Objets

Vision haut niveau

- Un **objet** est “juste” un nœud dans le graphe de communication qui se déploie quand on exécute un programme OO.
- Il est caractérisé par une certaine **interface** [1] de communication.
- Un objet a un **état** (modifiable ou non), en grande partie caché vis-à-vis des autres objets (l’état est **encapsulé**).
- Le graphe de communication est dynamique, ainsi, les objets naissent (sont instanciés) et meurent (sont détruits, désalloués).

... oui mais concrètement ?

1. Au moins implicitement : ici, “interface” ne désigne pas forcément la construction `interface` de Java.

Objets

Vision bas niveau (en Java)

Object = entité...

- caractérisée par un enregistrement contigü de données typées (**attributs** [1])
- accessible via une référence [2] vers cet enregistrement ;
- manipulable/interrogeable via un ensemble de **méthodes** qui lui est propre.

La variable référence `Personne toto` référence ⟼ l'objet

classe de l'objet :	réf. ↦ `Personne.class`
`int age`	42
`String nom`	réf. ↦ chaîne `"Dupont"`
`String prenom`	réf. ↦ chaîne `"Toto"`
...	
`boolean marie`	`true`

1. On dit aussi champs, comme pour les **struct** de C/C++.
Pour la représentation mémoire, un objet et une instance de **struct** sont similaires.
2. Il s'agit en vrai d'un pointeur. Les références de C++ sont un concept (un peu) différent.

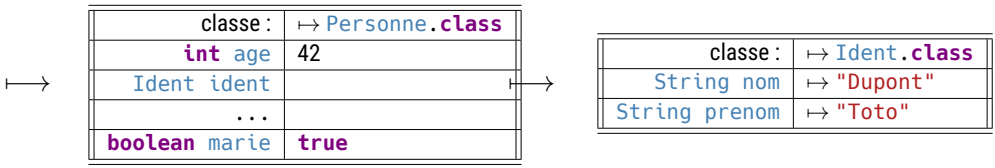
Objets

Quelle bonne vision bas niveau ?

Cette dernière vision est en fait réductrice.

En POO, on veut **s'abstraire** de l'implémentation concrète des concepts.

À service égal, les objets-`Personne` peuvent aussi être représentés ainsi :



Les méthodes seraient écrites différemment mais, à l'usage, cela ne se verrait pas. [1]

Pourtant cela aurait encore du sens de parler d'objets-`Personne` contenant des propriétés `nom` et `prenom`.

1. À condition qu'on n'utilise pas directement les attributs. D'où l'intérêt de les rendre privés !

Objets

Autre vision bas niveau, plus en phase avec le haut niveau

- **Objet** = ensemble d'informations regroupées en un certain nombre d'enregistrements contigüs, se référençant les uns les autres, tel que tout est accessible depuis un enregistrement principal [1].
- C'est donc un graphe orienté connexe dont les nœuds sont des enregistrements et les arcs les référencements.
  L'enregistrement principal est une origine de ce graphe.
- Les informations stockées dans ce graphe fournissent les services (méthodes) prévus par le type (interface) de l'objet.

Si nécessaire, on peut distinguer cette notion de celle d'« objet simple » (= **struct**), en utilisant l'expression **graphe d'objet**.

1. l'"objet" visible depuis le reste du programme

Objets

Autre vision bas niveau, plus en phase avec le haut niveau

Question : où arrêter le graphe d'un objet ?

- 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 ! [1]
- 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 ?

Cela n'est pas anodin :

- 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 ?

Cette discussion a trait aux notions d'encapsulation et de composition. À suivre !

1. En effet : les enregistrements non référencés par le programme, sont assez vite détruits par le GC.

Classes

Langages à prototypes, langages à classes

- **Besoin :** créer de nombreux objets similaires (même interface, même schéma de données).
- **2 solutions → 2 familles de LOO :**
  - **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).

