

Objets

Vision bas niveau (en Java)

Objets

Compléments en P00
Aldric Degorre

Vision haut niveau	
Introduction	
Généralités	
Style	
Objets et classes	
Objets et classes	
Membres et constructeurs	
Objets et classes	
Encapsulation	
Types imbriqués	
Types et polymorphisme	
Héritage	
Généricité	
Concurrence	
Interfaces graphiques	
Gestion des erreurs et exceptions	
Analyse	
<p>Un objet est "juste" un nœud dans le graph de communication qui se déploie quand on exécute un programme OO.</p> <p>Il est caractérisé par une certaine interface¹ de communication.</p> <p>Un objet a un état (modifiable ou non), en grande partie caché vis-à-vis des autres objets (l'état est encapsulé).</p> <p>Le graph de communication est dynamique, ainsi, les objets naissent (sont instanciés) et meurent (sont détruits, désalloués).</p> <p>... où mais concrètement?</p> <p>1. Au moins implicitement : ici, "interface" ne désigne pas forcément la construction interface de Java.</p>	

Compléments en P00	Aldric Degorre												
<p>Object = entité...</p> <ul style="list-style-type: none"> caractérisée par un enregistrement contigu de données typées (attributs¹) accessible via une référence² vers cet enregistrement; manipulable/interrogeable via un ensemble de méthodes qui lui est propre. <table border="1"> <tr> <td>classe de l'objet :</td> <td>réf. ↗ Personne.class</td> </tr> <tr> <td> int age</td> <td>42</td> </tr> <tr> <td> String nom</td> <td>réf. ↗ chaîne "Dupont"</td> </tr> <tr> <td> String prenom</td> <td>réf. ↗ chaîne "Toto"</td> </tr> <tr> <td>...</td> <td>...</td> </tr> <tr> <td>boolean marie</td> <td>true</td> </tr> </table> <p>La variable référence l'objet</p> <p>Personne toto</p> <p>référence l'objet</p> <p>...</p> <p>boolean marie true</p>		classe de l'objet :	réf. ↗ Personne.class	int age	42	String nom	réf. ↗ chaîne "Dupont"	String prenom	réf. ↗ chaîne "Toto"	boolean marie	true
classe de l'objet :	réf. ↗ Personne.class												
int age	42												
String nom	réf. ↗ chaîne "Dupont"												
String prenom	réf. ↗ chaîne "Toto"												
...	...												
boolean marie	true												

Autre vision bas niveau, plus en phase avec le haut niveau							
Introduction							
Généralités							
Style							
Objets et classes							
Objets et classes							
Membres et constructeurs							
Objets et classes							
Encapsulation							
Types imbriqués							
Types et polymorphisme							
Héritage							
Généricité							
Concurrence							
Interfaces graphiques							
Gestion des erreurs et exceptions							
Discussion							
<p>Cette dernière vision est en fait réductrice.</p> <p>En POO, on veut s'abstraire de l'implémentation concrète des concepts.</p> <p>À service égal, les objets Personne peuvent aussi être représentés ainsi :</p> <table border="1"> <tr> <td>classe :</td> <td>→ Ident.class</td> </tr> <tr> <td> String ident</td> <td>→ "Dupont"</td> </tr> <tr> <td> String prenom</td> <td>→ "Toto"</td> </tr> </table> <p>Les méthodes seraient écrites différemment mais, à l'usage, cela ne se verrait pas.¹</p> <p>Pourtant cela aurait encore du sens de parler d'objets-Personne contenant des propriétés nom et prenom.</p> <p>1. À condition qu'on n'utilise pas directement les attributs. D'où l'intérêt de les rendre privés !</p>		classe :	→ Ident.class	String ident	→ "Dupont"	String prenom	→ "Toto"
classe :	→ Ident.class						
String ident	→ "Dupont"						
String prenom	→ "Toto"						

<h2>Classes</h2> <p>Langages à prototypes, langages à classes</p>	<p>Compléments en POO Aldric Degorre</p> <ul style="list-style-type: none"> • Besoin : créer de nombreux objets similaires (même interface, même schéma de données). • 2 solutions → 2 familles de LOO : <ul style="list-style-type: none"> • LOO à classes (Java et la plupart des LOO) : les objets sont instanciés à partir de la description donnée par une classe, • LOO à prototypes (toutes les variantes d'ECMAScript dont JavaScript; Self, Lisaac, ...) : les objets sont obtenus par extension d'un objet existant (le prototype). <p>→ l'existence de classes n'est pas une nécessité en POO</p>	<p>Introduction Généralités Style Objets et classes objets et classes Membres et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrency Interfaces graphiques Gestion des erreurs et exceptions Analyse</p>
<h2>Objets</h2> <p>Autre vision bas niveau, plus en phase avec le haut niveau</p>	<p>Question : où arrêter le graphe d'un objet ?</p> <ul style="list-style-type: none"> • Est-ce que les éléments d'une liste font partie de l'objet-liste ? • En exagérant un peu, un programme ne contient en réalité qu'un seul objet ! • Clairement, le graphe d'un objet ne doit pas contenir tous les enregistrements accessibles depuis l'enregistrement principal. Mais où s'arrêter et sur quel critère ? <p>Cela n'est pas anodin :</p> <ul style="list-style-type: none"> • Que veut dire « copier » un objet ? (Quelle « profondeur » pour la copie ?) • Si on parle d'un objet non modifiable, qu'est-ce qui n'est pas modifiable ? • Est-ce qu'une collection non modifiable peut contenir des éléments modifiables ? <p>Cette discussion a trait aux notions d'<u>encapsulation</u> et de <u>composition</u>. À suivre !</p> <p>1. En effet : les enregistrements non référencés par le programme, sont assez vite détruits par le GC.</p>	<p>Compléments en POO Aldric Degorre</p>
<p>Compléments en POO Aldric Degorre</p>	<p>Introduction Généralités Style Objets et classes objets et classes Membres et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrency Interfaces graphiques Gestion des erreurs et exceptions Discussion</p>	<p>Introduction Généralités Style Objets et classes objets et classes Membres et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrency Interfaces graphiques Gestion des erreurs et exceptions Exemple</p>
<p>Compléments en POO Aldric Degorre</p>	<p>Introduction Généralités Style Objets et classes objets et classes Membres et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrency Interfaces graphiques Gestion des erreurs et exceptions Exemple</p>	<p>Introduction Généralités Style Objets et classes objets et classes Membres et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrency Interfaces graphiques Gestion des erreurs et exceptions Exemple</p>

