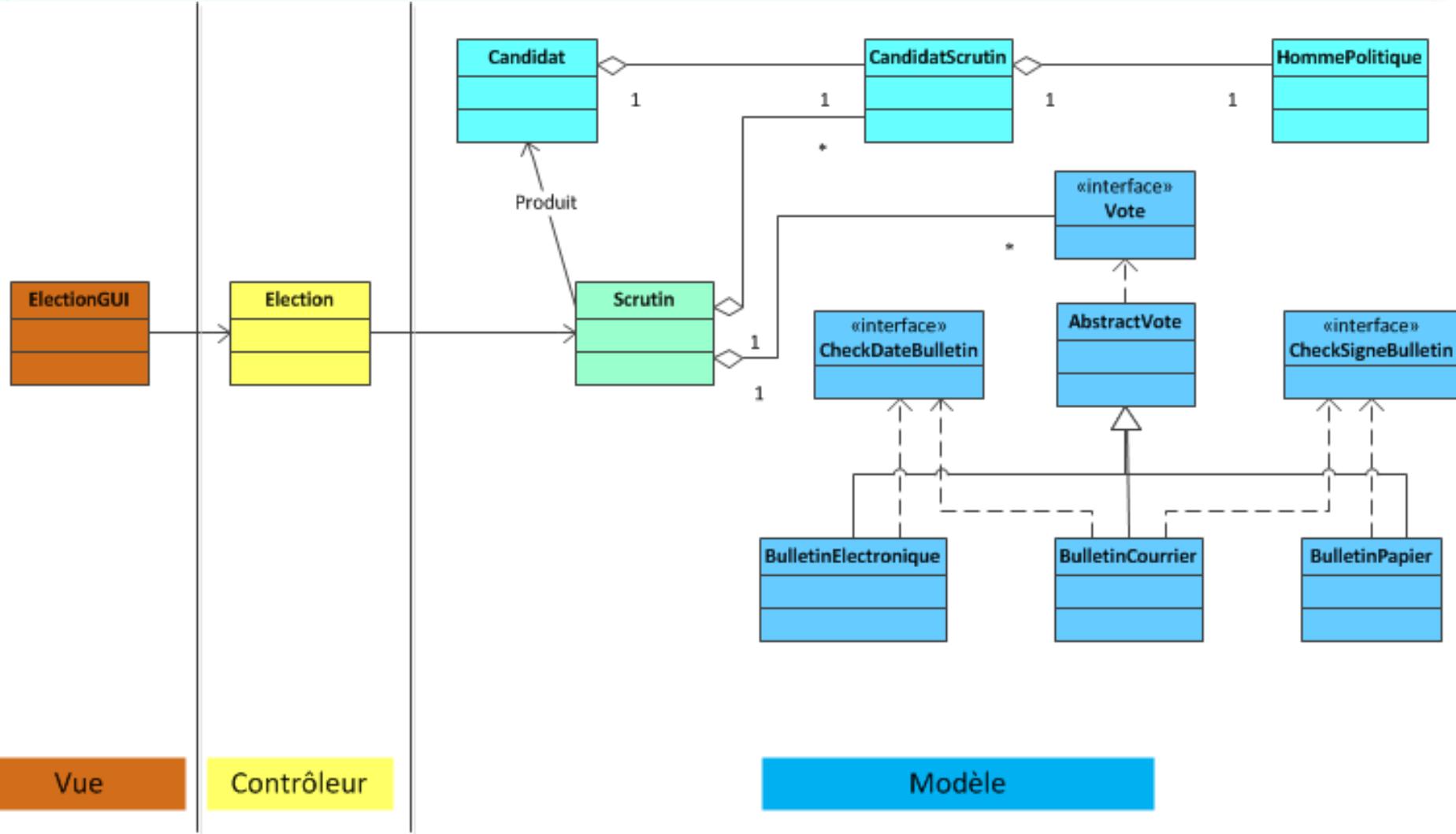
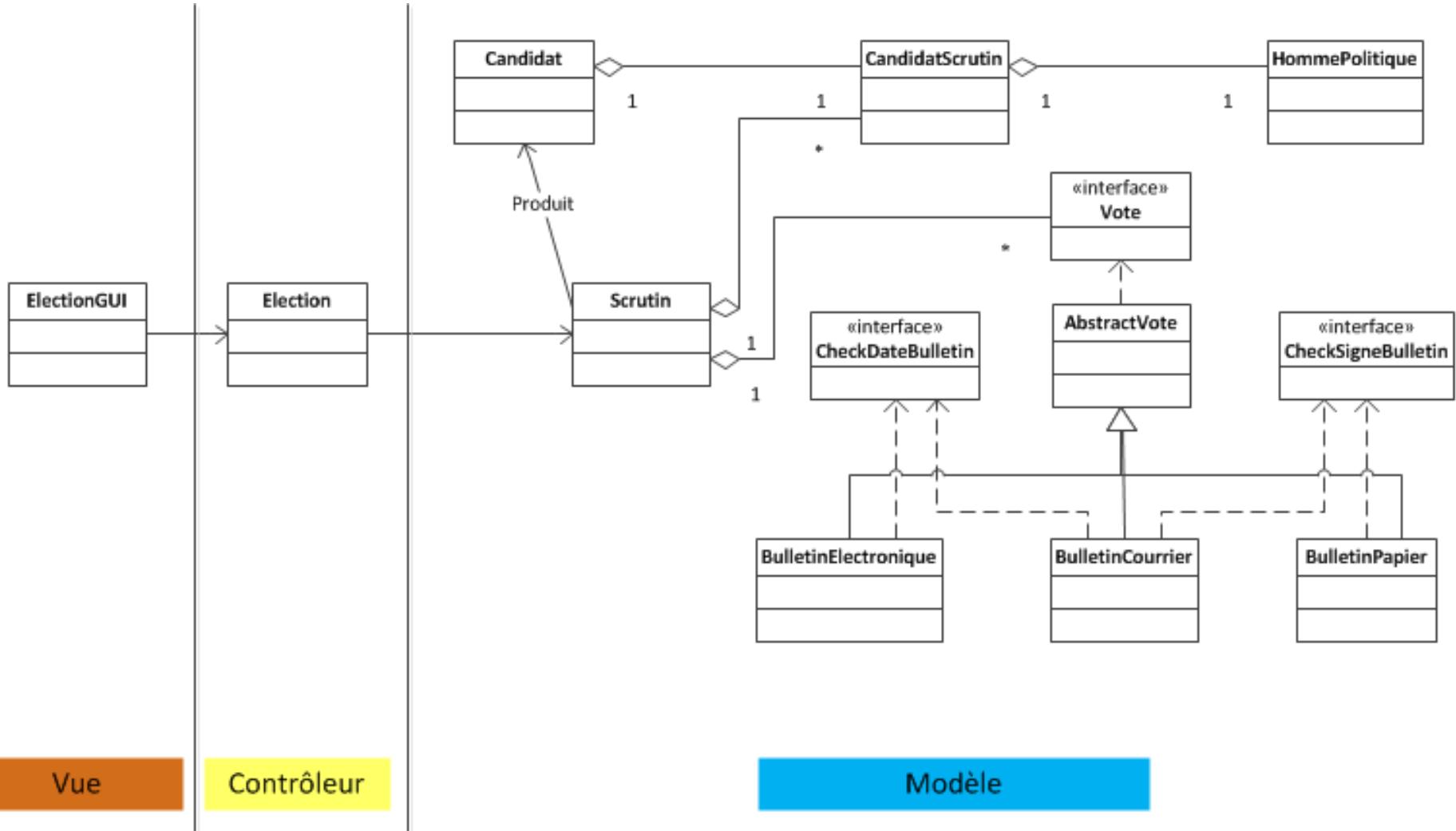


