Compléments en Programmation Orientée Objet

Aldric Degorre

Version `2022.00.01` du 16 septembre 2022

En remerciant mes collaborateurs des années passées, qui ont aidé à élaborer ce cours et à le faire évoluer.

Aspects pratiques

Horaires, enseignants

- **Volume horaire :**
  - 6x2h de CM (dates : 16/9, 23/9, 7/10, 21/10, 18/11, 9/12)
  - 12x2h de TP (chaque semaine, à partir de la semaine du 20/9, pause semaine du 31/10)
- Intervenants :
  - CM : Aldric Degorre
  - TP : **Chargés de TP :** Emmanuel Bigeon [1] (jeudi 16h15), Wael Boutglay [2] (mercredi 16h15), Wieslaw Zielonka [3] (mardi 8h30, jeudi 10h45), Aldric Degorre [4] (mardi 14h00)

1. bigeon@irif.fr
2. boutglay@irif.fr
3. zielonka@irif.fr
4. adegorre@irif.fr

Aspects pratiques

TP

- Utilisez les machines de TP de l'UFR...
- ... ou bien votre portable avec
  - le JDK 17 ou plus
  - votre IDE favori [1] (Eclipse, IntellIJ, NetBeans, VSCode, vi, emacs...)

Si vous utilisez les machines le l'UFR, asssurez-vous d'avoir vos identifiants pour vos comptes à l'UFR d'Informatique en arrivant au premier TP ! [2]

1. Celui avec lequel vous êtes efficace !
2. Vous avez dû les obtenir par email. Sinon, contactez les administrateurs du réseau de l'UFR : `jmm@informatique.univ-paris-diderot.fr`, `pietroni@informatique.univ-paris-diderot.fr` (batiment Sophie Germain, bureau 3061).

Aspects pratiques

2 comptes informatiques !

En fait, pouvez avoir besoin de 2 comptes :

- le compte U-Paris pour accéder à moodle [1] (indispensable !)
  Les support de cours et TP, les annonces et les rendus seront sur Moodle !
  Vérifiez que vous êtes bien inscrit. (ou contactez-moi au plus vite !)
- le compte de l'UFR d'info pour utiliser les machines de TP (recommandé !)

1. https://moodle.u-paris.fr/course/view.php?id=1650

Aspects pratiques

Modalités de contrôle

(Sous réserve... mais vous seriez prévenus tôt le cas échéant.)

- En première session :
  Un projet de programmation en binôme $\rightarrow$ 50% de la note,
  Interrogations et TPs rendus $\rightarrow$ 50% de la note,
- En session de rattrapage : un nouveau projet. Note = 60% projet2 + 40% CC.

Bibliographie

- Ce cours [1]
- Mes sources (livresques...) :
  - Effective Java, 3rd edition (Joshua Bloch)
  - Java Concurrency in Practice (Brian Goetz)
  - Design Patterns : Elements of Reusable Object-Oriented Software (Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides... autrement connus sous le nom “the Gang of Four”)
- Plein de ressources sur le web, notamment la doc des API : `https://docs.oracle.com/en/java/javase/17/docs/api/`.

1. Sa forme pourrait changer : si j’ai le temps, j’espère le transformer en, d’une part, un petit livre et, d’autre part, des transparents résumés.

On a déjà « fait Java », pourquoi ce cours ?

- Vous connaissez la syntaxe de Java.
- Vous savez même écrire des programmes [1] qui compilent.
  Évidemment ! Vous avez passé POO-IG !
- Vous savez brillamment résoudre un problème présenté en codant en Java.
  Vous avez fait un très joli projet en PI4, n'est-ce pas ?
- Vous pensez savoir programmer ... **vraiment ?**

1. même dans le style objet !

On a déjà « fait Java », pourquoi ce cours ?

Sauriez-vous encore ?

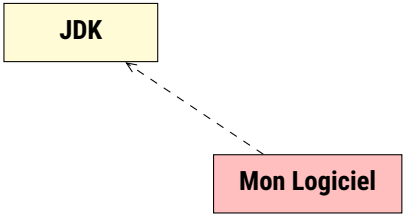
- Supprimer ce fameux bug que vous n'aviez pas eu le temps de corriger avant la deadline de PI4 l'année dernière ?
- Remplacer une des dépendances de votre projet par une nouvelle bibliothèque, sans tout modifier et ajouter 42 nouveaux bugs ?
- Ajouter une extension que vous n'aviez pas non plus eu le temps de faire.

Et auriez-vous l'audace d'imprimer le code et de l'afficher dans votre salon ?

(Ou plus prosaïquement, de le publier sur GitHub pour y faire participer la communauté ?)

On a déjà « fait Java », pourquoi ce cours ?

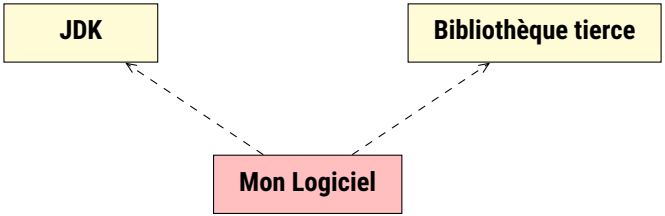
La vraie vie ?



Jusqu'à présent, vos projets ressemblaient à ça (un logiciel exécutable qui ne dépend que du JDK).

On a déjà « fait Java », pourquoi ce cours ?

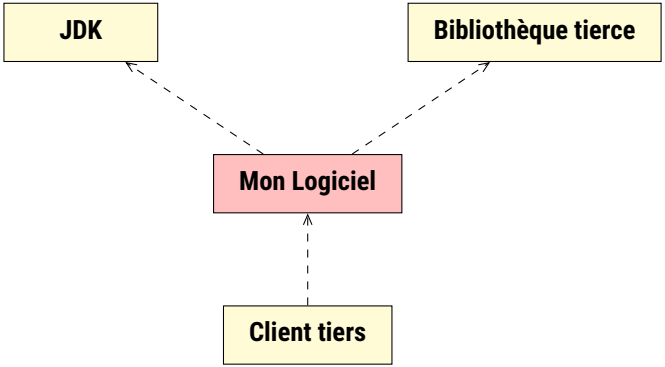
La vraie vie ?



À la rigueur, à ça (dépendance à une ou des bibliothèques tierces).

On a déjà « fait Java », pourquoi ce cours ?

La vraie vie ?



Mais souvent, dans la vraie vie [1], on fait aussi ça : on programme une bibliothèque réutilisable par des tiers.

1. mais rarement en L2...

On a déjà « fait Java », pourquoi ce cours ?

La vraie vie ?

Problème : tous ces composants logiciels évoluent (versions successives).