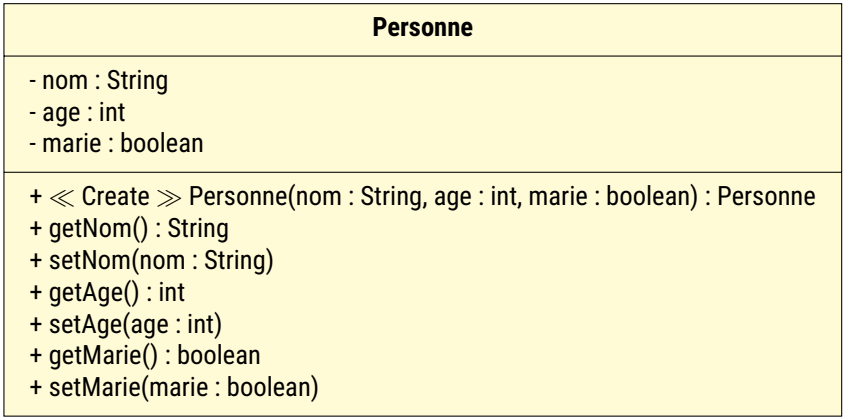
→ l'existence de classes n'est pas une nécessité en POO

Pour l'objet juste donné en exemple, la classe `Personne` pourrait être :

```java
public class Personne {
    // attributs
    private String nom; private int age; private boolean marie;

    // constructeur
    public Personne(String nom, int age, boolean marie) {
        this.nom = nom; this.age = age; this.marie = marie;
    }

    // méthodes (ici : accesseurs)
    public String getNom() { return nom; }
    public void setNom(String nom) { this.nom = nom; }

    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }

    public boolean getMarie() { return marie; }
    public void setMarie(boolean marie) { this.marie = marie; }
}
```

Classes

Exemple (le même en UML)



Classes

Points de vue pertinents pour Java

Classe = patron/modèle/moule/... pour définir des objets similaires [1].

Autres points de vue :

Classe =

- 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]

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.

Classes

Autres points de vue (non-OO) :

Classe =

- 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 instanciable [1]
  exemples de classes non instanciables du JDK : `System`, `Arrays`, `Math`, ...

Les aspects ci-dessus sont pertinents en Java, mais ne retenir que ceux-ci serait manquer l'essentiel : **i.e. : classe** = **concept de POO**.

1. Java force à tout définir dans des classes → encourage cet usage détourné de la construction `class`.

Instanciation

De la classe à l'objet

- Une classe permet de "fabriquer" plusieurs objets selon un même modèle : les **instances** [1] de la classe.
- Ces objets ont le même type, dont le nom est celui de la classe.
- La fabrication d'un objet s'appelle **l'instanciation.** Celle-ci consiste à
  - réserver la mémoire (≃ `malloc` en C)
  - initialiser les données [2] de l'objet
- On instancie la classe `Truc` via l'expression "`new Truc(params)`", dont la valeur est une référence vers un objet de type `Truc` nouvellement créé. [3]

1. En POO, "instance" et "objet" sont synonymes. Le mot "instance" souligne l'appartenance à un type.
2. En l'occurence : les attributs d'instance déclarés dans cette classe.
3. Ainsi, on note que le type défini par une classe est un type référence.

Instanciation

Constructeurs (1)

Constructeur : fonction [1] servant à construire une instance d'une classe.

- **Déclaration :**

```
MaClasse(/* paramètres */) {
  // instructions ; ici "this" désigne l'objet en construction
}
```

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.
- Appel toujours précédé du mot-clé `new` :

```
MaClasse monObjet = new MaClasse(... parametres... );
```

Cette instruction déclare un objet `monObjet`, crée une instance de `MaClasse` et l'affecte à `monObjet`.

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).

Il est possible de :

- définir plusieurs constructeurs (tous le même nom → cf. surcharge) ;
- 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 défaut [3].

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.

Membres d'une classe

Le **corps** d'une classe `C` consiste en une séquence de définitions : constructeurs [1] et **membres** de la classe.

Plusieurs catégories de membres : attributs, méthodes et types membres [2].

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` existe et ailleurs en écrivant « `recepteurDeTypeC.m` ».
- 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".

