Compléments en Programmation Orientée Objet

Aldric Degorre

Version `2022.03.01` du 9 octobre 2022

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

- **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 ?

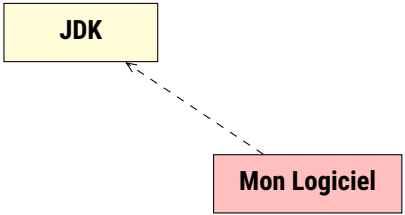
- Supprimer ce fameux bug que vous n'aviez pas eu le temps de corriger avant la deadline de PI4 l'année dernière ?
- Remplacer une des dépendances de votre projet par une nouvelle bibliothèque, sans tout modifier et ajouter 42 nouveaux bugs ?
- Ajouter une extension que vous n'aviez pas non plus eu le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

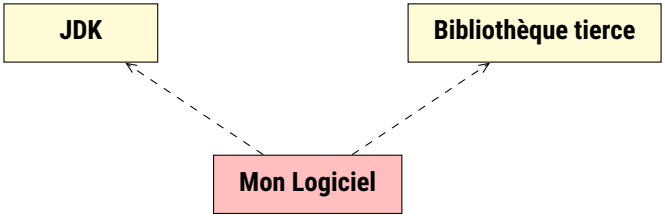
La vraie vie ?



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 ?

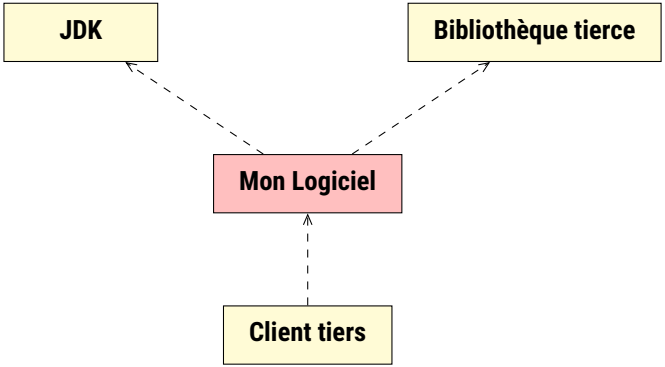
La vraie vie ?



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 !

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.

- **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...

Java

Quel Java dans ce cours ?

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 ($\rightarrow$ 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 (→ 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 $\rightarrow$ 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

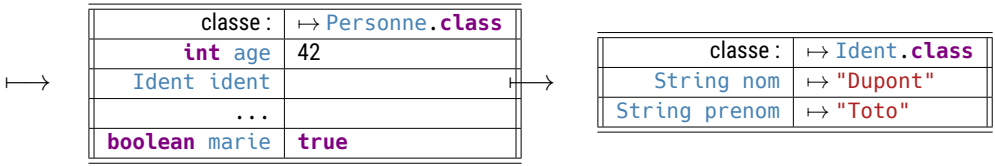
classe de l'objet :	réf. ↦ `Personne.class`
`int age`	42
`String nom`	réf. ↦ chaîne `"Dupont"`
`String prenom`	réf. ↦ chaîne `"Toto"`
...	
`boolean marie`	`true`

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 références 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).

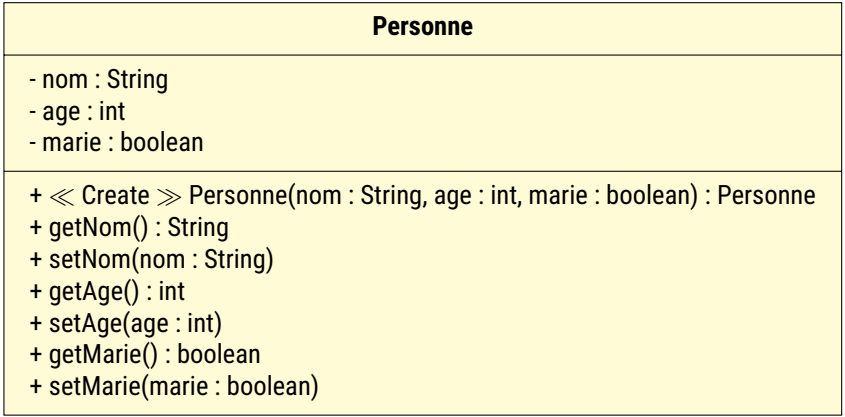
→ l'existence de classes n'est pas une nécessité en POO

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

```
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 ($\simeq$ `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`.

Membres d'une classe

Statique ou pas ?

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
```

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 !

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 : (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)

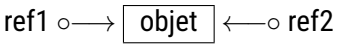
Comportements envisageables pour `get` et `set` :

- 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.

Types imbriqués

Qu'est-ce que c'est ?