Comment s'assurer alors :

- **que votre programme « résiste » aux évolutions de ses dépendances ?**
- **qu'il fournit bien à ses clients le service annoncé, dans toutes les conditions annoncées comme compatibles** [1]
- **que les évolutions de votre programme ne « cassent » pas ses clients ?**

Pensez-vous que le projet rendu l'année dernière garantit tout cela ?

1. Il faut essayer de penser à tout. Notamment, la bibliothèque fonctionne-t-elle comme prévu quand elle est utilisée par plusieurs *threads* ?

On a déjà « fait Java », pourquoi ce cours ?

→ Tout cela n'est possible qu'en respectant une certaine « hygiène » [1].

La programmation orientée objet permet une telle hygiène :

- à condition qu'elle soit réellement pratiquée
- de respecter certaines bonnes pratiques [2]

1. Si vous avez réussi vos projets de programmation sans effort d'hygiène, vous n'en ressentez peut-être pas encore le besoin.
Mais il faut avoir conscience du contexte bien particulier de ces projets.
2. Notamment utiliser des patrons de conception.

À propos de ce cours

Objectifs, contexte

Objectif : explorer les concepts de la programmation orientée objet (POO) au travers du langage Java et enseigner les principes d'une programmation fiable, pérenne et évolutive.

Contexte :

- Ce cours fait suite à POO-IG et PI4 en L2.
- Il introduit plusieurs des cours de Master : C++, Android, concurrence, POCA...
- Liens avec Génie logiciel et PF.

Contenu :

- quelques rappels [1], avec approfondissement sur certains thèmes ;
- thème supplémentaire : la programmation concurrente (*threads* et APIs les utilisant)
- **surtout**, nous insisterons sur la POO (objectifs, principes, techniques, stratégies et patrons de conception, bonnes pratiques ...).

1. Mal nécessaire pour s'assurer d'une terminologie commune. Si vous en voulez encore plus, relisez vos notes de l'année dernière !

À propos de ce cours

Pourquoi Java ?

Pourquoi un autre “cours de Java” ?

- Java convient tout à fait pour illustrer les concepts OO. [1] [2]
- $(n+1)^{\text{ième}}$ contact avec Java $\rightarrow$ économie du ré-apprentissage d'une syntaxe $\rightarrow$ il reste du temps pour parler de POO.
- C'est aussi l'occasion de perfectionner la maîtrise du langage Java (habituellement, il faut plusieurs “couches” !)
- Et Java reste encore très pertinent dans le « monde réel ».

Cela dit, d'autres langages [3] illustrent aussi ce cours (C, C++, OCaml, Scala, Kotlin, ...).

1. On pourra disserter sur le côté langage OO “impur” de Java... mais Java fait l'affaire !
2. Les autres langages OO pour la JVM conviendraient aussi, mais ils sont moins connus.
3. Eux aussi installés en salles de TP. Soyez curieux, expérimentez !

À propos de ce cours

Notre méthode

Pour chaque sujet abordé :

- Se rappeler (L2) ou découvrir l'approche de base ;
- Voir ses limitations et risques ;
- Étudier les techniques avancées pour les pallier (souvent : *design patterns*) ;
- Discuter de leurs avantages et inconvénients.

La programmation orientée objet

Historique

- **Paradigme de programmation** inventé dans les années 1960 par Ole-Johan Dahl et Kristen Nygaard (Simula : langage pour simulation physique)...
- ... complété par Alan Kay dans les années 70 (Smalltalk), qui a inventé l'expression "*object oriented programming*".
- Premiers langages OO "historiques" : Simula 67 (1967 [1]), Smalltalk (1980 [2]).
- Autres LOO connus : C++, Objective-C, Eiffel, Java, Javascript, C#, Python, Ruby...

1. Simula est plus ancien (1962), mais il a intégré la notion de classe en 1967.
2. Première version publique de Smalltalk, mais le développement à commencé en 1971.

La programmation orientée objet

Principes

- **Principe de la POO :** des messages [1] s'échangent entre des objets qui les traitent pour faire progresser le programme.
- → **POO = paradigme centré sur la description de la communication entre objets.**
- Pour faire communiquer un objet a avec un objet b, il est nécessaire et suffisant de connaître les messages que b accepte : l'**interface** de b.
- Ainsi objets de même interface interchangeables → **polymorphisme**.
- Fonctionnement interne d'un objet [2] caché au monde extérieur → **encapsulation**.

Pour résumer : la POO permet de raisonner sur des **abstractions** des composants réutilisés, en ignorant leurs détails d'implémentation.

1. appels de **méthodes**
2. Notamment son état, représenté par des **attributs**.

La programmation orientée objet

Avantages et raisons du succès de la POO

La POO permet de découper un programme en composants

- peu dépendants les uns des autres (faible **couplage**)
  → code **robuste et évolutif**
  (composants testables et déboguables indépendamment et aisément remplaçables) ;
- réutilisables, au sein du même programme, mais aussi dans d'autres ;
  → facilite la création de logiciels de grande taille.

POO → (discutable) façon de penser naturelle pour un cerveau humain "normal" [1] : chaque entité définie représente de façon abstraite un concept du problème réel modélisé.

1. Non « déformé » par des connaissances mathématiques pointues comme la théorie des catégories (cf. programmation fonctionelle).

- **1992 :** langage Oak chez *Sun Microsystems* [1], dont le nom deviendra Java ;
- **1996 :** JDK 1.0 (première version de Java) ;
- **2009 :** rachat de *Sun* par la société *Oracle* ;

En 2022, Java est donc « dans la force de l'âge » (26 ans [2]), à comparer avec :

- même génération : Haskell : 32 ans, Python : 31 ans, JavaScript, OCaml : 26 ans
- anciens : C++ : 37 ans, C : 44 ans, COBOL : 63 ans, Lisp : 65 ans, Fortran : 68 ans...
- modernes : Scala : 18, Go : 13, Rust : 12, Swift : 8, Kotlin : 11, Carbon : 0...

Java est le 2er ou 3ème langage le plus populaire. [3]

1. Auteurs principaux : James Gosling et Patrick Naughton.
2. Je considère la version 1.0 de chaque langage.
3. Dans les classements principaux : TIOBE, RedMonk, PYPL, ...

Le haut du palmarès est généralement occupé par Java, C, Javascript et Python dans un ordre ou un autre...

Versions « récentes » de Java :

- **09/2021 :** Java SE 17, la version long terme actuelle ; [1]
- **03/2022 :** Java SE 18, la version actuelle ;
- **20/09/2022 :** Java SE 19, la prochaine version. Sort mardi prochain !