Classes		Classes	
Compléments en POO : Aldric Degorre		Autres points de vue (non-OO) : Aldric Degorre	
Introduction	Points de vue pertinents pour Java	Introduction	Points de vue pertinents pour Java
Généralités		Généralités	
Style		Style	
Objets et classes		Objets et classes	
Objets et classes Méthodes & constructeurs Étapsulation Types immuables		Objets et classes Méthodes & constructeurs Étapsulation Types immuables	
Types et polymorphisme		Types et polymorphisme	
Héritage		Héritage	
Généricité		Généricité	
Concurrence		Concurrence	
Interfaces graphiques		Interfaces graphiques	
Gestion des erreurs et exceptions		Gestion des erreurs et exceptions	
Révision		Révision	
Classe = patron/modèle/moule/... pour définir des objets similaires ¹ .		Classe =	
Autres points de vue :		Classe =	
Classe =		<ul style="list-style-type: none"> • sous-division syntaxique du programme • espace de noms (définitions de nom identique possibles si dans classes différentes) • parfois, juste une bibliothèque de fonctions statiques, non instantiable¹ • exemples de classes non instantiables du JDK : System, Arrays, Math, ... 	
<ul style="list-style-type: none"> • ensemble cohérent de définitions (champs, méthodes, types auxiliaires, ...), en principe relatives à un même type de données • conteneur permettant l'encapsulation (= limite de visibilité des membres privés).² 		<p>Les aspects ci-dessus sont pertinents en Java, mais ne retenir que ceux-ci serait manquer l'essentiel : i.e. : classe = concept de POO.</p>	
<ol style="list-style-type: none"> 1. "similaires" = utilisables de la même façon (même type) et aussi structurés de la même façon. 2. Remarque : en Java, l'encapsulation se fait par rapport à la classe et au paquetage et non par rapport à l'objet. En Scala, p. ex., un attribut peut avoir une visibilité limitée à l'objet qui le contient. 		<ol style="list-style-type: none"> 1. Java force à tout définir dans des classes → encourage cet usage détourné de la construction class. 	
Instanciation		Instanciation	
De la classe à l'objet		Constructeurs (1)	
Compléments en POO : Aldric Degorre		Compléments en POO : Aldric Degorre	
Introduction		Introduction	
Généralités		Généralités	
Style		Style	
Objets et classes		Objets et classes	
Objets et classes Méthodes & constructeurs Étapsulation Types immuables		Objets et classes Méthodes & constructeurs Étapsulation Types immuables	
Types et polymorphisme		Types et polymorphisme	
Héritage		Héritage	
Généricité		Généricité	
Concurrence		Concurrence	
Interfaces graphiques		Interfaces graphiques	
Gestion des erreurs et exceptions		Gestion des erreurs et exceptions	
Révision		Détails	
Constructeur : fonction ¹ servant à construire une instance d'une classe.		Constructeur : fonction ¹ servant à construire une instance d'une classe.	
Déclaration :		Déclaration :	
<pre>MacClasse(* paramètres */) { // instructions ; ici "this" désigne l'objet en construction }</pre>		<pre>MacClasse(* paramètres */) { // instructions ; ici "this" désigne l'objet en construction }</pre>	
NB : même nom que la classe, pas de type de retour, ni de return dans son corps.		NB : même nom que la classe, pas de type de retour, ni de return dans son corps.	
Typiquement , // instructions = initialisation des attributs de l'instance.		Typiquement , // instructions = initialisation des attributs de l'instance.	
Appel toujours précédé du mot-clé new :		Appel toujours précédé du mot-clé new :	
<pre>MacClasse monObjet = new MacClasse(... paramètres...);</pre>		<pre>MacClasse monObjet = new MacClasse(... paramètres...);</pre>	
Cette instruction déclare un objet monObjet , crée une instance de MacClasse et l'affecte à monObjet .		Cette instruction déclare un objet monObjet , crée une instance de MacClasse et l'affecte à monObjet .	
1. En POO, "instance" et "objet" sont synonymes. Le mot "instance" souligne l'appartenance à un type.		1. En toute rigueur, un constructeur n'est pas une méthode. Notons tout de même les similarités dans les syntaxes de déclaration et d'appel et dans la sémantique (exécution d'un bloc de code).	
2. En l'occurrence : les attributs d'instance déclarés dans cette classe.		2. En l'occurrence : les attributs d'instance déclarés dans cette classe.	
3. Ainsi, on note que le type défini par une classe est un type référence.		3. Ainsi, on note que le type défini par une classe est un type référence.	

Le corps d'une classe **C** consiste en une séquence de définitions : constructeurs¹ et **membres** de la classe.

- définir plusieurs constructeurs (tous le même nom → cf. surchage);
- définir un constructeur secondaire à l'aide d'un autre constructeur déjà défini : sa première instruction doit alors être **this**(**paramsAutreConstructeur**) ;¹
- ne pas écrire de constructeur :

- Si on ne le fait pas, le compilateur ajoute un **constructeur par défaut** sans paramètre.²
 - Si on a écrit un constructeur, alors il n'y a pas de constructeur par défaut³.
1. Ou bien **super**(**params**) ; si utilisation d'un constructeur de la superclasse.
 2. Les attributs restent à leur valeur par défaut (0, **false** ou **null**), ou bien à celle donnée par leur initialiseur, s'il y en a un.
 3. Mais rien n'empêche d'écrire, en plus, à la main, un constructeur sans paramètre.

Il est possible de :

- définir plusieurs constructeurs (tous le même nom → cf. surchage);
 - définir un constructeur secondaire à l'aide d'un autre constructeur déjà défini : sa première instruction doit alors être **this**(**paramsAutreConstructeur**) ;¹
 - ne pas écrire de constructeur :
- Si on ne le fait pas, le compilateur ajoute un **constructeur par défaut** sans paramètre.²
 - Si on a écrit un constructeur, alors il n'y a pas de constructeur par défaut³.

- Plusieurs catégories de membres : attributs, méthodes et types membres².
- Un membre **m** peut être
- soit non statique ou **d'instance** (relatif à une instance de **C**)
 - Utilisable en écrivant « **m** » n'importe où où un **this** (**récepteur** implicite) de type **C**.
Type et polymorphisme
 - soit **statique** (relatif à la classe **C**) → mot-clé **static** dans déclaration.
 - Utilisable sans préfixe dans le corps de **C** et ailleurs en écrivant « **C.m** ».
- Les **membres d'un objet** donné sont les membres non statiques de la classe de l'objet.
1. D'après la JLS 8.2, les constructeurs ne sont pas des membres. Néanmoins, sont déclarés à l'intérieur d'une classe et acceptent, comme les membres, les modificateurs de visibilité (**private**, **public**, ...).
 2. Souvent abusivement appelés "classes internes".