Membres d'une classe

Exemple


```java
public class Personne {
    // attributs
    public static int derNumINSEE = 0;
    public final NomComplet nom;
    public final int numInsee;

    // constructeur
    public Personne(String nom, String prenom) {
        this.nom = new NomComplet(nom, prenom);
        this.numInsee = ++derNumINSEE;
    }

    // méthode
    public String toString() {
        return String.format("%s %s (%d", nom.nom, nom.prenom, numInsee);
    }

    // et même... classe imbriquée !
    public static final class NomComplet {
        public final String nom;
        public final String prenom;

        private NomComplet(String nom, String prenom) {
            this.nom = nom;
            this.prenom = prenom;
        }
    }
}
```

Membres d'une classe

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

Contexte (associé à tout point du code source) :

- dans une définition [1] 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 classe [3] (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

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 classe [1].

	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`).
type membre	juste une classe/interface définie à l'intérieur d'une autre classe (à des fins d'encapsulation).	comme statique, mais instances contenant une référence vers instance de la classe englobante.

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.

Membres statiques et membres non-statiques

Zoom sur le cas des attributs

Qu'affiche le programme suivant ?

```
class Element {
    private static int a = 0; private int b = 1;
    public void plusUn() { a++; b++; }
    @Override public String toString() { return "" + a + b; }
}

public class Compter {
    private static Element e = new Element(), f = new Element();
    public static void main(String [] args) {
        printall(); e.plusUn(); printall(); f.plusUn(); printall();
    }
    private static void printall() { System.out.println("e : " + e + " et f : " + f); }
}
```

Membres statiques et membres non-statiques

Zoom sur le cas des attributs

Qu'affiche le programme suivant ?

```
class Element {
    private static int a = 0; private int b = 1;
    public void plusUn() { a++; b++; }
    @Override public String toString() { return "" + a + b; }
}

public class Compter {
    private static Element e = new Element(), f = new Element();
    public static void main(String [] args) {
        printall(); e.plusUn(); printall(); f.plusUn(); printall();
    }
    private static void printall() { System.out.println("e : " + e + " et f : " + f); }
}
```

Réponse :

```
e : 01 et f : 01
e : 12 et f : 11
e : 22 et f : 22
```

Membres statiques et membres non-statiques

Zoom sur le cas des méthodes

Remarque, on peut réécrire une méthode statique comme non statique de même comportement, et vice-versa :

```
class C { // ici f et g font la même chose
    void f() { instr(this); } // exemple d'appel : x.f()
    static void g(C that) { instr(that); } // exemple d'appel : C.g(x)
}
```

Mais différences essentielles :

- `f`, pour que `this` soit de type `C`, doit être déclarée dans `C` alors qu'il n'y a pas de relation entre le lieu de déclaration de `g` et le type de `that`.
  → Conséquences en termes de visibilité/encapsulation.
- Les appels `x.f()` et `C.g(x)` sont équivalents si `x` est instance directe de `C`.
  Mais c'est faux si `x` est instance de `D`, sous-classe de `C` redéfinissant `f` (cf. héritage), car la version redéfinie sera appelée : f est sujette à la liaison dynamique.

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 ;
- si appel à constructeur auxiliaire, nécessairement en première instruction ;
- obligation de retourner une nouvelle instance → pas de contrôle d'instances [1] ;
- 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
    ...
    // la fabrique :
    public static C of(D arg) {
        if (arg ...) return new CImpl1(arg);
        else if (arg ...) return ...
        else return ...
    }
}
```

```
final class CImpl1 extends C { // implémentation
    package-private (possible aussi : classe
    imbriquée privée)
    ...
    // constructeur package-private
    CImpl1(D arg) { ... }
}
```

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

Encapsulation

Concept essentiel de la POO

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 clients [1] (= 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'intéractions 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.

Encapsulation

Exemple

Est-il vrai que « le $n^{\text{ième}}$ appel à `next` retourne le $n^{\text{ième}}$ 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`

$\rightarrow$ on ne peut rien prouver !

Encapsulation

Exemple

Est-il vrai que « le $n^{\text{ième}}$ appel à `next` retourne le $n^{\text{ième}}$ 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 !

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]

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

Encapsulation

Remarque

- 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

Niveaux de visibilité : private, 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 [1]	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
}
```

1. voir héritage

Encapsulation

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

Notion de visibilité : s'applique aussi aux déclarations de premier niveau [1].

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.