Ce cours utilise Java 17, mais :

- la plupart de ce qui y est dit vaut aussi pour les versions antérieures ;
- on ne s'interdit pas de s'interdire certaines nouveautés pour voir comment on verait sans ;
- on ne s'interdit pas de parler de ce qui arrive bientôt !

1. Depuis Java 9, une version "normale" sort tous les 6 mois et une à "support long terme", tous les 3 ans.

Java

Caractérisitiques principales

« Java » (Java SE) est en réalité une plateforme de programmation caractérisée par :

- le langage de programmation Java
  - orienté objet à classes,
  - à la syntaxe inspirée de celle du langage C [1],
  - au typage statique,
  - à gestion automatique de la mémoire, via son **ramasse-miettes** (*garbage collector*).
- sa machine virtuelle (**JVM** [2]), permettant aux programmes Java d'être multi-plateforme (le **code source** se compile en **code-octet** pour JVM, laquelle est implémentée pour nombreux types de machines physiques).
- les bibliothèques officielles du JDK (fournissant l'**API** [3] Java), très nombreuses et bien documentées (+ nombreuses bibliothèques de tierces parties).

1. C sans pointeurs et **struct** $\simeq$ Java sans objet
2. *Java Virtual Machine*
3. *Application Programming Interface*

Domaines d'utilisation :

- applications de grande taille [1] ;
- partie serveur « *backend* » des applications web (technologies *servlet* et JSP) [2] ;
- applications « desktop » [3] (via Swing et JavaFX, notamment) multiplateformes (grâce à l'exécution dans une machine virtuelle) ;
- applications mobiles (années 2000 : J2ME/MIDP ; années 2010 : Android) ;
- cartes à puces (via spécification Java Card).

1. Facilité à diviser un projet en petits modules, grâce à l'approche OO. Pour les petits programmes, la complexité de Java est, en revanche, souvent rebutante.
2. En concurrence avec PHP, Ruby, Javascript (Node.js) et, plus récemment, Go.
3. Appelées aujourd'hui "**clients lourds**", par opposition à ce qui tourne dans un navigateur web.

Mais :

- Java ne s'illustre plus pour les clients « légers » (dans le navigateur web) : les *applets* Java ont été éclipsées [1] par Flash [2] puis Javascript.
- Java n'est pas adapté à la programmation système [3].
  $\rightarrow$ C, C++ et Rust plus adaptés.
- Idem pour le temps réel [4].
- Idem pour l'embarqué [5] (quoique... cf. Java Card).

---

1. Le plugin Java pour le navigateur a même été supprimé dans Java 10.
2. ... technologie aussi en voie de disparition
3. APIs trop haut niveau, trop loin des spécificités de chaque plateforme matérielle.
Pas de gestion explicite de la mémoire.
4. Ramasse-miette qui rend impossible de donner des garanties temps réel. Abstractions coûteuses.
5. Grosse empreinte mémoire (JVM) + défauts précédents.

Question(s) de style

- Aucun programme n'est écrit directement dans sa version définitive.
- Il doit donc pouvoir être facilement modifié par la suite.
- Pour cela, ce qui est déjà écrit doit être **lisible et compréhensible**.
  - lisible par le programmeur d'origine
  - lisible par l'équipe qui travaille sur le projet
  - lisible par toute personne susceptible de travailler sur le code source (pour le logiciel libre : la Terre entière !)

Les commentaires [1] et la javadoc peuvent aider, mais rien ne remplace un code source bien écrit.

1. Si un code source contient plus de commentaires que de code, c'est en réalité assez "louche".

Question de goût ?

- “être lisible” $\rightarrow$ évidemment très subjectif
- un programme est lisible s’il est écrit tel qu’“on” a l’habitude de les lire
- $\rightarrow$ habitudes communes prises par la plupart des programmeurs Java (d’autres prises par seulement par telle ou telle organisation ou communauté)

Langage de programmation $\rightarrow$ comme une langue vivante !

Il ne suffit pas de connaître par cœur le livre de grammaire pour être compris des locuteurs natifs (il faut aussi prendre l’accent et utiliser les tournures idiomatiques).

Une hiérarchie de normes

Habitudes dictées par :

1. le compilateur (la syntaxe de Java [1])
2. le guide [2] de style qui a été publié par Sun en même temps que le langage Java (→ conventions à vocation universelle pour tout programmeur Java)
3. les directives de son entreprise/organisation
4. les directives propres au projet

… et ainsi de suite (il peut y avoir des conventions internes à un package, à une classe, etc.)

» et enfin... le bon sens ! [3]

Nous parlerons principalement du 2ème point et des conventions les plus communes.

1. L'équivalent du livre de grammaire dans l'analogie avec la langue vivante.
2. À rapprocher des avis émis par l'Académie Française ?
3. Mais le bon sens ne peut être acquis que par l'expérience.

Nommer les entités
(classes, méthodes, variables, ...)

Règles de capitalisation pour les noms (auxquelles on ne déroge pratiquement jamais) :

- ... de classes, interfaces, énumérations et annotations [1] ⟶ `UpperCamelCase`
- ... de variables (locales et attributs), méthodes ⟶ `lowerCamelCase`
- ... de constantes (**`static final`** ou valeur d'**`enum`**) ⟶ `SCREAMING_SNAKE_CASE`
- ... de packages ⟶ tout en minuscules sans séparateur de mots [2]. Exemple : `com.masociete.biblitothequetruc` [3].

⟶ rend possible de reconnaître à la première lecture quel genre d'entité un nom désigne.

1. c.-à-d. tous les types référence
2. "_" autorisé si on traduit des caractères invalides, mais pas spécialement encouragé
3. pour une bibliothèque éditée par une société dont le nom de domaine internet serait `masociete.com`

Nommer les entités

Codage, et langue

- **Se restreindre aux caractères suivants :**
  - `a-z`, `A-Z` : les lettres minuscules et capitales (non accentuées),
  - `0-9` : les chiffres,
  - `_` : le caractère soulignement (seulement pour `snake_case`).

  Explication :
  - `$` (dollar) est autorisé mais réservé au code automatiquement généré ;
  - les autres caractères ASCII sont réservés (pour la syntaxe du langage) ;
  - la plupart des caractères unicode non-ASCII sont autorisés (p. ex. caractères accentués), mais aucun standard de codage imposé pour les fichiers `.java`.[1]
- **Interdits :** commencer par `0-9` ; prendre un nom identique à un mot-clé réservé.
- **Recommandé :** Utiliser l'Anglais américain (pour les noms utilisés dans le programme **et** les commentaires **et** la javadoc).