- Plusieurs catégories de membres : attributs, méthodes et types membres².
- Un membre **m** peut être
- soit non statique ou **d'instance** (relatif à une instance de **C**)
 - Utilisable en écrivant « **m** » n'importe où où un **this** (**récepteur** implicite) de type **C**.
Type et polymorphisme
 - soit **statique** (relatif à la classe **C**) → mot-clé **static** dans déclaration.
 - Utilisable sans préfixe dans le corps de **C** et ailleurs en écrivant « **C.m** ».
- Les **membres d'un objet** donné sont les membres non statiques de la classe de l'objet.
1. D'après la JLS 8.2, les constructeurs ne sont pas des membres. Néanmoins, sont déclarés à l'intérieur d'une classe et acceptent, comme les membres, les modificateurs de visibilité (**private**, **public**, ...).
 2. Souvent abusivement appelés "classes internes".

- Accéder à un membre : où et comment ? → notion de contexte

Contexte (associé à tout point du code source) :

- dans une définition¹ statique : contexte = la classe contenant la définition ;
- dans une définition non-statique : contexte = l'objet "courant", le **récepteur**².

- Désigner un membre **m** déjà défini quelque part :

- écrire soit juste **m** (**nom simple**), soit **chemin.m** (**nom qualifié**)
 - "**chemin**" donne le contexte auquel appartient le membre **m** :
 - pour un membre statique : la classe³ (ou interface ou autre...) où il est défini
 - pour un membre d'instance : une instance de la classe où il est défini
 - " **chemin** ." est facultatif si **chemin** == contexte local.
-
1. typiquement, remplacer "définition" par "corps de méthode"
 2. L'objet qui, à cet endroit, serait référencé par **this**.
 3. Et pour désigner une classe d'un autre paquetage : **chemin = paquetage.NomDeClasse**.

Membres statiques et membres non statiques

Membres statiques et membres non-statiques

Compléments en 200

Compléments en 200

En bref : Membre non statique = lié à (la durée de vie et au contexte d') une instance.

Membre statique = lié à (la durée de vie et au contexte d') une classe¹.

statique (ou "de classe")	non statique (ou "d'instance")
attribut donnée globale ² , commune à toutes les instances de la classe.	donnée propre ³ à chaque instance (nouvel exemplaire de cette variable alloué et initialisé à chaque instanciation).
méthode "fonction", comme celles des langages impératifs.	message à instance concernée : le récepteur de la méthode (this).

1. +permanent et « global ». NB : ça ne veut pas dire visible de partout : **static private** est possible!
2. Correspond à variable globale dans d'autres langages.
3. Correspond à champ de **struct** en C.

Résumé

Membres statiques et membres non-statiques

Zoom sur le cas des attributs

Introduction

Introduction

Généralités

Généralités

Style

Style

Objets et classes

Objets et classes

objets et classes

objets et classes

Méthodes et constructeurs

Méthodes et constructeurs

Encapsulation

Encapsulation

Type et polymorphisme

Type et polymorphisme

Héritage

Héritage

Généricité

Généricité

Concurrence

Concurrence

Interfaces graphiques

Interfaces graphiques

Gestion des erreurs et exceptions

Gestion des erreurs et exceptions

Exemple

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Introduction

Introduction

Généralités

Généralités

Style

Style

Objets et classes

Objets et classes

objets et classes

objets et classes

Méthodes et constructeurs

Méthodes et constructeurs

Encapsulation

Encapsulation

Type et polymorphisme

Type et polymorphisme

Héritage

Héritage

Généricité

Généricité

Concurrence

Concurrence

Interfaces graphiques

Interfaces graphiques

Gestion des erreurs et exceptions

Gestion des erreurs et exceptions

Exemple

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Introduction

Introduction

Généralités

Généralités

Style

Style

Objets et classes

Objets et classes

objets et classes

objets et classes

Méthodes et constructeurs

Méthodes et constructeurs

Encapsulation

Encapsulation

Type et polymorphisme

Type et polymorphisme

Héritage

Héritage

Généricité

Généricité

Concurrence

Concurrence

Interfaces graphiques

Interfaces graphiques

Gestion des erreurs et exceptions

Gestion des erreurs et exceptions

Exemple

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Introduction

Introduction

Généralités

Généralités

Style

Style

Objets et classes

Objets et classes

objets et classes

objets et classes

Méthodes et constructeurs

Méthodes et constructeurs

Encapsulation

Encapsulation

Type et polymorphisme

Type et polymorphisme

Héritage

Héritage

Généricité

Généricité

Concurrence

Concurrence

Interfaces graphiques

Interfaces graphiques

Gestion des erreurs et exceptions

Gestion des erreurs et exceptions

Exemple

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Introduction

Introduction

Généralités

Généralités

Style

Style

Objets et classes

Objets et classes

objets et classes

objets et classes

Méthodes et constructeurs

Méthodes et constructeurs

Encapsulation

Encapsulation

Type et polymorphisme

Type et polymorphisme

Héritage

Héritage

Généricité

Généricité

Concurrence

Concurrence

Interfaces graphiques

Interfaces graphiques

Gestion des erreurs et exceptions

Gestion des erreurs et exceptions

Exemple

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Qu'affiche le programme suivant?

Compléments en 200

Alain Degrave

Zoom sur le cas des attributs

Résumé

Fabriques statiques

Un cas d'utilisation intéressant pour les méthodes statiques (EJ3 Item 1)

Problème, les limitations des constructeurs :

- même nom pour tous, qui ne renseigne pas sur l'usage fait des paramètres;
- impossibilité d'avoir 2 constructeurs avec la même signature;
- appel à constructeur auxiliaire, nécessairement en première instruction;
- obligation de retourner une nouvelle instance → pas de contrôle d'instances¹;
- obligation de retourner une instance directe de la classe.

En écrivant une **fabrique statique** on contourne toutes ces limitations :

```
public abstract class C { // ou bien interface
    ...
    public static C of(D arg) {
        if (arg ...) return new CImpl1(arg);
        else if (arg ...) return ...
        else return ...
    }
}
```

1. I.e. : possibilité de choisir de réutiliser une instance existante au lieu d'en créer une nouvelle.

L'encapsulation

- = restriction de l'accès depuis l'extérieur aux choix d'implémentation internes.
 - **bonne pratique** favorisant la pérennité d'une classe.
 - Minimiser la « surface » qu'une classe expose à ses clients¹ (= en réduisant leur **couplage**) facilite son déboguage et son évolution future.²
 - empêche les clients d'accéder à un objet de façon incorrecte ou non prévue. Ainsi, la correction d'un programme est plus facile à vérifier (moins d'interactions à vérifier); plus généralement, seuls les **invariants de classe**³ ne faisant pas intervenir d'attributs non privés peuvent être prouvés.
- L'encapsulation rend donc aussi la classe plus fiable.

1. **Clients** d'une classe : les classes qui utilisent cette classe.