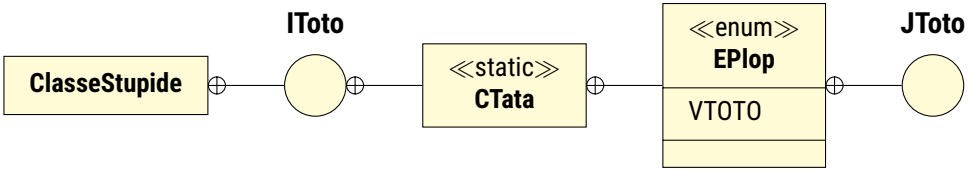
Java permet de définir un **type (classe ou interface) imbriqué** [1] à l'intérieur d'une autre définition de type (dit **englobant** [2]) :

```
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]) :

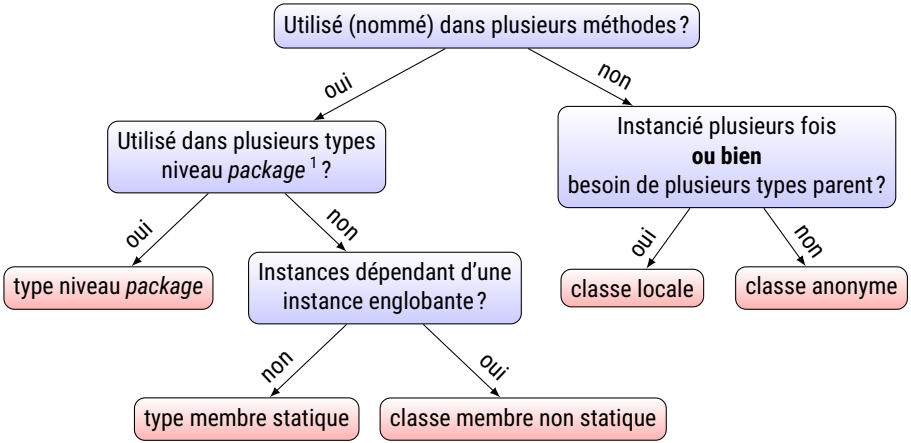
- dans tous les cas : les membres statiques du type englobant ;
- pour les classes membres non statiques et classes locales dans bloc non statique : tous les membres non statiques du type englobant ;
- pour les classes locales : les définitions locales [2].

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

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.

Notion de sous-typage

Définition abstraite

- **Définition :** le type A est **sous-type** de B ($A <: B$) (ou bien B **supertype** de A ($B :> A$)) si toute entité [1] de type A
  - est aussi de type B
  - (autrement dit :) « peut remplacer » une entité de type B.

→ plusieurs interprétations possibles (mais contraintes par notre définition de « type »).

1. Pour Java, entité = soit expression, soit objet.

Notion de sous-typage

Interprétations

- **Interprétation faible :** ensembliste. Tout sous-ensemble d'un type donné forme un sous-type de celui-ci.

  Exemple : tout carré est un rectangle, donc le type carré est sous-type de rectangle.

→ insuffisant car un type n'est pas un simple ensemble [1] : il est aussi muni d'opérations, d'une structure, de contrats [2], ...

Contrat : propriété que les implémentations d'un type s'engagent à respecter. Un type honore un tel contrat si et seulement si **toutes ses instances** ont cette propriété.

1. Pour les algébristes, on peut faire l'analogie avec les groupes, par exemple : un sous-ensemble d'un groupe n'est pas forcément un groupe (il faut aussi qu'il soit stable par les opérations de groupe, afin que la structure soit préservée).
2. Formels (langage de spécification formel) ou informels (documentation utilisateur, comme javadoc).

Notion de sous-typage

Interprétations

- **Interprétation « minimale » :** sous-typage structurel. *A* est sous-type de *B* si toute instance de *A* sait traiter les messages qu'une instance de *B* sait traiter.

  Concrètement : *A* possède toutes les méthodes de *B*, avec des signatures au moins aussi permissives en entrée et au moins aussi restrictives en sortie. [1]

→ sous-typage plus fort et utilisable car vérifiable en pratique, mais insuffisant pour prouver des propriétés sur un programme polymorphe (toujours pas de contrats).

1. Contravariance des paramètres et covariance du type de retour.

Notion de sous-typage

Interprétations

Pourquoi le sous-typage structurel est insuffisant ?

Exemple :

- *Dans le cours précédent, les instances de la classe* `FiboGen` *génèrent la suite de Fibonacci.*
- *Contrat possible*[1] *: « le rapport de 2 valeurs successives tend vers* $\varphi = \frac{1+\sqrt{5}}{2}$ *(nombre d'or) ».*
  (On sait prouver ce contrat pour la méthode `next` *des instances directes de* `FiboGen`*.)*
- *Or rien empêche de créer un sous-type* `BadFib` *(sous-classe*[2]*) de* `Fibogen` *dont la méthode* `next` *retournerait toujours* 0.
  $\rightarrow$ *Les instances de* `BadFib` *seraient alors des instances de* `FiboGen` *violant le contrat.*

1. Raisonnable, dans le sens où c'est une propriété mathématique démontrée pour la suite de Fibonacci, qui donc doit être vraie dans toute implémentation correcte.
2. Une sous-classe est bien un sous-type au sens structurel : les méthodes sont héritées.

Notion de sous-typage

Interprétations

- **Interprétation idéale : Principe de Substitution de Liskov**[1] (LSP). Un sous-type doit respecter tous les contrats du supertype.
  Les propriétés du programme prouvables comme conséquence des contrats du supertype sont alors effectivement vraies quand on utilise le sous-type à sa place.

  Exemple : les propriétés largeur et hauteur d'un rectangle sont modifiables indépendamment. Un carré ne satisfait pas ce contrat. Donc, selon le LSP, le type carré modifiable n'est pas sous-type de rectangle modifiable.

  En revanche, carré non modifiable est sous-type de rectangle non modifiable, selon le LSP.

1. C'est le « L » de la méthodologie SOLID (*Design Principles and Design Patterns*. Robert C. Martin.).

Notion de sous-typage

Interprétations

→ Hélas, le LSP est une notion trop forte pour les compilateurs : pour des contrats non triviaux, aucun programme ne sait vérifier une telle substituabilité (indécidable).

Cette notion n'est pas implémentée par les compilateurs, mais c'est bien celle que le programmeur doit avoir en tête pour écrire des programmes corrects !

→ **Interprétation en pratique :** tout langage de programmation possède un système de règles simples et vérifiables par son compilateur, définissant « **son** » sous-typage.

Notion de sous-typage

Interprétations : en pratique

Les grandes lignes du sous-typage selon Java : (détails dans JLS 4.10 et ci-après)

- Pour les 8 types primitifs, il y a une relation de sous-typage pré-définie.
- Pour les types référence, le sous-typage est nominal : *A* n'est sous-type de *B* que si *A* est déclaré comme tel (`implements` ou `extends`).
  Mais la définition de *A* ne passe la compilation que si certaintes contraintes structurelles [1] sont vérifiées, concernant les redéfinitions de méthodes.
- Types primitifs et types référence forment deux systèmes déconnectés. Aucun type référence n'est sous-type ou supertype d'un type primitif.

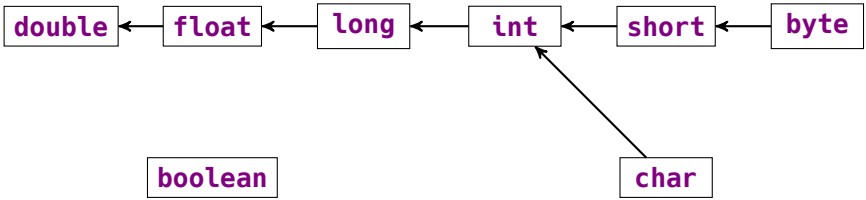
1. Cf. cours sur les interfaces et sur l'héritage pour voir quelles sont les contraintes exactes.

Notion de sous-typage pour Java

Relation de sous-typage en Java (1)

Types primitifs : (fermeture transitive et réflexive de la relation décrite ci-dessous)

- Un type primitif numérique est sous-type de tout type primitif numérique plus précis : `byte` $<:$ `short` $<:$ `int` $<:$ `long` et `float` $<:$ `double`.
- Par ailleurs `long` $<:$ [1] `float` et `char` $<:$ [2] `int`.
- `boolean` est indépendant de tous les autres types primitifs.



1. `float` (1 mot) n'est pas plus précis ou plus « large » que `long` (2 mots), mais il existe néanmoins une conversion automatique du second vers le premier.
2. Via la valeur unicode du caractère.

Notion de sous-typage pour Java

Relation de sous-typage en Java [4] (2)

Types référence :

$A <: B$ ssi B est `Object` [1] ou s'il existe des types $A_0(=A), A_1, \ldots, A_n(=B)$ tels que pour tout $i \in 1..n$, une des règles suivantes s'applique : [2]

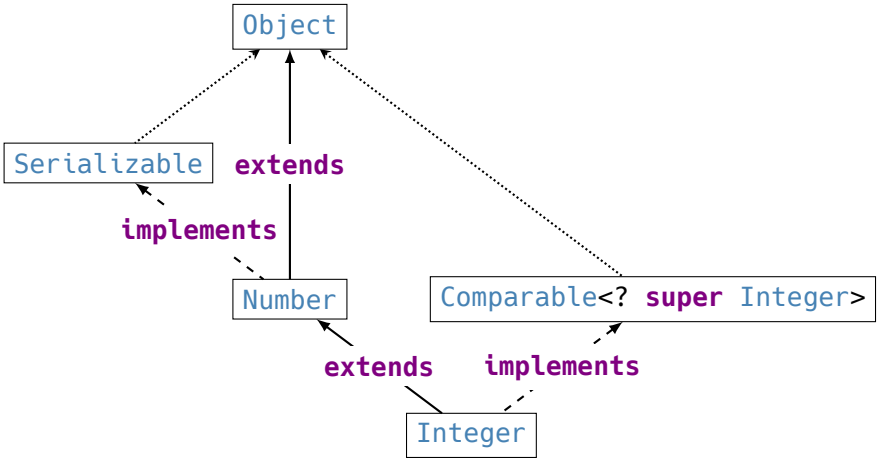
- (**implémentation d'interface**) A_{i-1} est une classe, A_i est une interface et A_{i-1} implémente A_i ;
- (**héritage de classe**) A_{i-1} et A_i sont des classes et A_{i-1} étend A_i ;
- (**héritage d'interface**) A_{i-1} et A_i sont des interfaces et A_{i-1} étend A_i ;
- (**covariance des types tableau** [3]) A_{i-1} et A_i resp. de forme a`[]` et b`[]`, avec $a <: b$;

1. C'est vrai même si A est une interface, alors même qu'aucune interface n'hérite de `Object`.
2. Pour être exhaustif, il manque les règles de sous-typage pour les types génériques.
3. Les types tableau sont des classes (très) particulières, implémentant les interfaces `Cloneable` et `Serializable`. Donc tout type tableau est aussi sous-type de `Object`, `Cloneable` et `Serializable`.
4. Cf. JLS 4.10.

Notion de sous-typage pour Java

Relation pour les types références, diagramme

Une partie du graphe de sous-typage : le type `Integer` et ses supertypes.



Principe fondamental

Dans un programme qui compile, remplacer une expression de type A par une de type B [1], avec $B <: A$, donne un programme qui compile encore

(à moins que sa compilation échoue pour cause de surcharge ambiguë [2]).

Remarque : seule la compilation est garantie, ainsi que le fait que le résultat de la compilation est exécutable.

La correction du programme résultant n'est pas garantie !

(pour cela, il faudrait au moins que java impose le respect du LSP, ce qui est impossible)

1. Syntaxiquement correcte ; avec identifiants tous définis dans leur contexte ; et bien typée.
2. En effet, le type statique des arguments d'une méthode surchargée influe sur la résolution de la surcharge et peut créer des ambiguïtés. Cf. chapitre sur le sujet.

Notion de sous-typage pour Java

Principe fondamental – explications

Pourquoi ce remplacement ne gène pas l'exécution :

- les objets sont utilisables sans modification comme instances de tous leurs supertypes [1] (**sous-typage inclusif**). P. ex. : `Object o = "toto"` fonctionne.
- Java [2] s'autorise, si nécessaire, à remplacer une valeur primitive par la valeur la plus proche dans le type cible (**sous-typage coercitif**). P. ex. : après l'affectation `float f = 1_000_000_000_123L;`, la variable `f` vaut `1.0E12` (on a perdu les derniers chiffres).

1. Les contraintes d'implémentation d'interface et d'héritage garantissent que les méthodes des supertypes peuvent être appelées.
2. Si nécessaire, javac convertit les constantes et insère des instructions dans le code-octet pour convertir les valeurs variables à l'exécution.

Notion de sous-typage pour Java

Corollaires :

- on peut affecter à toute variable une expression de son sous-type (ex : `double z = 12;`);
- on peut appeler toute méthode avec des arguments d'un sous-type des types déclarés dans sa signature (ex : `Math.pow(3, 'z')`);
- on peut appeler toute méthode d'une classe `T` donné sur un récepteur instance d'une sous-classe de `T` (ex : `"toto".hashCode()`).

Ces caractéristiques font du sous-typage la base du système de polymorphisme de Java.

Transtypage (*type casting*)

Transtypage = *type casting* = conversion de type d'une expression.

Plusieurs mécanismes [1] [2] :

- ***upcasting*** : d'un type vers un supertype (ex : `Double` vers `Number`)
- ***downcasting*** : d'un type vers un sous-type (ex : `Object` vers `String`)
- ***boxing*** : d'un type primitif vers sa version "emballée" (ex : `int` vers `Integer`)
- ***unboxing*** : d'un type emballé vers le type primitif correspondant (ex : `Boolean` vers `boolean`)
- conversion en `String` : de tout type vers `String` (implicite pour concaténation)
- parfois combinaison implicite de plusieurs mécanismes.

1. Détaillés dans la JLS, chapitre 5.
2. On ne mentionne pas les mécanismes explicites et évidents tels que l'utilisation de méthodes penant du `A` et retournant du `B`. Si on va par là, tout type est convertible en tout autre type.

Transtypage (*type casting*)

Tous ces mécanismes sont des règles permettant vérifier, à la compilation, si une expression peut être placée là où elle l'est.

Parfois, conséquences à l'exécution :

- vraie modification des données (types primitifs),
- ou juste vérification de la classe d'un objet (*downcasting* de référence).

Transtypage (*type casting*)

Autres termes employés (notamment dans la JLS)

- Élargissement/*widening* et rétrécissement/*narrowing* : dans la JLS (5.1), synonymes respectifs de *upcasting* et *downcasting*.

  Inconvénient : le sens étymologique (= réécriture sur resp. + de bits ou - de bits), ne représente pas la réalité en Java (cf. la suite).

- Promotion : synonyme de *upcasting*. Utilisé dans la JLS (5.6) seulement pour les conversions implicites des paramètres des opérateurs arithmétiques. [1]

1. Alors qu'on pourrait expliquer ce mécanisme de la même façon que la résolution de la surcharge.

Transtypage (*type casting*)

Autres termes employés (notamment dans la JLS)

- Coercition : conversion implicite de données d'un type vers un autre.
  Cf. **sous-typage coercitif** : mode de sous-typage où un type est sous-type d'un autre s'il existe une fonction [1] de conversion et que le compilateur insère implicitement des instructions dans le code octet pour que cette fonction soit appliquée.
  Inconvénient : incohérences entre définitions de coercition et sous-typage coercitif ;
  - la coercition ne suppose pas l'application d'une fonction ;
  - Java utilise des coercitions sans rapport avec le sous-typage (*auto-boxing*, *auto-unboxing*, conversion en chaîne, …).

→ **On ne prononcera plus élargissement, rétrécissement, promotion ou coercition !**

1. Au sens mathématique du terme. Pas forcément une méthode.

Transtypage, cas 1 : *upcasting*

- Cas d'application : on souhaite obtenir une expression d'un supertype à partir d'une expression d'un sous-type.
- L'*upcasting* est en général implicite (pas de marque syntaxique).
  Exemple :