1. Or il en existe plusieurs. En ce qui vous concerne : il est possible que votre PC personnel et celle de la salle de TP n'aient pas le même réglage par défaut → incompatibilité du code source.

Nommer les entités

Nature grammaticale (1)

Nature grammaticale des identifiants :

- types ($\rightarrow$ notamment noms des classes et interfaces) : nom au singulier
  ex : `String`, `Number`, `List`, ...
- classes-outil (non instanciables, contenu statique seulement) : nom au pluriel
  ex : `Arrays`, `Objects`, `Collections`, ... [1]
- variables : nom, singulier sauf pour collections (souvent nom pluriel) ; et booléens (souvent adjectif ou verbe au participe présent ou passé). ex :

```
int count = 0; // noun (singular)
boolean finished = false; // past participle
while (!finished) {
    finished = ...;
    ...
    count++;
    ...
}
```

1. attention, il y a des contre-exemples au sein même du JDK : `System`, `Math`... oh !

Nommer les entités

Nature grammaticale (2)

Les noms de méthodes contiennent généralement **un verbe**, qui est :

- `get` si c'est un accesseur en lecture ("getteur"); ex : `String getName();`
- `is` si c'est un accesseur en lecture d'une propriété booléenne;
  ex : `boolean isInitialized();`
- `set` si c'est un accesseur en écriture ("setteur");
  ex : `void getName(String name);`
- tout autre verbe, à l'indicatif, si la méthode retourne un booléen (méthode prédicat);
- à l'impératif [1], si la méthode sert à effectuer une action avec **effet de bord** [2]
  `Arrays.sort(myArray);`
- au participe passé si la méthode retourne une version transformée de l'objet, sans modifier l'objet (ex : `list.sorted()`).

1. ou infinitif sans le "*to*", ce qui revient au même en Anglais
2. c.-à-d. mutation de l'état ou effet physique tel qu'un affichage; cela s'oppose à fonction pure qui effectue juste un calcul et en retourne le résultat

Nommer les entités

Concision versus information

- Pour tout idenficateur, il faut trouver le bon compromis entre information (plus long) et facilité à l'écrire (plus court).
- Typiquement, plus l'usage est fréquent et local, plus le nom est court :
  ex. : variables de boucle
  ```
  for (int idx = 0; idx < anArray.length; idx++){ ... }
  ```
- plus l'usage est lointain de la déclaration, plus le nom doit être informatif
  (sont particulièrement concernés : classes, membres publics... mais aussi les paramètres des méthodes !)
  ex. : paramètres de constructeur
  ```
  Rectangle(double centerX, double centerY, double width, double length){ ... }
  ```

Toute personne lisant le programme s'attend à une telle stratégie → ne pas l'appliquer peut l'induire en erreur.

Nombre de caractères par ligne

- On limite le nombre de caractères par ligne de code. Raisons :
  - certains programmeurs préfèrent désactiver le retour à la ligne automatique [1] ;
  - même la coupure automatique ne se fait pas forcément au meilleur endroit ;
  - longues lignes illisibles pour le cerveau humain (même si entièrement affichées) ;
  - certains programmeurs aiment pouvoir afficher 2 fenêtres côte à côte.
- Limite traditionnelle : 70 caractères/ligne (les vieux terminaux ont 80 colonnes [2]).
  De nos jours (écrans larges, haute résolution), 100-120 est plus raisonnable [3].
- Arguments contre des lignes trop petites :
  - découpage trop élémentaire rendant illisible l'intention globale du programme ;
  - incitation à utiliser des identifiants plus courts pour pouvoir écrire ce qu'on veut en une ligne ($\rightarrow$ identifiants peu informatifs, mauvaise pratique).

1. De plus, historiquement, les éditeurs de texte n'avaient pas le retour à la ligne automatique.
2. Et d'où vient ce nombre 80 ? C'est le nombre du de colonnes dans le standard de cartes perforées d'IBM inventé en... 1928 ! Et pourquoi ce choix en 1928 ? Parce que les machines à écrire avaient souvent 80 colonnes... bref c'est de l'histoire très ancienne !
3. Selon moi, mais attention, c'est un sujet de débat houleux !

Indentation

- **Indenter** = mettre du blanc en tête de ligne pour souligner la structure du programme. Ce blanc est constitué d'un certain nombre d'**indentations**.
- En Java, typiquement, 1 indentation = 4 espaces (ou 1 tabulation).
- Le nombre d'indentations est égal à la profondeur syntaxique du début de la ligne $\simeq$ nombre de paires de symboles [1] ouvertes mais pas encore fermées. [2]
- Tout éditeur raisonnablement évolué sait indenter automatiquement (règles paramétrables dans l'éditeur). Pensez à demander régulièrement l'indentation automatique, afin de vérifier qu'il n'y a pas d'erreur de structure !

Exemple :

```
voici un exemple (
    qui n'est pas du Java;
    mais suit ses "conventions
        d'indentation"
)
```

1. Parenthèses, crochets, accolades, guillemets, chevrons, ...
2. Pas seulement : les règles de priorité des opérations créent aussi de la profondeur syntaxique.

Où couper les lignes

- On essaye de privilégier les retours à la ligne en des points du programme “hauts” dans l’arbre syntaxique ($\rightarrow$ minimise la taille de l’indentation).
  P. ex., dans “$(x + 2) * (3 - 9/2)$”, on préfèrera couper à côté de “$*$” $\rightarrow$

```
(x + 2)
* (3 - 9 / 2)
```

- Parfois difficile à concilier avec la limite de caractères par ligne $\rightarrow$ compromis nécessaires.
- $\rightarrow$ pour le lieu de coupure et le style d’indentation, essayez juste d’être raisonnable et consistent. Dans le cadre d’un projet en équipe, se référer aux directives du projet.

Taille des classes

Quelle est la bonne taille pour une classe ?

- Déjà, plusieurs critères de taille : nombre de lignes, nombre de méthodes, ...
- Le découpage en classes est avant tout guidé par l'abstraction objet retenue pour modéliser le problème qu'on veut résoudre.
- En pratique, une classe trop longue est désagréable à utiliser. Ce désagrément traduit souvent une décomposition insuffisante de l'abstraction. [1]
- Conseil : se fixer une limite de taille et décider, au cas par cas, si et comment il faut "réparer" les classes qui dépassent la limite (cela incite à améliorer l'aspect objet du programme).
- En général, pour un projet en équipe, suivre les directives du projet.