2. En effet : on peut modifier la classe sans modifier ses clients.

3. Différence avec l'item du dessus : les invariants de classe doivent rester vrais dans tout contexte d'utilisation de la classe, pas seulement dans le programme courant.

Est-il vrai que « le n^{ième} appel à next retourne le n^{ième} terme de la suite de Fibonacci ?

Pas bien :

```
public class FiboGen {
    public int a = 1, b = 1;
    public int next() {
        int ret = a; a = b; b += ret;
        return ret;
    }
}
```

Toute autre classe peut interférer en modifiant directement les valeurs de a ou b → on ne peut rien prouver!

Encapsulation

- Introduction
- Généralités
- Style
- Objets et classes
- Membres et constantes
- Encapsulation
- Types imbriqués
- Types et polymorphisme
- Héritage
- Généricité
- Concurrence
- Interfaces graphiques
- Gestion des erreurs et exceptions

Est-il vrai que « le n^{ième} terme de la suite de Fibonacci ?

Bien : (ou presque)

```
public class FiboGen {
    private int a = 1, b = 1;
    public int next() {
        int ret = a; a = b; b += ret;
        return ret;
    }
}
```

Seule la méthode next peut modifier directement les valeurs de a ou b → S'il y a un bug, c'est dans la méthode next et pas ailleurs!¹

Encapsulation

- Introduction
- Généralités
- Style
- Objets et classes
- Membres et constantes
- Encapsulation
- Types imbriqués
- Types et polymorphisme
- Héritage
- Généricité
- Concurrence
- Interfaces graphiques
- Gestion des erreurs et exceptions

Est-il vrai que « le n^{ième} terme de la suite de Fibonacci ?

Pas bien :

```
public class FiboGen {
    public int a = 1, b = 1;
    public int next() {
        int ret = a; a = b; b += ret;
        return ret;
    }
}
```

Toute autre classe peut interférer en modifiant directement les valeurs de a ou b → on ne peut rien prouver!

1. Or il y a un bug, en théorie, si on exécute next plusieurs fois simultanément (sur plusieurs threads).

Encapsulation

Remarque

Niveaux de visibilité : privé, public, etc.

- L'encapsulation est mise en œuvre via les **modificateurs de visibilité** des membres.
- 4 niveaux de visibilité en faisant précéder leurs déclarations de **private**, **protected** ou **public** ou d'aucun de ces mots (→ visibilité *package-private*).

Visibilité	classe	paquetage	sous-classes	partout
private	X			
package-private	X	X		
protected	X	X	X	
public	X	X	X	X

Exemple :

```
class A {
    int x; // visible dans le package
    private double y; // visible seulement dans A
    public final String nom = "Toto"; // visible partout
}
```

Révision

1. voir héritage

Encapsulation

En passant : niveaux de visibilité pour les déclarations de premier niveau

Niveaux de visibilité et documentation

- Au contraire de nombreux autres principes exposés dans ce cours, l'encapsulation ne favorise pas directement la réutilisation de code.
- À première vue, c'est le contraire : on interdit l'utilisation directe de certaines parties de la classe.
- En réalité, l'encapsulation augmente la confiance dans le code réutilisé (ce qui, indirectement, peut inciter à le réutiliser davantage).

Encapsulation

Notion de visibilité : s'applique aussi aux déclarations de premier niveau¹.Ici, 2 niveaux seulement : **public** ou **package-private**.

Visibilité	paquetage	partout
package-private	X	
public	X	X

Rappel : une seule déclaration publique de premier niveau autorisée par fichier. La classe/interface/... définie porte alors le même nom que le fichier.

1. Précisions/rappels :

- "premier niveau" = hors des classes, directement dans le fichier;
- seules les déclarations de type (classes, interfaces, énumérations, annotations) sont concernées.

Niveaux de visibilité : privé, public, etc.

- L'encapsulation est mise en œuvre via les **modificateurs de visibilité** des membres.
- 4 niveaux de visibilité en faisant précéder leurs déclarations de **private**, **protected** ou **public** ou d'aucun de ces mots (→ visibilité *package-private*).

Visibilité	classe	paquetage	sous-classes	partout
private	X			
package-private	X	X		
protected	X	X	X	
public	X	X	X	X

Exemple :

```
class A {
    int x; // visible dans le package
    private double y; // visible seulement dans A
    public final String nom = "Toto"; // visible partout
}
```

Révision

1. voir héritage

Niveaux de visibilité et documentation

- Toute déclaration de membre non **private** est susceptible d'être utilisée par un autre programmeur dès lors que vous publiez votre classe.
- Elle fait partie de l'API¹ de la classe.
- vous devez donc **la documenter**² (EJ3 Item 56)
- et vous vous engagez à ne pas **modifier**³ sa spécification⁴ dans le futur, sous peine de "casser" tous les clients de votre classe.

Ainsi il faut bien réfléchir à ce que l'on souhaite exposer.⁵**1. Application Programming Interface**

2. cf. JavaDoc

- On peut modifier si ça va dans le sens d'un renforcement compatible.
- Et, évidemment, à faire en sorte que le comportement réel respecte la spécification!
- Il faut aussi réfléchir à une stratégie : tout mettre en **private** d'abord, puis relâcher en fonction des besoins ? Ou bien le contraire ? Les opinions divergent !

Attention, les niveaux de visibilité ne font pas forcément ce à quoi on s'attend.

- **package-private** → on peut, par inadvertance, créer une classe dans un paquetage déjà existant¹ → garantie faible.

- **protected** → de même et, en +, toute sous-classe, n'importe où, voit la définition.
- **Aucun niveau ne garantit la confidentialité des données.**
- Constantes : lisibles directement dans le fichier **Class**.

Variables : lisibles, via réflexion, par tout programme s'exécutant sur la même JVM.
Si la sécurité importe : bloquer la réflexion².

L'encapsulation évite les erreurs de programmation mais **n'est pas un outil de sécurité**³.

1. Même à une dépendance tierce, même sans recopilation. En tout cas, si on n'utilise pas JPMS.
2. En utilisant un **SecurityManager** ou en configurant **module-info.java** avec les bonnes options.
3. Méditer la différence entre sûreté (safety) et sécurité (security) en informatique. Attention, cette distinction est souvent faite, mais selon le domaine de métier, la distinction est différente, voire inversée !

- Java permet désormais de regrouper les packages en **modules**.

- Chaque module contient un fichier **module-info.java** déclarant quels packages du module sont **exportés** et de quels autres modules il **dépend**.
- Le module dépendant a alors accès aux packages exportés par ses dépendances.
- **Les autres packages de ses dépendances lui sont invisibles**!¹

Syntaxe du fichier **module-info.java** :

```
module nom_du_module {
    requires nom_d'un_module_dont_on_depends;
    exports nom_d'un_package_defini_icis;
```

Ce sujet sera développé en TP.