```
double z = 3; // upcasting (implicite) de int vers double
```

- Utilité, polymorphisme par sous-typage : partout où une expression de type `T` est autorisée, toute expression de type `T'` est aussi autorisée si `T'` $<:$ `T`.
  Exemple : si `class B extends A {}`, `void f(A a)` et `B b`, alors l'appel `f(b)` est accepté.
- L'*upcasting* implicite permet de faire du polymorphisme de façon transparente.
- On peut aussi demander explicitement l'*upcasting*, ex : `(double)4`
- L'*upcasting* explicite sert rarement, mais permet parfois de guider la résolution de la surcharge : remarquez la différence entre `3/4` et `((double)3)/ 4`.

Transtypage, cas 2 : *downcasting*

***Downcasting* :**

- Cas d'application : on veut écrire du code spécifique pour un sous-type de celui qui nous est fourni.
- Dans ce cas, il faut demander une conversion explicite.
  Exemple : `int x = (int)143.32`.
- Utilité :
  - (pour les objets) dans un code polymorphe, généraliste, on peut vouloir écrire une partie qui travaille seulement sur un certain sous-type, dans ce cas, on teste la classe de l'objet manipulé et on *downcast* l'expression qui le référence :

```
if (x instanceof String) { String xs = (String) x; ... ;}
```

  - Pour les nombres primitifs, on peut souhaiter travailler sur des valeurs moins précises : `int partieEntiere = (int)unReel;`

Transtypage, cas 2 : *downcasting*

Avertissement

Le code ci-dessous est probablement symptôme d'une conception non orientée objet :

```
// Anti-patron :
void g(Object x) { // Object ou bien autre supertype commun à  C1 et C2
    if (x instanceof C1) { C1 y = (C1) x; f1(y); }
    else if (x instanceof C2) { C2 y = (C2) x; f2(y); }
    else { /* quoi en fait ? on génère une erreur ? */}
}
```

Quand c'est possible, on préfère utiliser la liaison dynamique :

```
public interface I { void f(); }
void g(I x) { x.f(); } // déjà , programmons à  l'interface
```