Diagramme simplifié du sujet de TP



Vue

Contrôleur

Modèle

Insertion dans une Map

```
private static Map<Civilite, List<String>> loadMap(  
                                List<Candidat> candidats) {  
    Map<Civilite, List<String>> map =  
        new TreeMap<Civilite, List<String>>();  
    List<String> hommePolitiques = null;  
    for (Candidat candidat : candidats) {  
        hommePolitiques = map.get(candidat.getCivilite());  
        //si cette cle n'existe pas dans la Map, on cree un nouvel element  
        if(hommePolitiques==null){  
            hommePolitiques = new LinkedList<String>(); }  
        //et on inclut l'HommePolitique enveloppé dans le Candidat ds la liste  
        hommePolitiques.add(candidat.getNom());  
        //puis l'element dans la Map  
        map.put(candidat.getCivilite(), hommePolitiques);  
    }  
    return map;  
}
```

Suppression dans liste : attention à l'itérateur

```
for (Candidat element : candidats) {  
    if (element.getPourCentVoix() < 20)  
        candidatClones.remove(element);  
}
```

```
for (Iterator<Candidat> it = candidatClones.iterator();  
     it.hasNext();) {  
    Candidat element = it.next();  
    if (element.getPourCentVoix() < 20)  
        it.remove();  
}
```