1. Et les dépendances sont fermées à la réflexion, mais on peut permettre la réflexion sur un package en le déclarant avec **opens** dans **module-info.java**.

- Une propriété se base souvent sur un attribut (**privé**), mais d'autres implémentations sont possibles. P. ex. :

```
// propriété "numberOfFingers" :
public int getNumberOfFingers() { return 10; }
```

(accès en lecture seule à une valeur constante → on retourne une expression constante)

- L'utilisation d'accesseurs laisse la **possibilité de changer ultérieurement l'implémentation** de la propriété, sans changer son mode d'accès public¹.
Ainsi, quand cela sera fait, il ne sera **pas nécessaire de modifier les autres classes** qui accèdent à la propriété.

1. EJ3 Item 16 : "In public classes, use accessor methods, not public fields"
2. Variante : **public boolean isX()**, seulement si **T** est **boolean**.
3. Terminologie utilisée dans la spécification JavaBeans pour le couple getteur+setteur. Dans nombre de LOO (C#, Kotlin, JavaScript, Python, Scala, Swift, ...), les propriétés sont cependant une sorte de membre à part entière supportée par le langage.

1. ici, le couple de méthodes **getX()**/**setX()**

Exemple : propriété en lecture/écriture avec contrôle validité des données.

```
public final class Person {
    // propriété "age"
    // attribut de base (qui doit rester positif)
    private int age;

    // getteur, accesseur en lecture
    public int getAge() {
        return age;
    }

    // setteur, écriture contrôlée
    public void setAge(int a) {
        if (a >= 0) age = a;
    }
}
```

Exemple

Exemple : propriété en lecture seule avec évaluation paresseuse.

```
public final class Entier {
    public final int valeur; // this.valeur = valeur;

    private final int valeur;
    // propriété ``diviseurs`` :
    private List<Integer> diviseurs;

    public List<Integer> getDiviseurs() {
        if (diviseurs == null) diviseurs =
            Collections.unmodifiableList(ouutils.factorise(valeur)); // <- calcul
        return diviseurs;
    }
}
```

Exemple

Exemple : propriété en lecture seule avec évaluation paresseuse.

Exemple

Exemple : propriété en lecture seule avec évaluation paresseuse.

Exemple

Exemple : propriété en lecture seule avec évaluation paresseuse.

Exemple

Exemple : propriété en lecture seule avec évaluation paresseuse.

Exemple

Encapsulation

Accessseurs (get, set, ...) et propriétés (5)

Encapsulation

Accessseurs (get, set, ...) et propriétés (3)

Aliasing = existence de références multiples vers un même objet.

Aliasing : pourquoi les restrictions de visibilité ne suffisent pas pour garantir l'encapsulation

ref1 → objet ← ref2

Quand un attribut référence un objet qui est aussi référencé à l'extérieur de cette classe, le bénéfice de l'encapsulation est alors annulé.

À éviter :

- attribut privé de C → objet ← référence externe à C
- comportements envisageables pour **get** et **set** :
- contrôle de validité avant modification;
- initialisation paresseuse : la valeur de la propriété n'est calculée que lors du premier accès (et non dès la construction de l'objet);
- consignation dans un journal pour déboguage ou surveillance;
- observabilité : le setteur informe les objets observateurs lors des modifications;
- vétoabilité : le setteur n'a d'effet que si aucun objet (dans une liste connue de "veto-eurs") ne s'oppose au changement;
- ...

1. Quasiment : en effet, si l'attribut est privé, il reste impossible de modifier la valeur de l'attribut, i.e. l'adresse qu'il stocke, depuis l'extérieur.

Encapsulation

Aliasing : comment l'empêcher.

Encapsulation

Aliasing : exercice

Lesquelles des classes A, B, C et D garantissent que l'entier contenu dans l'attribut d garde la valeur qu'on y a mise à la construction ou lors du dernier appel à setData ?

```

class Data {
    public int x;
    public Data(int x) { this.x = x; }
    public Data copy() { return new Data(x); }
}

class A {
    private final Data d;
    public A(Data d) { this.d = d; }

    class B {
        private final Data d;
        // copie défensive (EJ3 Item 50)
        public B(Data d) { this.d = d.copy(); }
        public Data getData() { return d; }
    }
}

```

Revenir à répondre à : les attributs de ces classes peuvent-ils avoir des alias extérieurs ?

Aliasing souvent indésirable (pas toujours !) → il faut savoir l'empêcher. Pour cela :

```

class A {
    // Mettre les attributs sensibles en private :
    private Data data;
    // Effectuer des copies défensives (EJ3 Item 50)...
    // - de tout objet qu'on souhaite partager,
    // - qu'il soit retourné par un getteur :
    public Data getData() { return data.copy(); }
    // - ou passé en paramètre d'une méthode extérieure :
    public void foo() { X.bar(data.copy()); }
    // - de tout objet passé en argument pour être stocké dans un attribut
    // - que ce soit dans les méthodes
    public void setData(Data data) { this.data = data.copy(); }
    // - ou dans les constructeurs
    public A(Data data) { this.data = data.copy(); } //
}

```

Résumé : ni divulguer ses références, ni conserver une référence qui nous a été donnée.

Compléments en 200

Aldric Degorre

Introduction Généralités Style Objets et classes Objets et classes Méthodes et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions

Compléments en 200

Aldric Degorre

Introduction Généralités Style Objets et classes Objets et classes Méthodes et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions

Copie défensive

Qu'est-ce que c'est et comment la réalise-t-on ? (exemple)

```

public class Item {
    int productNumber; Point location; String name;
    public Item copy() { // Item est mutable, donc cette méthode est utile
        Item ret = new Item();
        ret.productNumber = productNumber; // int est primitive, une copie simple suffit
        ret.location = new Point(location.x, location.y); // Point est mutable, il faut
        // une copie profonde
        ret.name = name; // String est immuable, une copie simple suffit
        return ret;
    }
}

```

Remarque : il est impossible¹ de faire une copie profonde d'une classe mutable dont on n'est pas l'auteur si ses attributs sont privés et l'auteur n'a pas prévu lui-même la copie.

1. La relation d'égalité est celle donnée par la méthode `equals`.
 2. Type **immutable** (*immutable*) : type (en fait toujours une classe) dont toutes les instances sont des objets non modifiables.
- C'est une propriété souvent recherchée, notamment en programmation concurrente.

Exemple

Compléments en 200

Aldric Degorre

Introduction Généralités Style Objets et classes Objets et classes Méthodes et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions

Compléments en 200

Aldric Degorre

Introduction Généralités Style Objets et classes Objets et classes Méthodes et constructeurs Encapsulation Types imbriqués Types et polymorphisme Héritage Généricité Concurrence Interfaces graphiques Gestion des erreurs et exceptions

Copie défensive

Qu'est-ce que c'est et comment la réalise-t-on ?