```
// puis dans d'autres fichiers (voire autres packages)
public class C1 implements I { public void f() { f1(this); }}
public class C2 implements I { public void f() { f2(this); }}
```

Avantage : les types concrets manipulés ne sont jamais nommés dans `g`, qui pourra donc fonctionner avec de nouvelles implémentations de `I` sans modification.

Transtypage, cas 3 : *auto-boxing/unboxing*

Types “emballés”

- Pour tout type primitif, il existe un type référence “**emballé**” ou “mis en boîte” (*wrapper type* ou *boxed type*) équivalent : `int`↔`Integer`, `double`↔`Double`, ...
- **Attention, contrairement à leurs équivalents primitifs, les différents types emballés ne sont pas sous-types les uns des autres !** Ils ne peuvent donc pas être transtypés de l’un dans l’autre.

  ~~`Double d = new Integer(5);`~~ →
  `Double d = new Integer(5).doubleValue();` ou encore
  `Double d = 0. + (new Integer(5));`[1]

1. Spoiler : cet exemple utilise des conversions automatiques. Voyez-vous lesquelles ?

Transtypage, cas 3 : *auto-boxing/unboxing*

Conversions automatiques, depuis Java 5

- Partout où un type emballé est attendu, une expression de type valeur correspondant sera acceptée : ***auto-boxing***.
- Partout où un type valeur est attendu, sa version emballée sera acceptée : ***auto-unboxing***.
- Cette conversion automatique permet, dans de nombreux usages, de confondre un type primitif et le type emballé correspondant.

Exemple :

- `Integer x = 5;` est équivalent de `Integer x = Integer.valueOf(5);` [1]
- Réciproquement `int x = new Integer(12);` est équivalent de `int x = (new Integer(12)).intValue();`

1. La différence entre `Integer.valueOf(5)` et `new Integer(5)` c'est que la fabrique statique `valueOf()` réutilise les instances déjà créées, pour les petites valeurs (mise en cache).

Transtypage : à propos du “*cast*”

i.e., la notation `(Type)expr`

- Particulièrement utile pour le *downcasting*, mais sert à toute conversion.
- Soient `A` et `B` des types et `e` une expression de type `A`, alors l’expression “`(B)e`”
  - est de type statique `B`
  - et a pour valeur (à l’exécution), autant que possible, la “même” que `e`.
- L’expression “`(B)e`” passe la compilation à condition que (au choix) :
  - `A` et `B` soient des types référence avec `A <: B` ou `B <: A`;
  - que `A` et `B` soient des types primitifs tous deux différents de **`boolean`**[1];
  - que `A` soit un type primitif et `B` la version emballée de `A`;
  - ou que `B` soit un type primitif et `A` la version emballée d’un sous-type de `B` (combinaison implicite d’*unboxing* et *upcasting*).
- Même quand le programme compile, effets indésirables possibles à l’exécution :
  - perte d’information quand on convertit une valeur primitive;
  - `ClassCastException` quand on tente d’utiliser un objet en tant qu’objet d’un type qu’il n’a pas.

1. NB : `(char)((byte)0)` est légal, alors qu’il n’y a pas de sous-typage dans un sens ou l’autre.

Subtilités du transtypage

Cas des types primitifs : possible perte d'information (2)

- Cas avec perte d'information possible :
  - tous les *downcastings* primitifs ;
  - *upcastings* de `int` vers `float` [1], `long` vers `float` ou `long` vers `double` ;
  - *upcasting* de `float` vers `double` hors contexte `strictfp` [2].
- Cas sans perte d'information : (= les autres cas = les "vrais" *upcastings*)
  - *upcasting* d'entier vers entier plus long ;
  - *upcasting* d'entier $\leq$ 24 bits vers `float` et `double` ;
  - *upcasting* d'entier $\leq$ 53 bits vers `double` ;
  - *upcasting* de `float` vers `double` sous contexte `strictfp`.

1. Par exemple, `int` utilise 32 bits, alors que la **mantisse** de `float` n'en a que 24 (+ 8 bits pour la position de la virgule) $\rightarrow$ certains `int` ne sont pas représentables en `float`.
2. Selon implémentation, mais pas de garantie. Cherchez à quoi sert ce mot-clé !

Subtilités du transtypage

Cas des types primitfs : points d'implémentation

- Pour *upcasting* d'entier de $\leq$ 32 bits vers $\leq$ 32 bits : dans code-octet et JVM, granularité de 32 bits $\rightarrow$ tous les "petits" entiers codés de la même façon $\rightarrow$ aucune modification nécessaire. [1]
- Pour une conversion de littéral [2], le compilateur fait lui-même la conversion et remplace la valeur originale par le résultat dans le code-octet.
- Dans les autres cas, conversions à l'exécution dénotées, dans le code-octet, par des instructions dédiées :
  - *downcasting* : **`d2i`**, **`d2l`**, **`d2f`**, **`f2i`**, **`f2l`**, **`i2b`**, **`i2c`**, **`i2s`**, **`l2i`**
  - *upcasting* avec perte : **`f2d`** (sans **`strictfp`**), **`i2f`**, **`l2d`**, **`l2f`**
  - *upcasting* sans perte : **`f2d`** (avec **`strictfp`**), **`i2l`**, **`i2d`**

1. C'est du sous-typage inclusif, comme pour les types référence !
2. Littéral numérique = nombre écrit en chiffres dans le code source.

Subtilités du transtypage

Cas des types primitfs : points d'implémentation

Et concrètement, que font les conversions ?

- *Upcasting* d'entier $\leq$ 32 bits vers **`long`** (**i2l**) : on complète la valeur en recopiant le bit de gauche 32 fois. [1]
- *Downcasting* d'entier vers entier *n* bits (**i2b**, **i2c**, **i2s**, **l2i**) : on garde les *n* bits de droite et on remplit à gauche en recopiant $32 - n$ fois le bit le plus à gauche restant. [2]

1. Pour plus d'explications : chercher "représentation des entiers en complément à 2".
2. Ainsi, la valeur d'origine est interprétée modulo 2^n sur un intervalle centré en 0.

Subtilités du transtypage

Cas des types primitfs : exemples

- `int i = 42; short s = i;` : pour copier un **`int`** dans un **`short`**, on doit le rétrécir. La valeur à convertir est inconnue à la compilation → ce sera fait à l'exécution. Ainsi le compilateur insère l'instruction **i2s** dans le code-octet.
- `short s = 42;` : 42 étant représentable sur 16 bits, ne demande pas de précaution particulière. Le compilateur compile "tel quel".
- `short s = 42; int i = s;` : comme un **`short`** est représenté comme un **`int`**, il n'y a pas de conversion à faire (~~`s2i`~~ n'existe pas).
- `float x = 9;` : ici on convertit une constante littérale entière en flottant. Le compilateur fait lui-même la conversion et met dans le code-octet la même chose que si on avait écrit `float x = 9.0f;`
- Mais si on écrit `int i = 9; float x = i;`, c'est différent. Le compilateur ne pouvant pas convertir lui-même, il insère **i2f** avant l'instruction qui va copier le sommet de la pile dans `x`.

Subtilités du transtypage

Cas des types référence

Types références : **exécuter un transtypage ne modifie pas l'objet référencé** [1]

- *downcasting* : le compilateur ajoute une instruction **checkcast** dans le code-octet.
  À l'exécution, **checkcast** lance une `ClassCastException` si l'objet référencé par le sommet de pile (= valeur de l'expression "castée") n'est pas du type cible.

```
// Compile et s'exécute sans erreur :
    Comestible x = new Fruit(); Fruit y = (Fruit) x;}
// Compile mais ClassCastException à  l'exécution :
    Comestible x = new Viande(); Fruit y = (Fruit) x;
// Ne compile pas !
    // Viande x = new Viande(); Fruit y = (Fruit) x;
```

- *upcasting* : invisible dans le code-octet, aucune instruction ajoutée
  → pas de conversion réelle à l'exécution. car l'inclusion des sous-types garantit, dès la compilation, que le *cast* est correct (**sous-typage inclusif**).

1. en particulier, pas son type : on a déjà vu que la classe d'un objet était définitive

Subtilités du transtypage

Cas des types référence

- Ainsi, après le *cast*, Java sait que l'objet « converti » est une instance du type cible [1].
- Les méthodes exécutées sur un objet donné (avec ou sans *cast*), sont toujours celles de sa classe, peu importe le type statique de l'expression. [2].

  Le *cast* change juste le type statique de l'expression et donc les méthodes qu'on a le droit d'appeler dessus (indépendamment de son type dynamique).

Dans l'exemple ci-dessous, c'est bien la méthode `f()` de la classe `B` qui est appelée sur la variable `a` de type `A` :

```
class A { public void f() { System.out.println("A"); } }
class B extends A { @Override public void f() { System.out.println("B"); } }
A a = new B(); // upcasting B -> A
// ici, a: type statique A, type dynamique B
a.f(); // affichera bien "B"
```

1. Sans que l'objet n'ait jamais été modifié par le *cast* !
2. Principe de la **liaison dynamique**.

Polymorphisme

Un principe fondamental de la POO

Definition (Polymorphisme)

Une instruction/une méthode/une classe/... est dite **polymorphe** si elle peut travailler sur des données de types concrets différents, qui se comportent de façon similaire.

- Le fait pour un même morceau de programme de pouvoir fonctionner sur des types concrets différents favorise de façon évidente la réutilisation.
- Or tout code réutilisé, plutôt que dupliqué quasi à l'identique, n'a besoin d'être corrigé qu'une seule fois par bug détecté.
- Donc le polymorphisme aide à « bien programmer », ainsi la POO en a fait un de ses « piliers ».

Il y a en fait plusieurs formes de polymorphisme en Java...

Polymorphisme

Exemples et mécanismes divers

- L'opérateur + fonctionne avec différents types de nombres. C'est une forme de polymorphisme résolue à la compilation [1].
- ```
  static void f(Showable s, int n) {
      for(int i = 0; i < n; i ++) s.show();
  }
  ```

  `f` est polymorphe : toute instance directe ou indirecte de `Showable` peut être passé à cette méthode, sans recompilation !

  L'appel `s.show()` fonctionne toujours car, à son exécution, la JVM cherche une implémentation de `show` dans la classe de `s` (liaison dynamique), or le sous-typage garantit qu'une telle implémentation existe.

- Et dans l'exemple suivant : `System.out.println(z);`, `z` peut être de n'importe quel type. Quel(s) mécanisme(s) intervien(en)t-il(s) ? [2]

1. En fonction du type des opérandes, `javac` traduit "+" par une instruction parmi **dadd**, **fadd**, **iadd** et **ladd**.
2. Consultez la documentation de la classe `java.io.PrintStream` !
  ```

Formes de polymorphisme

Les 3 formes de polymorphisme en Java

- **polymorphisme *ad hoc*** (via la surcharge) : le même code recompilé dans différents contextes peut fonctionner pour des types différents.

  Attention : résolution à la compilation → après celle-ci, type concret fixé.

  Donc pas de réutilisation du code compilé → forme très faible de polymorphisme.

→ **polymorphisme par sous-typage** : le code peut être exécuté sur des données de différents sous-types d'un même type (souvent une **interface**) sans recompilation.