Encapsulation

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'API [1] de la classe.
- → vous devez donc **la documenter** [2] (EJ3 Item 56)
- → et vous vous engagez **à ne pas modifier** [3] sa spécification [4] 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
3. On peut modifier si ça va dans le sens d'un renforcement compatible.
4. Et, évidemment, à faire en sorte que le comportement réel respecte la spécification !
5. 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 !

Encapsulation

Niveaux de visibilité : avertissements

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 existant [1] → garantie faible.
- `protected` → de même et, en +, toute sous-classe, n'importe où, voit la définiton.
- **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 réflexion [2].

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

1. Même à une dépendance tierce, même sans recompilation. 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 !

Encapsulation

Java Platform Module System (JPMS, Java $\geq$ 9)

- 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_depend;
    exports nom_d_un_package_defini_ici;
}
```

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`.

Encapsulation

Accesseurs (get, set, ...) et propriétés (1)

- Pour les classes publiques, il est recommandé [1] de mettre les attributs en **`private`** et de donner accès aux données de l'objet en définissant des méthodes **`public`** appelées **accesseurs**.
- Par convention, on leur donne des noms explicites :
  - **`public`** `T getX()` [2] : retourne la valeur de l'attribut `x` (“**getteur**”).
  - **`public void`** `setX(T nx)` : affecte la valeur `nx` a l'attribut `x` (“**setteur**”).
- Le couple `getX` et `setX` définit la **propriété** [3] `x` de l'objet qui les contient.
- Il existe des propriétés en lecture seule (si juste getteur) et en lecture/écriture (getteur et setteur).

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.

Encapsulation

Accesseurs (get, set, ...) et propriétés (2)

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

```
// propriété "numberOfFingers" :
public 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 public [1].
  Ainsi, quand cela sera fait, il ne sera **pas nécessaire de modifier les autres classes** qui accèdent à la propriété.

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

Encapsulation

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

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;
    }
}
```

Encapsulation

Accesseurs (get, set, ...) et propriétés (4)

Exemple : propriété en lecture seule avec évaluation paresseuse.

```
public final class Entier {
    public Entier(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(Outils.factorise(valeur)); // <- calcul
             coûteux, à  n'effectuer que si nécessaire
        return diviseurs;
    }
}
```

Encapsulation

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

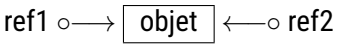
Comportements envisageables pour `get` et `set` :

- 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 notifie les objets observateurs lors des modifications ;
- vétoabilité : le setteur n'a d'effet que si aucun objet (dans une liste connue de "véto-eurs") ne s'oppose au changement ;
- ...

Encapsulation

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

Aliasing = existence de références multiples vers un même objet.



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 :

Cela revient [1] à laisser l'attribut en `public`, puisque le détenteur de cette référence peut faire les mêmes manipulations sur cet objet que la classe contenant l'attribut.

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 : 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; }
}
```

```
class C {
    private Data d;
    public void setData(Data d) {
        this.d = d;
    }
}

class D {
    private final Data d;
    public B(Data d) { this.d = d.copy(); }
    public void useData() {
        Client.use(d);
    }
}
```

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

Encapsulation

Aliasing : comment l'empêcher.

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

```
class A {
    // Mettre les attributs sensibles en private :
    private Data data;
    // Et 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.

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 $\rightarrow$ une copie directe suffit.
  - Si type mutable $\rightarrow$ 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'objet [3]).

---

1. La relation d'égalité est celle donnée par la méthode `equals`.
2. Type **immuable** (*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.
Contraire : **mutable** (*mutable*).
3. Il savoir en quoi consiste le graphe de l'objet, sinon la notion de copie profonde reste ambiguë.

Copie défensive

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


```java
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 primitif, 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 impossible [1] 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. Sauf à utiliser la réflexion... mais dans le cadre du JPMS, il ne faut pas trop compter sur celle-ci.

Immuabilité...

... 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 classe [1] dont la documentation dit qu'elle l'est.

En pratique, les 8 types primitfs (**`boolean`**, **`char`**, **`byte`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**) se comportent aussi [2] comme des types immuables [3].

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.

Encapsulation

Aliasing : pourquoi ce phénomène peut nous gêner plus tard...

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 concurrentes [1] de l'objet *alias*é, 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 concurrente.
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.

Encapsulation

Aliasing : remarque sans rapport avec l'encapsulation

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 possilité 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** ?