- **Copie défensive** = copie profonde réalisée pour éviter des alias indésirables.
- **Copie profonde** : technique consistant à obtenir une copie d'un objet « égale »¹ à son original au moment de la copie, mais dont les évolutions futures seront indépendantes.
- **2 cas, en fonction du genre de valeur à copier :**
 - Si type **primitif** ou **immuable**², pas d'évolutions futures → une copie directe suffit.
 - Si type **mutable** → on crée un nouvel objet dont les attributs contiennent des copies profondes des attributs de l'original (et ainsi de suite, récursivement : on copie le graphe de l'objet³).

1. La relation d'égalité est celle donnée par la méthode `equals`.
 2. Type **immutable** (*immutable*) : type (en fait toujours une classe) dont toutes les instances sont des objets non modifiables.
- C'est une propriété souvent recherchée, notamment en programmation concurrente.
3. Il sauf à utiliser la réflexion... mais dans le cadre du JMS, il ne faut pas trop compter sur celle-ci.

Immuabilité...

Compléments en 200
Aldric Degorre
Encapsulation
Aliasing : pourquoi ce phénomène peut nous gêner plus tard...

... ah et comment savoir si un type est immuable? Nous y reviendrons.

Sont notamment immuables :

- la classe **String**;
 - toutes les *primitive wrapper classes* : Boolean, Char, Byte, Short, Integer, Long, Float et Double;
 - d'autres sous-classes de Number : BigInteger et BigDecimal ;
 - plus généralement, toute classe¹ dont la documentation dit qu'elle l'est.
- En pratique, les 8 types primifs (**boolean, char, byte, short, int, long, float, double**) se comportent aussi² comme des types immuables³.
-
1. Les types définis par les interfaces ne peuvent pas être garantis immuables.
 2. Fonctionnellement. Pour d'autres aspects, comme la performance, le comportement est différent.
 3. Mais cette distinction n'a pas de sens pour des valeurs directes.

Compléments en 200
Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Objets et classes

Méthodes et constructeurs

Encapsulation

Type imbriqués

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Détails

En cas d'*alias* extérieur d'un attribut a de type mutable dans une classe C :

- on ne peut pas prouver d'invariant de C faisant intervenir a, notamment, la classe C n'est pas immuable (certaines instances pourraient être modifiées par un tiers);
- on ne peut empêcher les modifications concurrentes¹ de l'objet aliasé, dont le résultat est notoirement imprévisible.²

Il reste possible néanmoins de prouver des invariants de C ne faisant pas intervenir a ; cela peut être suffisant dans bien des cas (y compris dans un contexte concurrent).

1. Faisant intervenir un autre *thread*, cf. chapitre sur la programmation concurrenante.
2. Plus généralement, ce problème se pose dès qu'un objet peut être partagé par des méthodes de classes différentes.
Si la référence vers cet objet ne sort pas de la classe, il est possible de synchroniser les accès à cet objet.

Compléments en 200
Aldric Degorre
Encapsulation
Aliasing, est-ce toujours « mal » ?

Pour conclure sur l'*aliasing*.

Il n'y a pas que des inconvenients à partager des références :

- 1 **Aliaser** permet d'éviter le surcoût (en mémoire, en temps) d'une copie défensive.
- Optimisation à considérer si les performances sont critiques.
- 2 **Aliaser** permet de simplifier la maintenance d'un état cohérent dans le programme (vu qu'il n'y a plus de copies à synchroniser).

Mais dans tous les cas il faut être conscient des risques :

- dans 1., mauvaise idée si plusieurs des contextes partageant la référence pensent être les seuls à pouvoir modifier l'objet référencé ;
- dans 2., risque de modifications concurrentes dans un programme *multi-thread* → précautions à prendre.

Compléments en 200
Aldric Degorre

Introduction

Généralités

Style

Objets et classes

Objets et classes

Méthodes et constructeurs

Encapsulation

Type imbriqués

Types et polymorphisme

Héritage

Généricité

Concurrence

Interfaces graphiques

Gestion des erreurs et exceptions

Supplément

Compléments en 200
Aldric Degorre
Encapsulation
Aliasing : remarque sans rapport avec l'encapsulation

Compléments en 200
Aldric Degorre
Encapsulation
Aliasing : zone de mémoire dans la pile, dédiée au stockage des informations locales pour un appel de méthode donné.

L'impossibilité d'*alias* extérieur au *frame*¹ d'une méthode est aussi intéressante, car elle autorise la JVM à optimiser en allouant l'objet directement en pile plutôt que dans le tas. En effet : comme l'objet n'est pas référencé en dehors de l'appel courant, il peut être détruit sans risque au retour de la méthode.
La recherche de la possibilité qu'une exécution crée des *alias* externes (à une classe ou une méthode) s'appelle l'**escape analysis**².

1. *frame* = zone de mémoire dans la pile, dédiée au stockage des informations locales pour un appel de méthode donné.
2. Traduction proposée : **analyse d'échappement** ?

Types imbriqués

Qu'est-ce que c'est ?

Java permet de définir un **type (classe ou interface) imbriqué¹ à l'intérieur d'une autre définition de type (dit **englobant**²) :**

```
public class ClasseStupide {
    public static interface JToto {
        public static class CTata {
            public enum EPlop {
                VTOTO;
                public interface JToto { }
            }
        }
    }
}
```

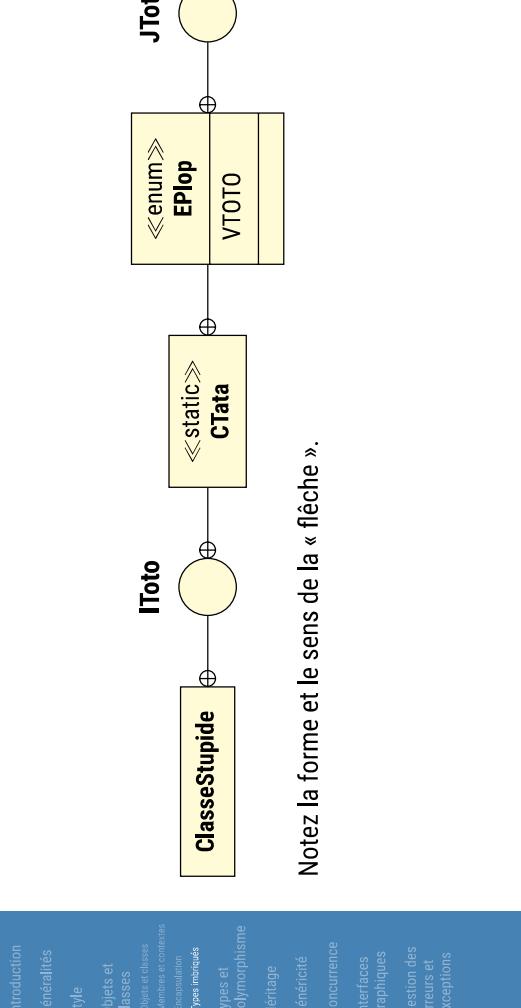
1. La plupart des documentations ne parlent en réalité que de "classes imbriquées" (*nested classes*), mais c'est trop réducteur. D'autres disent "classes internes"/*inner classes*, mais ce nom est réservé à un cas particulier. Voir la suite.
 2. *enclosing...* mais on voit aussi *outer/externe*