→ forme classique et privilégiée du polymorphisme en POO

- **polymorphisme paramétré** :

  Concerne le code utilisant les **type génériques** (ou paramétrés, cf. généricité).

  Le même code peut fonctionner, sans recompilation [1], quelle que soit la concrétisation des paramètres.

  Ce polymorphisme permet d'exprimer des relations fines entre les types.

1. Dans d'autres langages, comme le C++, le polymorphisme paramétré s'obtient en spécialisant automatiquement le code source d'un *template* et en l'intégrant au code qui l'utilise lors de sa compilation.

Surcharge

Définition

Surcharge = situation où existent plusieurs définitions (au choix)

- dans un contexte donné d'un programme, de plusieurs méthodes de même nom ;
- dans une même classe, plusieurs constructeurs ;
- d'opérateurs arithmétiques dénotés avec le même symbole. [1].

Signature d'une méthode = *n*-uplet des types de ses paramètres formels.

Remarques :

- Interdiction de définir dans une même classe [2] 2 méthodes ayant même nom et même signature (ou 2 constructeurs de même signature).

→ 2 entités surchargées ont forcément une signature différente [3].

1. P. ex. : "/" est défini pour `int` mais aussi pour `double`
2. Les méthodes héritées comptent aussi pour la surcharge. Mais en cas de signature identique, il y a masquage et non surcharge. Donc ce qui est dit ici reste vrai.
3. Nombre ou type des paramètres différent ; le type de retour ne fait pas partie de la signature et n'a rien à voir avec la surcharge !

Surcharge

Résolution : définitions préalables

- Une signature $(p_1, \ldots, p_n)$ **subsume** une signature $(q_1, \ldots, q_m)$ si $n = m$ et $\forall i \in [1, n], p_i :> q_i$.
  Dit autrement : une signature subsumant une autre accepte tous les arguments acceptés par cette dernière.
- Pour chaque appel de méthode $f(e_1, e_2, ..., e_n)$ dans un code source, la **signature d'appel** est le n-uplet de types $(t_1, t_2, ..., t_n)$ tel que t_i est le type de l'expression e_i (tel que détecté par le compilateur).

Surcharge

Résolution (simplifiée...)

Pour un appel à "`f`" donné, le compilateur va :

1. lister les méthodes de nom `f` du contexte courant ;
2. garder celles dont la signature subsume la signature d'appel (= trouver les méthodes admissibles) ;
3. éliminer celles dont la signature subsume la signature d'une autre candidate (= garder seulement les signatures les plus spécialisées) ;
4. appliquer quelques autres règles [1] pour éliminer d'autres candidates ;
5. s'il reste plusieurs candidates à ce stade, renvoyer une erreur (appel ambigü) ;
6. sinon, inscrire la référence de la dernière candidate restante dans le code octet. C'est celle-ci qui sera appelée à l'exécution. [2]

1. Notamment liées à l'héritage, nous ne détaillons pas.
2. Exactement celle-ci pour les méthodes statiques. Pour les méthodes d'instance, on a juste déterminé que la méthode qui sera choisie à l'exécution aura cette signature-là. Voir liaison dynamique.

Surcharge

Exemples


```java
public class Surcharge {
    public static void f(double z) { System.out.println("double"); }

    public static void f(int x) { System.out.println("int"); }

    public static void g(int x, double z) { System.out.println("int double"); }

    public static void g(double x, int z) { System.out.println("double int"); }

    public static void main(String[] args) {
        f(0); // affiche "int"
        f(0d); // affiche "double"
        // g(0, 0); ne compile pas
        g(0d, 0); // affiche "double int"
    }
}
```

Surcharge

Pourquoi elle ne permet que du polymorphisme “faible”


```java
public class PolymorphismeAdHoc {
    public static void f(String s) { ... }
    public static void f(Integer i) { ... }
    public static void g(??? o) { // <-- par quoi remplacer "???" ?
        f(o); // <-- instruction supposée "polymorphe"
    }
}
```

`g( )` doit être recompilée en remplaçant les `???` par `String` ou `Integer` pour accepter l’un ou l’autre de ces types (mais pas les 2 dans une même version du programme).

Surcharge

Pourquoi elle ne permet que du polymorphisme “faible”

Alternative, écrire la méthode, une bonne fois pour toutes, de la façon suivante :

```
public static void g(Object o) { // méthode "réellement" polymorphe
    if (o instanceof String) f((String) o);
    else if (o instanceof Integer) f((Integer) o);
    else { /* gérer l'erreur */ }
}
```

Mais ici, c’est en réalité du polymorphisme par sous-typage [1] (de `Object`).

1. En fait, une forme bricolée, maladroite de celui-ci : il faut, autant que possible, éviter **`instanceof`** au profit de la liaison dynamique.

Interfaces

Pour réaliser le polymorphisme via le sous-typage, de préférence, on définit une **interface**, puis on la fait **implémenter** par plusieurs classes.

Plusieurs façons de voir la notion d'interface :

- supertype de toutes les classes qui l'implémentent :
  Si la classe `Fruit` implémente l'interface `Comestible`, alors on a le droit d'écrire :

  ```
  Comestible x = new Fruit();
  ```

  (parce qu'alors `Fruit` $<:$ `Comestible`)
- contrat qu'une classe qui l'implémente doit respecter
  (ce contrat n'est pas entièrement écrit en Java, cf. *Liskov substitution principle*).
- type de tous les objets qui respectent le contrat.
- mode d'emploi pour utiliser les objets des classes qui l'implémentent.

Rappel : type de données $\leftrightarrow$ ce qu'il est possible de faire avec les données de ce type.
En POO $\rightarrow$ messages qu'un objet de ce type peut recevoir (méthodes appelables).

Interfaces : syntaxe de la déclaration


```
public interface Comparable { int compareTo(Object other); }
```

Déclaration comme une classe, en remplaçant **`class`** par **`interface`**, mais : [1]

- constructeurs interdits ;
- tous les membres implicitement [2] **`public`** ;
- attributs implicitement **`static final`** (= constantes) ;
- types membres nécessairement et implicitement **`static`** ;
- méthodes d'instance implicitement **`abstract`** (simple déclaration sans corps) ;
- méthodes d'instance non-abstraites signalées par mot-clé **`default`** ;
- les méthodes **`private`** sont autorisées (annule **`public`** et **`abstract`**), autres membres obligatoirement **`public`** ;
- méthodes **`final`** interdites.

1. Méthodes **`static`** et **`default`** depuis Java 8, **`private`** depuis Java 9.
2. Ce qui est implicite n'a pas à être écrit dans le code, mais peut être écrit tout de même.

Interfaces : syntaxe de la déclaration

Pourquoi ces contraintes ?

Limites dues plus à l'idéologie (qui s'est assouplie) qu'à la technique. [1]

- **À la base :** interface = juste description de la communication avec ses instances.
- **Mais, dès le début,** quelques « entorses » : constantes statiques, types membres.
- **Java 8** permet qu'une interface contienne des implémentations → la construction `interface` va au delà du concept d'« interface » de POO.
- **Java 9** ajoute l'idée que ce qui n'appartient pas à l'« interface » (selon POO) peut bien être privé (pour l'instant seulement méthodes).

Ligne rouge pas encore franchie : une interface ne peut pas imposer une implémentation à ses sous-types (interdits : constructeurs, attributs d'instance et méthodes `final`).

Conséquence : une interface n'est pas non plus directement [2] instanciable.

1. Il y a cependant des vraies contraintes techniques, notamment liées à l'héritage multiple.
2. Mais très facile via classe anonyme : `new UneInterface(){ ... }`.

Interfaces : usage (1)


```
public interface Comparable { int compareTo(Object other); }

class Mot implements Comparable {
  private String contenu;

  public int compareTo(Object other) {
    return ((Mot) autreMot).contenu.length() - contenu.length();
  }
}
```

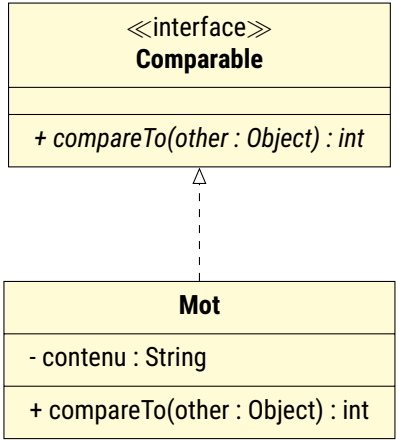
- Mettre **`implements`** `I` dans l'en-tête de la classe `A` pour implémenter l'interface `I`.
- Les méthodes de `I` sont définissables dans `A`. Ne pas oublier d'écrire **`public`**.
- Pour obtenir une « vraie » classe (non abstraite, i.e. instanciable) : nécessaire de définir toutes les méthodes abstraites promises dans l'interface implémentée.
- Si toutes les méthodes promises ne sont pas définies dans `A`, il faut précéder la déclaration de `A` du mot-clé **`abstract`** (classe abstraite, non instanciable)
- Une classe peut implémenter plusieurs interfaces : **`class`** `A` **`implements`** `I, J, K { ... }`.