1. Le « S » de « SOLID » : *single reponsibility principle*/principe de responsabilité unique.

Taille des méthodes

- Pour une méthode, la taille est le nombre de lignes.
- Principe de responsabilité unique [1] : une méthode est censée effectuer une tâche précise et compréhensible.
  $\rightarrow$ Un excès de lignes
  - nuit à la compréhension ;
  - peut traduire le fait que la méthode effectue en réalité plusieurs tâches probablement séparables.
- Quelle est la bonne longueur ?
  - Mon critère [2] : on ne peut pas bien comprendre une méthode si on ne peut pas la parcourir en un simple coup d'œil
    $\rightarrow$ faire en sorte qu'elle tienne en un écran ($\sim$ 30-40 lignes max.)
  - En général, suivre les directives du projet.

1. Oui, là aussi !
2. qui n'engage que moi !

Nombre de paramètres des méthodes

Autre critère : le nombre de paramètres.

Trop de paramètres (>4) implique :

- Une signature longue et illisible.
- Une utilisation difficile (“ah mais ce paramètre là, il était en 5e ou en 6e position, déjà ?”)

Il est souvent possible de réduire le nombre de paramètres

- en utilisant la surcharge,
- ou bien en séparant la méthode en plusieurs méthodes plus petites (en décomposant la tâche effectuée),
- ou bien en passant des objets composites en paramètre

  ex : un `Point p` au lieu de `int x, int y`.

  Voir aussi : patron “monteur” (le constructeur prend pour seul paramètre une instance du `Builder`).

Et ainsi de suite

- Pour chaque composant contenant des sous-composants, la question “combien de sous-composants ?” se pose.
- “Combien de packages dans un projet (ou module) ?”
  “Combien de classes dans un package ?”
- Dans tous les cas essayez d’être raisonnable et homogène/consistent (avec vous-même... et avec l’organisation dans laquelle vous travaillez).

Commentaires

Plusieurs sortes de commentaires (1)

- En ligne :

```
int length; // length of this or that
```

Pratique pour un commentaire très court tenant sur une seule ligne (ou ce qu'il en reste...)

- en bloc :

```
/*
 * Un commentaire un peu plus long.
 * Les "*" intermédiaires ne sont pas obligatoires, mais Eclipse
 * les ajoute automatiquement pour le style. Laissez-les !
 */
```

À utiliser quand vous avez besoin d'écrire des explications un peu longues, mais que vous ne souhaitez pas voir apparaître dans la documentation à proprement parler (la JavaDoc).

Commentaires

Plusieurs sortes de commentaires (2)

- en bloc JavaDoc :

```
/**
 * Returns an expression equivalent to current expression, in which
 * every occurrence of unknown var was substituted by the expression
 * specified by parameter by.
 *
 * @param var   variable that should be substituted in this expression
 * @param by    expression by which the variable should be substituted
 * @return      the transformed expression
 */
Expression subst(UnknownExpr var, Expression by);
```

Commentaires

La JavaDoc

À propos de la JavaDoc :

- Les commentaires au format JavaDoc sont compilables en documentation au format HTML (dans Eclipse : menu “Project”, “Generate JavaDoc...”).
- Pour toute déclaration de type (classe, interface, enum...) ou de membre (attribut, constructeur, méthode), un squelette de documentation au bon format (avec les bonnes balises) peut être généré avec la combinaison `Alt`+`Shift`+`J` (toujours dans Eclipse).
- Il est **indispensable** de documenter tout ce qui est public.
- Il est **fortement recommandé** de documenter tout ce qui n’est pas privé (car utilisable par d’autres programmeurs, qui n’ont pas accès au code source).
- Il est **utile** de documenter ce qui est privé, pour soi-même et les autres membres de l’équipe.

Patrons de conception (1)

ou *design patterns*

- Analogie langage naturel : patron de conception = figure de style
- Ce sont des stratégies standardisées et éprouvées pour arriver à une fin.
  ex : créer des objets, décrire un comportement ou structurer un programme
- Les utiliser permet d'éviter les erreurs les plus courantes (pour peu qu'on utilise le bon patron !) et de rendre ses intentions plus claires pour les autres programmeurs qui connaissent les patrons employés.
- Connaître les noms des patrons permet d'en discuter avec d'autres programmeurs. [1]

1. De la même façon qu'apprendre les figures de style en cours de Français, permet de discuter avec d'autres personnes de la structure d'un texte...

Patrons de conception (2)

ou *design patterns*

- Quelques exemples dans le cours : décorateur, délégation, observateur/observable, monteur.
- Patrons les plus connus décrits dans le livre du “Gang of Four” (GoF) [1]
- Les patrons ne sont pas les mêmes d’un langage de programmation à l’autre :
  - les patrons implémentables dépendent de ce que la syntaxe permet
  - les patrons utiles dépendent aussi de ce que la syntaxe permet :
    quand un nouveau langage est créé, sa syntaxe permet de traiter simplement des situations qui autrefois nécessitaient l’usage d’un patron (moins simple).
    Plusieurs concepts aujourd’hui fondamentaux (comme les « classes », comme les énumérations, ...) ont pu apparaître comme cela.

1. E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns : Elements of Reusable Object-oriented Software*, 1995, Addison-Wesley Longman Publishing Co., Inc.

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)

Objet java = entité...

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

La variable pointeur `Personne toto` pointe vers ⟼ l'objet

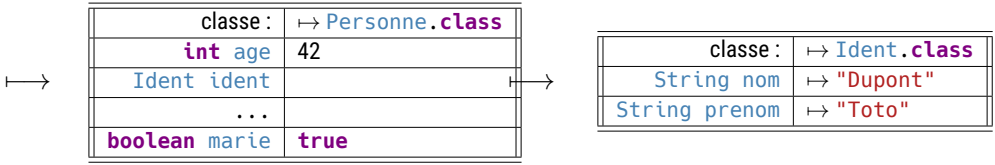
classe de l'objet :	réf. ↦ `Personne.class`
`int age`	42
`String nom`	réf. ↦ chaîne `"Dupont"`
`String prenom`	réf. ↦ chaîne `"Toto"`
...	
`boolean marie`	`true`

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. C'est implicite : à la différence de C, tout est fait en Java pour masquer le fait qu'on manipule des pointeurs. Par ailleurs, Java n'a pas d'arithmétique des pointeurs.

Objets

Quelle bonne vision bas niveau ?

À service égal [1], les objets-`Personne` pourraient aussi être représentés ainsi :



Les méthodes seraient écrites différemment mais, à l'usage, cela ne se verrait pas. [2]

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

1. Avec la même vision haut niveau.
2. 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~~ → **graphe d'objet** = un certain nombre d'enregistrements, se référençant les uns les autres, tels 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 par pointeur.
- Les informations stockées dans ce graphe permettent aux services (méthodes de l'enregistrement principal) prévus par le type (interface) de l'objet de fonctionner.

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.

Objets

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

Une piste : la distinction entre aggrégation et composition.