Compléments
en P00

Types imbriqués
En UML

Introduction
Généralités
Style
Objets et classes
objets et classes
Membres et constructeurs
Encapsulation
Types imbriqués
Types et polymorphisme
Héritage
Généricité
Concurrency
Interfaces graphiques
Gestion des erreurs et exceptions

Introduction
Généralités
Style
Objets et classes
objets et classes
Membres et constructeurs
Encapsulation
Types imbriqués
Types et polymorphisme
Héritage
Généricité
Concurrency
Interfaces graphiques
Gestion des erreurs et exceptions
Analyse



Types imbriqués

Pour quoi faire ?

- définitions du contexte englobant incluses dans contexte imbriqué (sans chemin).
- Type englobant et types membres peuvent accéder aux membres **privé** des uns des autres → utile pour partage de définitions privées entre classes "amies"¹.

L'exemple ci-dessous compile :

```
class TE {
    static class TIA {
        private static void fIA() { fE(); } // pas besoin de donner le chemin de fE
    }
}

static class TIB {
    static class TIA {
        private static void fIB() { fE(); } // pas besoin de donner le chemin de fE
    }
}

private static void fE() { TIB.fIB(); } // TIB.fIB() ; } // TIB visible malgré private
```

1. La notion de classe imbriquée peut effectivement, en outre, satisfaire le même besoin que la notion de friend class en C++ (quoique de façon plus grossière...).

Types imbriqués

Pour quoi faire ?

- L'imbrication permet l'**encapsulation des définitions de type et de leur contenu** :
- ```
class A {
 static class AA { static int x; } // définition de x à l'intérieur de AA
 private static class AB { } // comme pour tout membre, la visibilité peut être modifiée (ici private, mais public et protected sont aussi possibles)
}

void fa() {
 // System.out.println(x); // non, x n'est pas défini ici ! <- pas de pollution de l'espace de nom du type englobant par les membres du type imbriqué
 System.out.println(AB.x); // oui !
}

class B {
 void fb() {
 // new AA(); // non ! -> classe imbriquée pas dans l'espace de noms du package
 new A.AA(); // <- oui !
 // new A.AB(); <- non ! (AB est private dans A)
 }
}
```

Compléments  
en P00

Types imbriqués  
En UML

Introduction  
Généralités  
Style  
Objets et classes  
objets et classes  
Membres et constructeurs  
Encapsulation  
Types imbriqués  
Types et polymorphisme  
Héritage  
Généricité  
Concurrency  
Interfaces graphiques  
Gestion des erreurs et exceptions

Introduction  
Généralités  
Style  
Objets et classes  
objets et classes  
Membres et constructeurs  
Encapsulation  
Types imbriqués  
Types et polymorphisme  
Héritage  
Généricité  
Concurrency  
Interfaces graphiques  
Gestion des erreurs et exceptions  
Analyse

|                                                                                                              |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|--------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <h2>Plusieurs sortes de types imbriqués</h2> <p>Définitions et classification</p>                            | <p>Compléments en 200<br/>Aldric Degorre</p> <p><b>Classification des types imbriqués/nested types<sup>1</sup></b></p> <ul style="list-style-type: none"> <li>• <b>types membres statiques/static member classes<sup>2</sup></b> : types définis directement dans la classe englobante, définition précédée de <b>static</b>.</li> <li>• <b>classes internes/inner classes</b> : les autres types imbriqués (toujours des classes)       <ul style="list-style-type: none"> <li>• <b>classes membres non statiques/non-static member classes<sup>3</sup></b> : définies directement dans le type englobant</li> <li>• <b>classes locales/local classes</b> : définies dans une méthode avec la syntaxe habituelle (<b>class NomClasseLocale { /*contenu */ }</b>)</li> <li>• <b>classes anonymes/anonymous classes</b> : définies "à la volée" à l'intérieur d'une expression, afin d'instancier un objet unique de cette classe :<br/><b>new NonSuperTypeDirect() { /*contenu */ }.</b></li> </ul> </li> </ul> <ol style="list-style-type: none"> <li>1. J'essaye de suivre la terminologie de la JLS... traduite, puis complétée par la logique et le bon sens.</li> <li>2. La JLS les appelle <i>static nested classes</i>... oubliant que les interfaces membres existent aussi!</li> <li>3. parfois appelées juste <i>inner classes</i>; pas de nom particulier donné dans la JLS.</li> </ol> |
| <h2>Types imbriqués</h2> <p>Exemple de classe membre non statique</p>                                        | <p>Compléments en 200<br/>Aldric Degorre</p> <p><b>Définition similaire au cas précédent, mais sans le mot-clé static.</b></p> <pre>class Maliste&lt;T&gt; implements List&lt;T&gt; {     private class MonIterateur implements Iterator&lt;T&gt; {         // ces méthodes utilisent les attributs non statiques de Maliste directement         public boolean hasNext() {...}         public T next() {...}         public void remove() {...}     }     ...     public Iterateur&lt;T&gt; iterator() {         return new MonIterateur(); // possible parce que iterator() n'est pas statique     } }</pre> <p>• Pour créer une instance de <b>Maliste&lt;String&gt;.MonIterateur</b>, il faut évaluer "<b>new MonIterateur()</b>" dans le contexte d'une instance de <b>Maliste&lt;String&gt;</b>.</p> <p>• Si on n'est pas dans le contexte d'une telle instance, on peut écrire "<b>x.new MonIterateur()</b>" (où <b>x</b> instance de <b>Maliste&lt;String&gt;</b>).</p>                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <h2>Types imbriqués</h2> <p>Exemple de type membre statique</p>                                              | <p>Compléments en 200<br/>Aldric Degorre</p> <p><b>La définition de classe prend la place d'une déclaration de membre du type englobant et est précédée de static.</b></p> <pre>class Maliste&lt;T&gt; implements List&lt;T&gt; {     private static class MonIterateur&lt;U&gt; implements Iterator&lt;U&gt; {         // ces méthodes travaillent sur les attributs de listeBase         private final Maliste&lt;ListeBase&gt; listeBase;         public MonIterateur(Maliste&lt;ListeBase&gt; listeBase) { listeBase = l; }         public boolean hasNext() { ... }         public U next() { ... }         public void remove() { ... }     }     ...     public Iterateur&lt;T&gt; iterator() { return new MonIterateur&lt;T&gt;(this); } }</pre> <p>On peut créer une instance de <b>MonIterateur</b> depuis n'importe quel contexte (même statique) dans <b>Maliste</b> avec juste "<b>new MonIterateur&lt;T&gt;(this)</b>".</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <h2>Types imbriqués</h2> <p>Accès à l'instance imbriquée et à l'instance englobante : this et outer.this</p> | <p>Compléments en 200<br/>Aldric Degorre</p> <p><b>Soit TI un type imbriqué dans TE, type englobant. Alors, dans TI :</b></p> <ul style="list-style-type: none"> <li>• <b>this</b> désigne toujours (quand elle existe) l'instance courante de TI ;</li> <li>• <b>TE.this</b> désigne toujours (quand elle existe ) l'<b>instance englobante</b>, c.-à-d. l'instance courante de TE, c.-à-d. :       <ul style="list-style-type: none"> <li>• si TI classe membre non statique, la valeur de <b>this</b> dans le contexte où l'instance courante de TI a été créée. <b>Exemple :</b></li> </ul> </li> </ul> <pre>class CE {     int x = 1;     class CI {         int y = 2;         void f() { System.out.println(CE.this.x + " " + this.y); }     } } // alors new CE().new CI().f(); affichera "1 2"</pre> <p>• si TI classe locale, la valeur de <b>this</b> dans le bloc dans lequel TI a été déclarée.</p> <p>• si TI classe locale, la valeur de <b>this</b> dans toute instance de TI (attribut caché).</p> <p>La référence TE.<b>this</b> est en fait stockée dans toute instance de TI (attribut caché).</p>                                                                                                                                                                                                                                                                             |