Interfaces : usage (2)

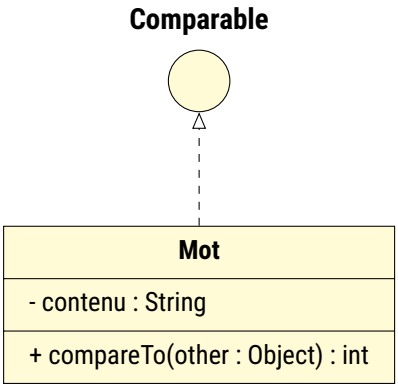
une méthode de tri polymorphe


```java
public class Tri {
    static void trie(Comparable [] tab) {
    /* ... algorithme de tri
        utilisant tab[i].compareTo(tab[j])
        ...
    */
    }

    public static void main(String [] argv) {
    Mot [] tableau = creeTableauMotsAuHasard();
        // on suppose que creeTableauMotsAuHasard existe
    trie(tableau);
        // Mot [] est compatible avec Comparable []
    }
}
```

Interfaces : notation UML



ou version ”abrégée” :



Notez l’italique pour la méthode abstraite et la flèche utilisée (pointillés et tête en triangle côté interface) pour signifier ”implémente”.

Interfaces : méthodes par défaut (Java $\geq$ 8)

- **Méthode par défaut** : méthode d'instance, non abstraite, définie dans une interface. Sa déclaration est précédée du mot-clé `default`.
- N'utilise pas les attributs de l'objet, encore inconnus, mais peut appeler les autres méthodes déclarées, même abstraites.
- Utilité : implémentation par défaut de cette méthode, héritée par les classes qui implémentent l'interface $\rightarrow$ moins de réécriture.
- Possibilité d'une forme (faible) d'héritage multiple (via superclasse + interface(s) implémentée(s)).
- **Avertissement :** héritage possible de plusieurs définitions pour une même méthode par plusieurs chemins.

  Il sera parfois nécessaire de « désambiguër » (on en reparlera).

Interfaces : méthodes par défaut

Exemple


```
interface ArbreBinaire {
  ArbreBinaire gauche();
  ArbreBinaire droite();
  default int hauteur() {
    ArbreBinaire g = gauche();
    int hg = (g == null)?0:g.hauteur();
    ArbreBinaire d = droite();
    int hd = (d == null)?0:d.hauteur();
    return 1 + (hg>hd)?hg:hd;
  }
}
```

Remarque : on ne peut pas (re)définir par défaut des méthodes de la classe `Object` (comme `toString` et `equals`).

Raison : une méthode par défaut n'est là que... par défaut. Toute méthode de même nom héritée d'une classe est prioritaire. Ainsi, une implémentation par défaut de `toString` serait tout le temps ignorée.

Héritage d'implémentations multiples

À cause des méthodes par défaut des interfaces

Une classe peut hériter de plusieurs implémentations d'une même méthode, via les interfaces qu'elle implémente (méthodes **`default`**, Java $\geq$ 8).

Cela peut créer des ambiguïtés qu'il faut lever. Par exemple, le programme ci-dessous est ambigu et **ne compile pas** (quel sens donner à `new A().f()` ?).

```
interface I { default void f() { System.out.println("I"); } }
interface J { default void f() { System.out.println("J"); } }
class A implements I, J {}
```

Pour le corriger, il faut redéfinir `f()` dans `A`, par exemple comme suit :

```
class A implements I,J {
    @Override public void f() { I.super.f(); J.super.f();
}
```

Cette construction `NomInterface.super.nomMethode()` permet de choisir quelle version appeler dans le cas où une même méthode serait héritée de plusieurs façons.

Héritage d'implémentations multiples

À cause des méthodes par défaut des interfaces

Quand une implémentation de méthode est héritée à la fois d'une superclasse et d'une interface, c'est la version héritée de la classe qui prend le dessus.

Java n'oblige pas à lever l'ambiguïté dans ce cas.

```
interface I {
    default void f() { System.out.println("I"); }
}

class B {
    public void f() { System.out.println("B"); }
}

class A extends B implements I {}
```

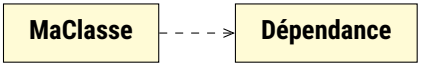
Ce programme compile et `new A().f();` affiche B.

Du bon usage des interfaces

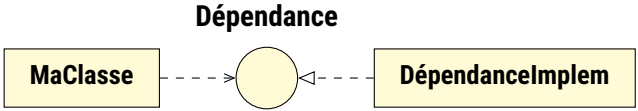
Programmez à l'interface (1)

Évitez d'écrire, dans votre programme, le nom [1] des classes des objets qu'il utilise.

Cela veut dire, évitez :



et préférez :



Cela s'appelle « **programmer à l'interface** ».

1. On parle alors de dépendance statique, c'est-à-dire le fait de citer nommément une entité externe (p.e. une autre classe) dans un code source.
Le fait de référencer un objet d'une autre classe à un moment de l'exécution ne compte pas.

Du bon usage des interfaces

Programmez à l'interface (2)

- plutôt facile quand le nom de classe est utilisé en tant que type (notamment dans déclarations de variables et de méthodes)
  → remplacer par des noms d'interfaces (ex : `List` à la place de `ArrayList`)
- pour instancier ces types, il faut bien que des constructeurs soient appelés, mais :
  - si vous codez une bibliothèque, laissez vos clients vous fournir vos dépendances (p. ex. : en les passant au constructeur de votre classe) → **injection de dépendance**[1]

```
public class MyLib {
  private final SomeInterface aDependency;
  public MyLib(SomeInterface aDependency) { this.aDependency = aDependency; }
}
```

  - sinon, circonscrire le problème en utilisant des **fabriques**[2] définies ailleurs (par vous ou par un tiers) : `List<Integer> l = List.of(4, 5, 6);`

1. Ici, injection via paramètre du constructeur. Mais il existe des *frameworks* d'injection de dépendance.
2. Plusieurs variantes du patron « fabrique », cf. GoF. Variante la plus aboutie : fabrique abstraite (*abstract factory*). Le client ne dépend que de la fabrique abstraite, la fabrique concrète est elle-même injectée !

Du bon usage des interfaces

Programmez à l'interface (3)

Pourquoi programmer à l'interface :

Une classe qui mentionne par son nom une autre classe contient une dépendance statique [1] à cette dernière. Cela entraîne des rigidités.

Au contraire, une classe A programmée « à l'interface », est

- polymorphe : on peut affecter à ses attributs et passer à ses méthodes tout objet implémentant la bonne interface, pas seulement des instances d'une certaine classe fixée « en dur ».
  $\rightarrow$ gain en adaptabilité
- évolutive : il n'y a pas d'engagement quant à la classe concrète des objets retournés par ses méthodes.
  Il est donc possible de changer leur implémentation sans « casser » les clients de A.

1. = écrite « en dur », sans possibilité de s'en dégager à moins de modifier le code et de le recompiler.

Du bon usage des interfaces

Fabriques abstraites

Besoin : dans `MyClass`, créer des instances d'une interface `Dep` connue, mais d'implémentation inconnue à l'avance.

Réponse classique : on écrit une interface `DepAbstractFactory` et on ajoute au constructeur de `MyClass` un argument `DepAbstractFactory factory`. Pour créer une instance de `Dep` on fait juste `factory.create()`.

```
public interface DepAbstractFactory { Dep create(); }
public class MyClass {
  private final DepAbstractFactory factory;
  public MyClass(DepAbstractFactory factory) { this.factory = factory; }
  /* plus loin */ Dep uneInstanceDeDep = factory.create();
}
```

```
// programme client
public class DepImpl implements Dep { ... }
public class DepContreteFactory implements DepAbstractFactory {
  @Override public Dep create() { return new DepImpl(...); }
}
  /* plus loin */ MyClass x = new MyClass(new MyDepFactory());
```

Du bon usage des interfaces

Fabriques abstraites à l'âge des lambdas

Version moderne : remplacer `DepFactory` par `java.util.function.Supplier`[1] :

```
public class MyClass {
  private final Supplier<Dep> factory;
  public MyClass(Supplier<Dep> factory) { this.factory = factory; }
  /* plus loin */ Dep uneInstanceDeDep = factory.get();
}
```

```
// programme client
public class DepImpl implements Dep { ... }
  /* plus loin */ MyClass x = new MyClass(() -> new DepImpl(...));
```

1. Si on veut quand-même déclarer `DepAbsractFactory`, l'usage de lambda-expressions reste possible à condition de n'y mettre qu'une seule méthode.