- **aggrégation** : un objet auxiliaire est utilisé (pointé par un attribut [1]) pour fournir certaines fonctionnalités de l'objet principal
- **composition** : aggrégation où, en plus, le cycle de vie de l'objet auxiliaire est lié à celui de l'objet principal (on peut parler de **sous-objet**)

Seulement dans la composition on peut considérer que l'objet auxiliaire appartient à l'objet principal.

En Java : pas de syntaxe pour distinguer entre les deux…

... mais on veut avoir cette distinction à l'esprit quand on conçoit une architecture objet.

1. Dans le cas de Java. Dans d'autres langages comme C++, un objet tiers peut être carrément inclus dans l'objet principal. Quand c'est le cas, il s'agit nécessairement de composition et non pas d'une aggrégation simple.

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).

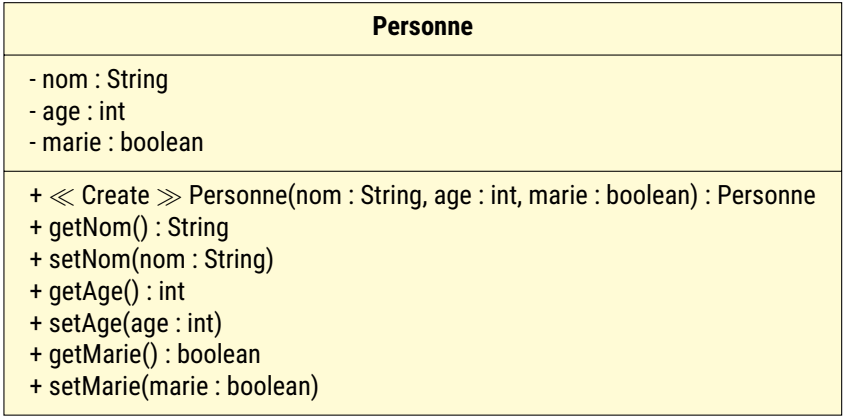
→ l'existence de classes n'est pas une nécessité en POO

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].

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 (pas considéré comme un membre !)
    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 ! (c'est un type membre)
    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`.

Un membre `m` d'une classe `C` peut être

- soit non statique ou **d'instance** = lié à (la durée de vie et au contexte d') une instance de `C`.
  Utilisable, en écrivant « `m` », partout où un `this` (**récepteur** implicite) de type `C` existe et, ailleurs, en écrivant « `expressionDeTypeC.m` ».
- soit **statique** = lié à (la durée de vie et au contexte d') une classe `C` [1].
  → 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.

Remarque : dans les langages objets purs, la notion de statique n'existe pas. Les membres d'une classe correspondent alors aux membres de ses instances.

1. ±permanent et « global ». NB : ça ne veut pas dire visible de partout : `static private` est possible !

Membres d'une classe

Statique ou pas ? → concrètement

	statique (ou "de classe")	non statique (ou "**d'instance**")
attribut	donnée globale[1], commune à toutes les instances de la classe.	donnée propre[2] 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. Correspond à variable globale dans d'autres langages.
2. 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 :

- **en termes de visibilité/encapsulation :** `f`, pour que `this` soit de type `C`, doit être déclarée dans `C`. Mais `g` pourrait être déclarée ailleurs sans changer le type de `that`.
- **en termes de comportement de l'appel :** 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 redéfintion de `f` 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

Encapsuler c'est empêcher le code extérieur d'accéder aux détails d'implémentation d'un composant.

- **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 à propos de `FiboGen` !

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 à propos de `FiboGen` !

Bien :

```
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, il est causé par l'exécution de `next`, pas de celle d'un code extérieur !

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 à propos de `FiboGen` !

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, il est causé par l'exécution de `next`, pas de celle d'un code extérieur ! [1]

1. Or un bug peut se manifester 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 !

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;
}
```

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)

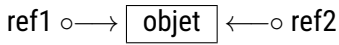
Comportements envisageables pour `get` et `set` :

- 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 → 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'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` ;
- les **`record`** (Java $\geq$ 14) ;
- plus généralement, toute classe [1] dont la documentation dit qu'elle l'est.

Les 8 types primitifs [2] se comportent aussi comme des types immuables [3].

1. Voire `sealed` **`interface`** (Java $\geq$ 15), sinon les types définis par les interfaces ne peuvent pas être garantis immuables !
2. **`boolean`**, **`char`**, **`byte`**, **`short`**, **`int`**, **`long`**, **`float`** et **`double`**
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.

Zones de la mémoire

Vue d'ensemble

La mémoire de la JVM s'organise en plusieurs zones :

- **zone des méthodes** : données des classes, dont méthodes (leurs codes-octet) et attributs statiques (leurs valeurs)
- **tas** : zone où sont stockés les objets alloués dynamiquement
- **pile(s)** (une par *thread* [1]) : là où sont stockées les données temporaires de chaque appel de méthode en cours
- **zone(s) des registres** (une par *thread*), contient notamment les registres suivants :
  - l'adresse de la prochaine instruction à exécuter (« *program counter* ») sur le *thread*
  - l'adresse du sommet de la pile du *thread*

1. Fil d'exécution parallèle. Cf. chapitre sur la programmation concurrente.

Zones de la mémoire

Le tas/*heap* et autres zones

Tas :

- Objets (tailles diverses) stockés dans le **tas**.
- Tas géré de façon automatique : quand on crée un objet, l'espace est réservé automatiquement et quand on ne l'utilise plus, la JVM le détecte et libère l'espace (**ramasse-miettes**/*garbage-collector*).
- L'intérieur de la zone réservée à un objet est constitué de champs, contenant chacun une valeur primitive ou bien une adresse d'objet.

Zones de la mémoire

La pile/*stack* (1)

Pile :

- chaque *thread* possède sa propre pile, consistant en une liste de ***frames***;
- 1 *frame* est empilé (au sommet de la pile) à chaque appel de méthode et dépilé (du sommet de la pile) à son retour (ordre LIFO);
- tous les *frames* d'une méthode donnée ont la même taille, calculée à la compilation;
- un *frame* contient en effet
  - les paramètres de la méthode (nombre fixe),
  - ses variables locales (nombre fixe)
  - et une pile bas niveau permettant de stocker les résultats des expressions (bornée par la profondeur syntaxique des expressions apparaissant dans la méthode). [1]

Chaque valeur n'occupe que 32 (ou 64) bits = valeur primitive ou adresse d'objet [2].

1. Remarquer l'analogie entre objet/classe (classe = code définissant la taille et l'organisation de l'objet) et *frame*/méthode (méthode = code définissant la taille et l'organisation du *frame*).
2. En réalité, la JVM peut optimiser en mettant les objets locaux en pile. Mais ceci est invisible.