**La définition de classe se place comme une instruction dans un bloc (gén. une méthode) :**

```
class MaListe<T> implements List<T> {
 ...
 public Iterator<T> iterator() {
 class MonIterator implements Iterator<T> {
 public boolean hasNext() {...}
 public T next() {...}
 public void remove() {...}
 }
 return new MonIterator()
 }
}
```

En plus des membres du type englobant, accès aux autres déclarations du bloc (notamment variables locales<sup>1</sup>).

1. Oui, mais seulement si **effectivement finales**... : si elles ne sont jamais ré-affectées.

**La définition de classe se place comme une expression dont la valeur est une instance<sup>1</sup> de la classe.**

```
class MaListe<T> implements List<T> {
 ...
 public Iterator<T> iterator() {
 return /* de là */ new Iterator<T>() {
 public boolean hasNext() {...}
 public T next() {...}
 public void remove() {...}
 } /* à là */
 }
}
```

1. La seule instance.

**Le mot-clé `var`<sup>1</sup> permet de faire des choses sympas avec les classes anonymes :**

```
// Création d'un objet singleton utilisable sans déclarer de classe nommée ou
// d'interface :
var plop = new Object() { int x = 23; };
System.out.println(plop.x);
```

Sans `var` il aurait fallu écrire le type de `plop`. En l'occurrence le plus petit type dénotable connu ici est **Object**.

Or la classe `Object` n'a pas de champ `x`, donc `plop.x` ne compilera pas.

**cas particulier de classe locale avec syntaxe allégée**  
→ comme classes locales, accès aux déclarations du bloc<sup>1</sup> ;

- déclaration "en ligne" : c'est syntaxiquement une expression, qui s'évalue comme une instance de la classe déclarée;
- déclaration de classe sans donner de nom ⇒ instanciable une seule fois  
→ c'est une classe singleton;
- autre restriction : un seul supertype direct<sup>2</sup> (dans l'exemple : `Iterator`).

Question : comment exécuter des instructions à l'initialisation d'une classe anonyme alors qu'il n'y a pas de constructeur ?  
→ Réponse : utiliser un "bloc d'initialisation" ! (Au besoin, cherchez ce que c'est !)

`x -> System.out.println(x)`.

1. Avec la même restriction : variables locales effectivement finales seulement.
2. Une classe peut généralement, sauf dans ce cas, implémenter de multiples interfaces.

1. Remplaçant un type dans une déclaration, pour demander d'inférer le type automatiquement.

|                                                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>class/interface/enum TypeEnglobant {<br/>    static int x = 4;<br/>    static class/interface/enum TypeMembre { static int y = x; }<br/>    static int z = TypeMembre.y;<br/>}</code> | <p>Le contexte interne du type imbriqué contient toutes les définitions du contexte externe. Ainsi, sont accessibles directement <u>sans chemin</u><sup>1</sup> :</p> <ul style="list-style-type: none"> <li>• dans tous les cas : les <u>membres statiques</u> du type englobant;</li> <li>• pour les classes <u>membres non statiques</u> et classes <u>locales</u> dans bloc non statique ;</li> <li>• tous les <u>membres non statiques</u> du type englobant;</li> <li>• pour les <u>classes locales</u> : les <u>définitions locales</u><sup>2</sup>.</li> </ul> <p>Réciroque fausse : depuis l'<u>extérieur</u> de TI, accès au membre <code>y</code> de TI en écrivant <code>TI.y</code>.</p> <ol style="list-style-type: none"> <li>1. sauf s'il faut lever une ambiguïté</li> <li>2. seulement effectivement finales pour les variables...</li> </ol> |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

- Dans une interface englobante, les types membres sont **public** et **static**.
  - Dans les classes locales (et anonymes), on peut utiliser les variables locales du bloc seulement si elles sont **effectivement finales** (c.-à-d. déclarées **final**, ou bien jamais modifiées).
- Explication : l'instance de la classe locale peut "survivre" à l'exécution du bloc. Donc elle doit contenir une copie des variables locales utilisées. Or les 2 copies doivent rester cohérentes → modifications interdites.
- Une alternative non retenue : stocker les variables partagées dans des objets dans le tas, dont les références seraient dans la pile. On pourrait aisément programmer ce genre de comportement au besoin.
- Les classes internes<sup>2</sup> ne peuvent pas contenir de membres statiques (à part attributs **final**).

La raison est le décalage entre ce qu'est censé être une classe interne (préndue dépendance à un certain contexte dynamique) et son implémentation (classe statique toute bête : ce sont en réalité les instances de la classe interne qui contiennent une référence vers, par exemple, l'instance englobante).

Une méthode statique ne pourrait donc pas accéder à ce contexte dynamique, rompant l'illusion recherchée.

1. Nécessairement et implicitement.
2. Tous les types imbriqués sauf les classes membres statiques