Du bon usage des interfaces

Principe d'inversion de dépendance [4] (DIP) (1)

- **Quand ?** quand on programme une bibliothèque dependant d'un certain composant et qu'il n'existe pas d'interface « standard » décrivant exactement les fonctionnalités de celui-ci. [1].
- **Quoi ?**

  → on définit alors une interface idéale que la dépendance devrait implémenter et on la joint au

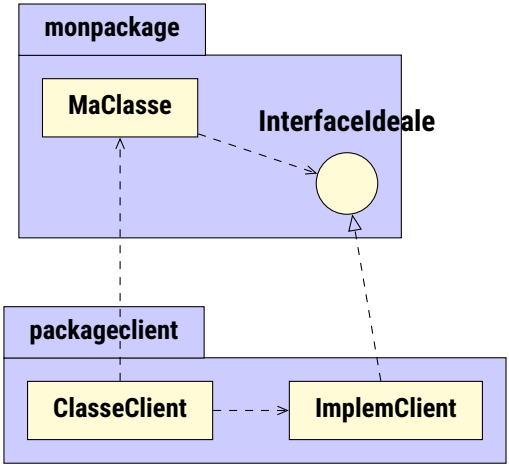
  package [2] de la bibliothèque.

  Les utilisateurs de la bibliothèque auront alors charge d'implémenter cette interface [3] (ou de choisir une implémentation existante) pour fournir la dépendance.

1. Ou simplement parce que vous voulez avoir le contrôle de l'évolution de cette interface.
2. Si on utilise JPMS : ce sera un des *packages* exportés.
3. Typiquement, les utilisateurs employeront le patron « adaptateur » pour implémenter l'interface fournie à partir de diverses classes existantes.
4. Le « D » de SOLID (Michael Feathers & Robert C. Martin)

Du bon usage des interfaces

Principe d'inversion de dépendance (2), sous forme de diagramme UML



(remarquer le sens des flèches entre les 2 *packages*)

Du bon usage des interfaces

Principe d'inversion de dépendance (3)

- **Pourquoi faire cela ?**
  - l'interface écrite est idéale et facile à utiliser pour programmer la bibliothèque
  - ses évolutions restent sous le contrôle de l'auteur de la bibliothèque, qui ne peut donc plus être « cassée » du fait de quelqu'un d'autre
  - la bibliothèque étant « programmée à l'interface », elle sera donc polymorphe.

- **Pourquoi dit-on « inversion » ?**

  Parce que le code source de la bibliothèque qui dépend, à l'exécution, d'un composant supposé plus « concret » [1], ne dépend pas statiquement de la classe implémentant ce dernier. Selon le DIP, c'est le contraire qui se produit (dépendance à l'interface).

  En des termes plus savants :
  « *Depend upon Abstractions. Do not depend upon concretions.* » [2].

1. et donc d'implémentation susceptible de changer plus souvent (justification du DIP par son inventeur)
2. Robert C. Martin (2000), dans *"Design Principles and Design Patterns"*.

Du bon usage des interfaces

Le patron « adaptateur » (GoF) (1)

- **Quand ?**
  - vous voulez utiliser une bibliothèque dont les méthodes ont des paramètres typés par une certaine interface `I`.
  - mais vous ne disposez pas de classe implémentant `I`
  - cependant, une autre bibliothèque vous fournit une classe `C` contenant la même fonctionnalité que `I` (ou presque)
- **Quoi ?** On crée alors une classe de la forme suivante :

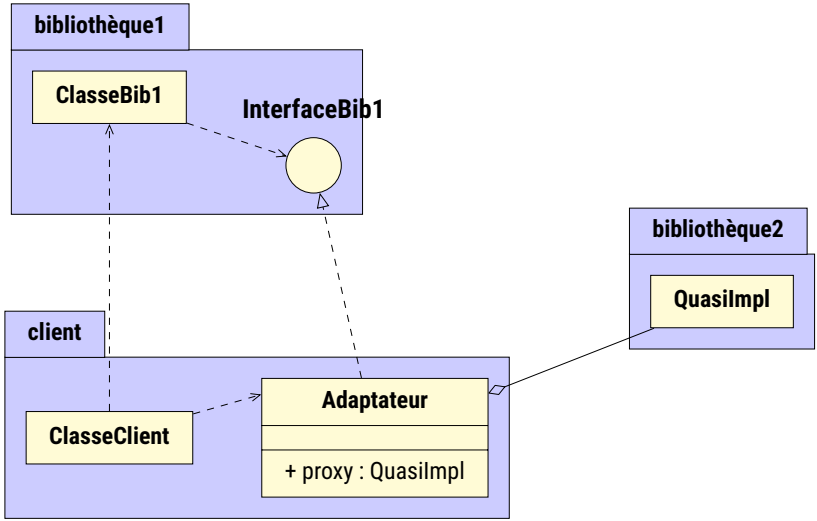
```
public class CToIAdapter implements I {
  private final C proxy;
  public CToIAdapter(C proxy) { this.proxy = proxy; }
  ...
}
```

et dans laquelle les méthodes de `I` sont implémentées [1] par des appels de méthodes sur `proxy`.

1. De préférence très simplement et brièvement...

Du bon usage des interfaces

Le patron « adaptateur » (GoF) (2), sous forme de diagramme UML



Héritage

Obtenir réutilisation et sous-typage avec élégance

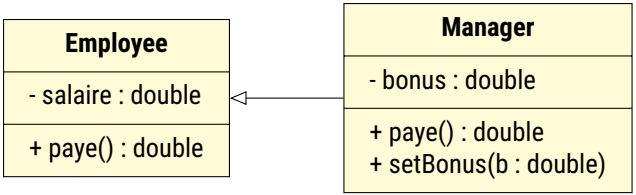
- L'**héritage** est un mécanisme pour définir une nouvelle classe [1] B à partir d'une classe existante A : B récupère les caractéristiques [2] de A et en ajoute de nouvelles.
- Ce mécanisme permet la réutilisation de code.
- L'héritage implique le sous-typage : les instances de la nouvelle classe **sont** [3] ainsi des instances (indirectes) de la classe héritée avec quelque chose en plus.

1. Ou bien une nouvelle interface à partir d'une interface existante.
2. concrètement : les membres
3. Par opposition au mécanisme de composition : dans ce cas, on remplacerait « sont » par « contiennent ».

Héritage – syntaxe…

… par l'exemple


```java
class Employee {
    private double salaire;
    public Employee(double s) { salaire = s; }
    public double paye() { return salaire; }
}

class Manager extends Employee { // <------ c'est là que ça se passe !
    private double bonus;

    public Manager(double s, double b) {
        super(s); // <------ appel du constructeur parent
        bonus = b;
    }

    public void setBonus(double b) { bonus = b; }

    @Override
    public double paye() { // <------ redéfinition !
        return super.paye() + bonus;
    }
}
```



Notez la flêche : trait plein et tête en triangle côté superclasse, pour signifier ”hérite de”.

La méthode redéfinie (`paye`) apparaît à nouveau dans la sous-classe.

Héritage

Folklore et « piliers » de la POO, avertissement

- D'après le folklore : **« piliers » de la POO** = encapsulation, polymorphisme et héritage.
- **Mais ce dernier principe ne définit pas du tout la POO !** [1]
- Héritage : mécanisme de réutilisation de code très pratique, mais non fondamental.
- ∃ LOO sans héritage : les premières versions de Smalltalk ; le langage Go. La POO moderne incite aussi à préférer la composition à l'héritage. [2]

Avertissement : l'héritage, mal utilisé, est souvent source de rigidité ou de fragilité [3]. Il faudra, suivant les cas, lui préférer l'implémentation d'interface ou la composition.

Faiblesses de l'héritage et alternatives possibles seront discutées à la fin de ce chapitre.

---

1. Rappel : POO = programmation faisant communiquer des objets.
2. Ce qui n'empêche que l'héritage soit évidemment au programme de ce cours.
3. EJ3 19 : « *Design and document for inheritance or else prohibit it* »

Extension vs. implémentation

En POO, en théorie :

- implémentation d'interface ↔ spécification d'un supertype
- héritage/extension ↔ récupération des membres hérités (= facilité syntaxique)

En Java, en pratique, distinction moins claire, car :

- implémentation et héritage impliquent tous deux le sous-typage
- quand on « implémente » on hérite des implémentations par défaut (**`default`**).