Zones de la mémoire

La pile/*stack* (2)

- Lors de l'appel d'une méthode [1] :
  - Un *frame* est instancié et mis en pile.
  - On y stocke immédiatement le pointeur de retour (vers l'instruction appelante), et les valeurs des paramètres effectifs.
- Lors de son exécution, les opérations courantes prennent/retirent leurs opérandes du sommet de la pile bas niveau et écrivent leurs résultats au sommet de cette même pile (ordre LIFO) ;
- Au retour de la méthode [2], le *program counter* du *thread* prend la valeur du pointeur de retour ; le cas échéant [3], la valeur de retour de la méthode est empilée dans la pile bas niveau du *frame* de l'appelant.
  Le *frame* est désalloué.

1. Dans le code-octet : **invokedynamic**, **invokeinterface**, **invokespecial**, **invokestatic** ou **invokevirtual**.
2. Dans le code-octet : **areturn**, **dreturn**, **freturn**, **ireturn**, **lreturn** ou **return**.
3. C.-à-d. sauf méthode **`void`**, (tout sauf **return** simple dans le code octet).

Types de données

- **type de données** = ensemble d'éléments représentant des données de forme similaire, traitables de la même façon par un même programme.
- Chaque langage de programmation a sa propre idée de ce à quoi il faut donner un type, de quand il faut le faire, de quels types il faut distinguer et comment, etc.
  On parle de différents **systèmes de types**.

Système de types de Java

Caractéristiques principales

- typage qualifié de “fort” (concept plutôt flou : on peut trouver bien plus strict !)
- **typage statique** : le compilateur vérifie le type des expressions du code source
- **typage dynamique** : à l’exécution, les objets connaissent leur type. Il est testable à l’exécution (permet traitement différencié [1] dans code polymorphe).
- sous-typage, permettant le polymorphisme : une méthode déclarée pour argument de type `T` est appelable sur argument pris dans tout sous-type de `T`.
- 2 “sortes” de type : types primitifs (valeurs directes) et types référence (objets)
- **typage nominatif** [2] : 2 types sont égaux ssi ils ont le même nom. En particulier, si `class A { int x; }` et `class B { int x; }` alors `A x = new B();` ne passe pas la compilation bien que `A` et `B` aient la même structure.

1. Via la liaison tardive/dynamique et via mécanismes explicites : `instanceof` et réflexion.
2. Contraire : typage structurel (ce qui compte est la structure interne du type, pas le nom donné)

Types de données en Java

Deux catégories

Pour des raisons liées à la mémoire et au polymorphisme, 2 catégories [1] de types :

	types primitifs	**types référence**
données représentées	données simples	données complexes (objets)
valeur [2] d'une expression	donnée directement	adresse d'un objet ou `null`
espace occupé	32 ou 64 bits	32 bits (adresse)
nombre de types	8 (fixé, cf. page suivante)	nombreux fournis dans le JDK et on peut en programmer
casse du nom	minuscules	majuscule initiale (par convention)

Il existe quelques autres différences, abordées dans ce cours.

1. Les distinctions primitif/objet et valeur directe/référence coïncident en Java, mais c'est juste en Java.
Ex : C++ possède à la fois des objets valeur directe et des objets accessibles par pointeur !
En Java ($\leq$ 15), on peut donc remplacer "type référence" par "type objet" et "type primitif" par "type à valeur directe" sans changer le sens d'une phrase... mais il est question que ça change (projet Valhalla) !
2. Les 32 bits stockés dans un champs d'objet ou empilés comme résultat d'un calcul.

Types de données en Java

Zoom sur les types primitifs

Les 8 types primitifs :

Nom	description	t. contenu	t. utilisée	exemples
byte	entier très court	8 bits	1 mot [1]	127,-19
short	entier court	16 bits	1 mot	-32_768 , 15_903
int	entier normal	32 bits	1 mot	23_411_431
long	entier long	64 bits	2 mots	3_411_431_434L
float	réel à virgule flottante	32 bits	1 mot	3_214.991f
double	idem, double précision	64 bits	2 mots	-223.12 , 4.324E12
char	caractère unicode	16 bits	1 mot	'a' '@' '\0'
boolean	valeur de vérité	1 bit	1 mot	**true**, **false**

Cette liste est exhaustive : le programmeur ne peut pas définir de types primitifs.

Tout type primitif a un nom **en minuscules**, qui est un mot-clé réservé de Java (~~String **int** = "truc"~~ ne compile pas, alors que **int** String = 12, oui !).

1. 1 **mot** = 32 bits

Références et valeurs directes

Interprétations

Valeurs calculées stockées, en pile ou dans un champ, sur 1 mot (2 si `long` ou `double`)

- types primitifs : directement la valeur intéressante
- types références : une adresse mémoire (pointant vers un objet dans le tas).

Dans les 2 cas : ce qui est stocké dans un champ ou dans la pile n'est qu'une suite de 32 bits, indistinguables de ce qui est stocké dans un champ d'un autre type.

L'interprétation faite de cette valeur dépendra uniquement de l'instruction qui l'utilisera, mais la compilation garantit que ce sera la bonne interprétation.

Cas des types référence : quel que soit le type, cette valeur est interprétée de la même façon, comme une adresse. Le type décrit alors l'objet référencé seulement.

Exemple : une variable de type `String` et une de type `Point2D` contiennent tous deux le même genre de données : un mot représentant une adresse mémoire. Pourtant la première pointera toujours sur une chaîne de caractères alors que la seconde pointera toujours sur la représentation d'un point du plan.

Références et valeurs directes

Conséquences sur l'affectation et l'égalité

La distinction référence/valeur directe a plusieurs conséquences à l'exécution.

Pour `x` et `y` variables de types références :

- Après l'affectation `x = y`, les deux variables désignent le même emplacement mémoire (***aliasing***).
  Si ensuite on exécute l'affectation `x.a = 12`, alors après `y.a` vaudra aussi `12`.
- Si les variables `x` et `z` désignent des emplacements différents, le test d'identité [1] `x == z` vaut **`false`**, **même si** le contenu des deux objets référencés est identique.

1. Pour les primitifs, **identité** et **égalité sémantique** sont la même chose. Pour les objets, le test d'égalité sémantique est la méthode `public boolean equals(Object other)`. Cela veut dire qu'il appartient au programmeur de définir ce que veut dire « être égal », pour les instances du type qu'il invente.

Références et valeurs directes

Conséquence sur le passage de paramètres à l'appel d'une méthode

Rappel : En Java, quand on appelle une méthode, on **passe les paramètres par valeur** uniquement : une copie de la valeur du paramètre est empilée avant appel.