Encapsulation

Aliasing, est-ce toujours « mal » ?

Pour conclure sur l'*aliasing*.

Il n'y a pas que des inconvénients 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.

Types imbriqués

Qu'est-ce que c'est ?

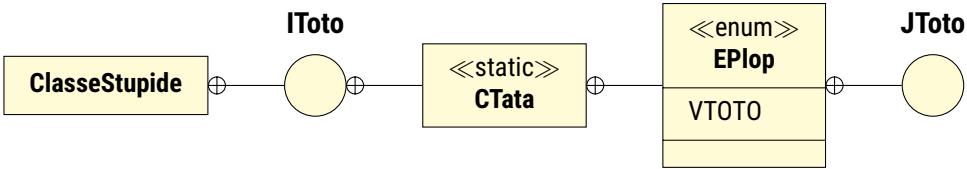
Java permet de définir un **type (classe ou interface) imbriqué** [1] à l'intérieur d'une autre définition de type (dit **englobant** [2]) :

```
public class ClasseStupide {
    public static interface IToto {
        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

Types imbriqués

En UML



Notez la forme et le sens de la « flêche ».

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(AA.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)
    }
}
```

Types imbriqués

Pour quoi faire ?

- définitions du contexte englobant incluses dans contexte imbriqué (sans chemin).
- Type englobant et types membres peuvent accéder aux membres **`private`** 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 {
        private static void fIB() { }
    }

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

Plusieurs sortes de types imbriqués

Définitions et classification

Classification des types imbriqués/*nested types* [1]

- **types membres statiques**/*static member classes* [2] : types définis directement dans la classe englobante, définition précédée de `static`
- **classes internes**/*inner classes* : les autres types imbriqués (toujours des classes)
  - **classes membres non statiques**/*non-static member classes* [3] : définies directement dans le type englobant
  - **classes locales**/*local classes* : définies dans une méthode avec la syntaxe habituelle (`class NomClasseLocale { /*contenu */}`)
  - **classes anonymes**/*anonymous classes* : définies “à la volée” à l’intérieur d’une expression, afin d’instancier un objet unique de cette classe : `new NomSuperTypeDirect(){ /*contenu */}`.

1. J’essaye de suivre la terminologie de la JLS... traduite, puis complétée par la logique et le bon sens.
2. La JLS les appelle *static nested classes*... oubliant que les interfaces membres existent aussi !
3. parfois appelées juste *inner classes* ; pas de nom particulier donné dans la JLS.

Types imbriqués

Exemple de type membre statique

La définition de classe prend la place d'une déclaration de membre du type englobant et est précédée de **`static`**.

```
class MaListe<T> implements List<T> {
    private static class MonIterateur<U> implements Iterator<U> {
        // ces méthodes travaillent sur les attributs de listeBase
        private final MaListe listeBase;
        public MonIterateur(MaListe l) { listeBase = l; }
        public boolean hasNext() {...}
        public U next() {...}
        public void remove() {...}
    }

    ...

    public Iterator<T> iterator() { return new MonIterateur<T>(this); }
}
```

On peut créer une instance de `MonIterateur` depuis n'importe quel contexte (même statique) dans `MaListe` avec juste "**`new`** `MonIterateur<_>(_)`".

Types imbriqués

Exemple de classe membre non statique

Définition similaire au cas précédent, mais sans le mot-clé **`static`**.

```
class MaListe<T> implements List<T> {
    private class MonIterateur implements Iterator<T> {
        // ces méthodes utilisent les attributs non statiques de MaListe directement
        public boolean hasNext() {...}
        public T next() {...}
        public void remove() {...}
    }
    ...
    public Iterator<T> iterator() {
        return new MonIterateur(); // possible parce que iterator() n'est pas statique
    }
}
```

- Pour créer une instance de `MaListe<String>.MonIterateur`, il faut évaluer “`new MonIterateur()`” dans le contexte d’une instance de `MaListe<String>`.
- Si on n’est pas dans le contexte d’une telle instance, on peut écrire “`x.new MonIterateur()`” (où `x` instance de `MaListe<String>`).