Ainsi :

- pour les types primitifs [1] → la méthode travaille sur une copie des données réelles
- pour les types référence → c'est l'adresse qui est copiée ; la méthode travaille avec cette copie, qui pointe sur... les mêmes données que l'adresse originale.

Conséquence :

- Dans tous les cas, affecter une nouvelle valeur à la variable-paramètre ne sert à rien : la modification serait perdue au retour.
- Mais si le paramètre est une référence, on peut modifier l'objet référencé. Cette modification persiste après le retour de méthode.

1. = types à valeur directe, pas les types référence

Références et valeurs directes

Conséquence sur le passage de paramètres à l'appel d'une méthode (remarques)

- Ainsi, si le paramètre est un objet non modifiable, on retrouve le comportement des valeurs primitives. [1]
- On entend souvent *"En Java, les objets sont passés par référence"*.
  Ce n'est pas rigoureux !
  Le passage par référence désigne généralement autre chose, de très spécifique [2] (notamment en C++, PHP, Visual Basic .NET, C#, REALbasic...).

---

1. Les types primitifs et les types immuables se comportent de la même façon pour de nombreux critères.
2. **Passage par référence** : quand on passe une variable v (plus généralement, une *lvalue*) en paramètre, le paramètre formel (à l'intérieur de la méthode) est un *alias* de v (un pointeur "déguisé" vers l'adresse de v, mais utilisable comme si c'était v).
Toute modification de l'*alias* modifie la valeur de v.
En outre, le pointeur sous-jacent peut pointer vers la pile (si v variable locale), ce qui n'est jamais le cas des "références" de Java.

Typages dynamique et statique

Quand vérifier la cohérence des données ?

- La vérification du bon typage d'un programme peut avoir lieu à différents moments :
  - langages très « bas niveau » (assembleur x86, p. ex.) : jamais ;
  - C, C++, OCaml, ... : dès la compilation (**typage statique**) ;
  - Python, PHP, Javascript, ... : seulement à l'exécution (**typage dynamique**) ;

  Remarque : typages statique et dynamique ne sont pas mutuellement exclusifs. [1]
- Les entités auxquelles ont attribue un type ne sont pas les mêmes selon le moment où cette vérification est faite.

  Typage statique $\rightarrow$ concerne les expressions du programme

  Typage dynamique $\rightarrow$ concerne les données existant à l'exécution.

Où se Java se situe-t-il ? Que type-t-on en Java ?

1. Il existe même des langages où le programmeur décide ce qui est vérifié à l'exécution ou à la compilation : « typage graduel ».

Stades de vérification et entités typables en Java

Java → langage à typage statique, mais avec certaines vérifications à l'exécution [1] :

- À la compilation on vérifie le type des **expressions** [2] (**analyse statique**).
  Toutes les expressions sont vérifiées.
- À l'exécution, la JVM peut vérifier le type des **objets** [3].
  Cette vérification a seulement lieu lors d'évènements bien précis :
  - quand l'on souhaite différencier le comportement en fonction de l'appartenance ou non à un type (lors d'un test `instanceof` [4] ou d'un appel de méthode d'instance [5]).
  - quand on souhaite interrompre le programme sur une exception en cas d'incohérence de typage [6] : notamment lors d'un *downcasting*, ou bien après exécution d'une méthode générique dont le type de retour est une variable de type.

1. C'est en fait une caractéristique habituelle des langages à typage essentiellement statique mais autorisant le polymorphisme par sous-typage.
2. Expression = élément syntaxique du programme représentant une valeur calculable.
3. Ces entités n'existent pas avant l'exécution, de toute façon !
4. Code-octet : **instanceof**.
5. Code-octet : **invokeinterface** ou **invokevirtual**.
6. Code-octet : **checkcast**.

Stades de vérification et entités typables en Java

À la compilation : les expressions

Type statique déterminé via les annotations de type explicites et par déduction. [1]

- Le compilateur sait que l'expression `"bonjour"` est de type `String`. (idem pour les types primitifs : `42` est toujours de type `int`).
- Si on déclare `Scanner s`, alors l'expression `s` est de type `Scanner`.
- Le compilateur sait aussi déterminer que `1.0 + 2` est de type `double`.
- (Java $\geq$ 10) Après `var m = "coucou";`, l'expression `m` est de type `String`.

Le compilateur vérifie la compatibilité du type de chaque expression avec son contexte :

- `int x = 1; System.out.println(x/2);` est bien typé.
- en revanche, ~~`Math.cos("bonjour")`~~ est mal typé.

1. Java ne dispose pas d'un système d'inférence de type évolué comme celui d'OCaml, néanmoins cela n'empêche pas de nombreuses déductions directes comme dans les exemples donnés ici.

Stades de vérification et entités typables en Java

À l'exécution : les objets

À l'instanciation d'un objet, le nom de sa classe y est inscrit, définitivement. Ceci permet :

- d'exécuter des tests demandés par le programmeur (comme `instanceof`) ;
- à la méthode `getClass()` de retourner un résultat ;
- de faire fonctionner la liaison dynamique (dans `x.f()`, la JVM regarde le type de l'objet référencé par `x` avant de savoir quel `f()` exécuter) ;
- de vérifier la cohérence de certaines conversions de type :
  `Object o; ... ; String s = (String)o;`
- de s'assurer qu'une méthode générique retourne bien le type attendu :
  `ListString s = listeInscrits.get(idx);`

Ceci ne concerne pas les valeurs primitives/directes : pas de place pour coder le type dans les 32 bits de la valeur directe ! (et, comme on va voir, ça n'aurait pas de sens, vu le traitement du polymorphisme et des conversions de type des primitifs.)

Relation entre type statique et type dynamique

Pour une variable ou expression :

- son **type statique** est son type tel que déduit par le compilateur (pour une variable : c'est le type indiqué dans sa déclaration) ;
- son **type dynamique** est la classe de l'objet référencé (par cette variable ou par le résultat de l'évaluation de cette expression).
- Le type dynamique ne peut pas être déduit à la compilation.
- Le type dynamique change [1] [2] au cours de l'exécution.

La vérification statique et les règles d'exécution garantissent la propriété suivante :

Le type dynamique d'une variable ou expression est toujours un sous-type (cf. juste après) **de son type statique.**

1. Pour une variable : après chaque affectation, un objet différent peut être référencé.
Pour une expression : une expression peut être évaluée plusieurs fois lors d'une exécution du programme et donc référencer, tour à tour, des objets différents.
2. Remarque : le type (la classe) d'un objet donné est, en revanche, fixé(e) dès son instanciation.