Types imbriqués

Accès à l'instance imbriquée et à l'instance englobante : **this** et `Outer.this`

Soit `TI` un type imbriqué dans `TE`, type englobant. Alors, dans `TI` :

- **`this`** désigne toujours (quand elle existe) l'instance courante de `TI` ;
- `TE.this` désigne toujours (quand elle existe) l'**instance englobante**, c.-à-d. l'instance courante de `TE`, c.-à-d. :
  - si `TI` classe membre non statique, la valeur de **`this`** dans le contexte où l'instance courante de `TI` a été créée. **Exemple :**

```
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"
```

  - si `TI` classe locale, la valeur de **`this`** dans le bloc dans lequel `TI` a été déclarée.

La référence `TE.this` est en fait stockée dans toute instance de `TI` (attribut caché).

Types imbriqués

Exemple de classe locale

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 MonIterateur implements Iterator<T> {
            public boolean hasNext() {...}
            public T next() {...}
            public void remove() {...}
        }
        return new MonIterateur()
    }
}
```

En plus des membres du type englobant, accès aux autres déclarations du bloc (notamment variables locales [1]).

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

Types imbriqués

Exemple de classe anonyme

La définition de classe est une expression dont la valeur est une instance [1] 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.

Classes anonymes

Classe anonyme =

- cas particulier de classe locale avec syntaxe allégée
  → comme classes locales, accès aux déclarations du bloc [1] ;
- 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 [2] (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.)

Syntaxe encore plus concise : lambda-expressions (cf. chapitre dédié), par ex.
`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.

Classes anonymes

Astuce avec Java $\geq$ 10

Le mot-clé **`var`** [1] permet de faire des choses sympas avec les classes anonymes :

```
// Création d'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 compilerait pas.

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

Types imbriqués

Accessibilité du contexte englobant


```
class/interface/enum TypeEnglobant {
    static int x = 4;
    static class/interface/enum TypeMembre { static int y = x; }
    static int z = TypeMembre.y;
}
```

Le contexte interne du type imbriqué contient toutes les définitions du contexte externe. Ainsi, sont accessibles directement (sans chemin [1]) :

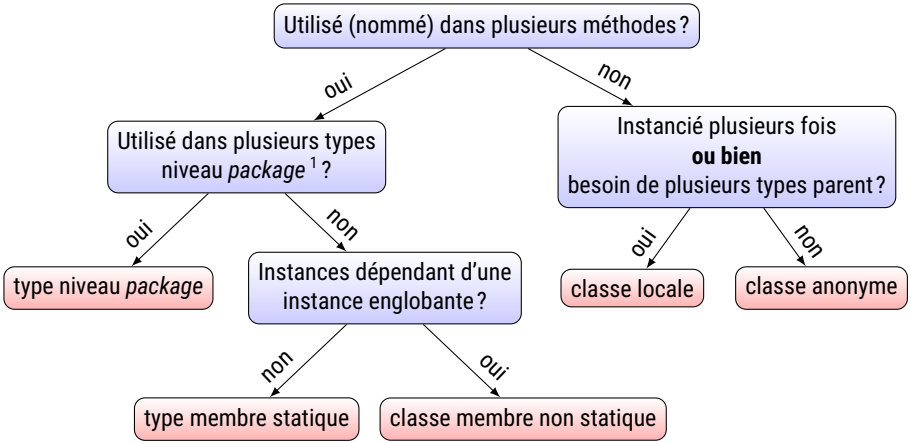
- dans tous les cas : les membres statiques du type englobant ;
- pour les classes membres non statiques et classes locales dans bloc non statique : tous les membres non statiques du type englobant ;
- pour les classes locales : les définitions locales [2].

Réciproque fausse : depuis l'extérieur de `TI`, accès au membre `y` de `TI` en écrivant `TI.y`.

1. sauf s'il faut lever une ambiguïté
2. seulement effectivement finales pour les variables...

Plusieurs sortes de types imbriqués

D'accord, mais lequel choisir ? [2]



1. C'est à dire non imbriqués, définis directement dans les fichiers `.java`.
2. Cf. *Effective Java 3rd edition*, Item 24 : *Favor static member classes over nonstatic*.

Types imbriqués

Remarques et limitations diverses

- Dans une interface englobante, les types membres sont [1] **`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 [2] 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 (prétendue 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