Les différences qui subsistent :

- extension de classe : la seule façon d'hériter de la description concrète d'un objet (attributs hérités + usage du constructeur **`super`**`()`);
- implémentation d'interface : seule façon d'avoir plusieurs supertypes directs.

Héritage

De classe ou d'interface ?

En Java, la notion d'héritage concerne à la fois les classes et les interfaces.

L'héritage n'a pas la même structure dans les 2 cas :

- Une classe peut hériter directement d'une (et seulement une) classe (« **héritage simple** »).
  Par ailleurs, toutes les classes héritent de la classe `Object`.
- Une interface peut hériter directement d'une ou de plusieurs interfaces.
  Elle peut aussi n'hériter d'aucune interface (pas d'ancêtre commun).

Remarque : avec l'héritage, on reste dans une même catégorie, classe ou interface, par opposition à la relation d'implémentation.

Héritage de classe et d'interface, implémentation et sous-typage

Comparatif des 4 relations

Pour résumer, il est possible de comparer 4 relations différentes :

	héritage de classe	héritage d'interface	implémentation d'interface	sous-typage de types référence
mot-clé	**`extends`**	**`extends`**	**`implements`**	(tout ça)
parent	classe	interface	interface	type
enfant	classe	interface	classe	type
nb. parents	1 [1]	≥ 0	≥ 0	≥ 1 [2]
graphe	arbre	DAG	DAG, hauteur 1	DAG
racine(s)	classe `Object`	multiples	multiples	type `Object`

1. 0 pour classe `Object`
2. 0 pour type `Object`

Héritage : vocabulaire

Héritage direct ou extension

- Une classe [1] A peut « **étendre** »/« hériter directement de »/« être dérivée de/être une sous-classe directe d'une autre classe B. Nous noterons $A \prec B$.
  (Par opposition, B est appelée **superclasse directe** ou classe mère de A : $B \succ A$.)
- Alors, tout se passe comme si les membres visibles de B étaient aussi définis dans A. On dit qu'ils sont **hérités** Conséquences :
  1. toute instance de A peut être utilisée comme [2] instance de B ;
  2. donc une expression de type A peut être substituée à une expression de type B.

  Le système de types en tient compte : $A \prec B \implies A <: B$ (A sous-type de B).
- Dans le code de A, le mot-clé `super` est synonyme de B. Il sert à accéder aux membres de B, même masqués par une définition dans A (ex : `super.f();`).

1. Pour l'héritage d'interfaces : remplacer partout « classe » par interface.
2. Parce qu'on peut demander à l'instance de A les mêmes opérations qu'à une instance de B (critère bien plus faible que le principe de substitution de Liskov !).

Héritage : vocabulaire

Héritage généralisé

Héritage (sous-entendu : « généralisé ») :

- Une classe A **hérite de**/est une **sous-classe** d'une autre classe B s'il existe une séquence d'extensions de la forme : $A \prec A1 \prec \cdots \prec An \prec B$ [1].
  Notation : $A \sqsubseteq B$ (remarques : $A \prec B \implies A \sqsubseteq B$, de plus $A \sqsubseteq A$).
- Par opposition, B est appelée superclasse (ou ancêtre) de A. On notera $B \sqsupseteq A$.
- Héritage implique [2] sous-typage : $A \sqsubseteq B \implies A <: B$.
- Pourtant, une classe n'hérite pas de tous les membres visibles de tous ses ancêtres, car certains ont pu être **masqués** par un ancêtre plus proche. [3]
- ~~`super.super`~~ n'existe pas ! Une classe est isolée de ses ancêtres indirects.

1. L'héritage généralisé est la fermeture transitive de la relation d'héritage direct.
2. Héritage $\Rightarrow_{\text{déf.}}$ chaîne d'héritages directs $\Rightarrow_{\prec \subset <:}$ chaîne de sous-typage $\Rightarrow_{\text{transitivité de } <:}$ sous-typage.
3. C'est pourquoi je distingue héritage direct et généralisé. **Attention :** une instance d'une classe contient, physiquement, tous les attributs d'instance définis dans ses superclasses, même masqués ou non visibles.

Héritage : la racine

La classe `Object`

La classe `Object` est :

- superclasse de toutes les classes ;
- superclasse directe de toutes les classes sans clause **`extends`** (dans ce cas, « **`extends`** `Object` » est implicite) ;
- racine de l'arbre d'héritage des classes [1].

Et le type `Object` qu'elle définit est :

- supertype de tous les types références (y compris interfaces) ;
- supertype direct des classes sans clause ni **`extends`** ni **`implements`** et des interfaces sans clause **`extends`** ;
- unique source du graphe de sous-typage des types références.

1. Ce graphe a un degré d'incidence de 1 (héritage simple) et une source unique, c'est donc un arbre. Notez que le graphe d'héritage des interfaces n'est pas un arbre mais un DAG (héritage multiple) à plusieurs sources et que le graphe de sous-typage des types références est un DAG à source unique.

Héritage : la racine

La classe `Object` : ses méthodes (1)

`Object` possède les méthodes suivantes :

- **`boolean`** `equals(Object other)` : teste l'égalité de **`this`** et `other`
- `String toString()` : retourne la représentation en `String` de l'objet [1]
- **`int`** `hashCode()` : retourne le « hash code » de l'objet [2]
- `Class<?> getClass()` : retourne l'objet-classe de l'objet.
- **`protected`** `Object clone()` : retourne un « clone » [3] de l'objet si celui-ci est `Cloneable`, sinon quitte sur exception `CloneNotSupportedException`.
- **`protected void`** `finalize()` : appelée lors de la destruction de l'objet.
- et puis `wait`, `notify` et `notifyAll` que nous verrons plus tard (cf. *threads*).

1. Utilisée notamment par `println` et dans les conversions implicites vers `String` (opérateur « + »).
2. Entier calculé de façon déterministe depuis les champs d'un objet, satisfaisant, par contrat, `a.equals(b)` $\implies$ `a.hashCode()== b.hashCode()`.
3. Attention : le rapport entre `clone` et `Cloneable` est plus compliqué qu'il en a l'air, cf. EJ3 Item 13.

Héritage : la racine

La classe `Object` : ses méthodes (2)

Conséquences :

- Grâce au sous-typage, tous les types référence ont ces méthodes, on peut donc les appeler sur toute expression de type référence.
- Grâce à l'héritage, tous les objets disposent d'une implémentation de ces méthodes...
  ... mais leur implémentation faite dans `Object` est souvent peu utile :
  - `equals` : teste l'identité (égalité des adresses, comme ==) ;
  - `toString` : retourne une chaîne composée du nom de la classe et du `hashCode`.

  $\rightarrow$ toute classe devrait redéfinir `equals` (et donc [1] `hashCode`) et `toString` (cf. EJ3 Items 10, 11, 12).

1. Rappel du transparent précédent : si `a.equals(b)` alors il faut `a.hashCode()== b.hashCode()`.

Héritage : ajout, redéfinition, masquage

Le panorama... (1)

Dans une sous-classe :

- On **hérite** des membres visibles [1] de la superclasse directe [2].
  Visible = **`public`**, **`protected`**, voire *package-private*, si superclasse dans même *package*.
- On peut **masquer** (*to hide*) n'importe quel membre hérité :
  - méthodes : par une définition de même signature
    dans ce cas, le type de retour doit être identique [3], sinon erreur de syntaxe !
  - autres membres : par une définition de même nom
- Les autres membres de la sous-classe sont dits **ajoutés**.

...

1. Il faut en fait visibles et non-**`private`**. En effet : **`private`** est parfois visible (cf. classes imbriquées).
2. que ceux-ci y aient été directement définis, ou bien qu'elle les aie elle-même hérités
3. En fait, si le type de retour est un type référence, on peut retourner un sous-type. Par ailleurs il y a des subtilités dans le cas des types paramétrés, cf généricité.

Héritage : ajout, redéfinition, masquage

Le panorama... (2)

...

- Une méthode d'instance (non statique) masquée est dite **redéfinie** [1] (*overridden*).

  Dans le cas d'une redéfinition, il est interdit de :

  - redéfinir une méthode `final`,
  - réduire la visibilité (*e.g.* redéfinir une méthode `public` par une méthode `private`),
  - ajouter une clause `throws` ou bien d'ajouter une exception dans la clause `throws` héritée (cf. cours sur les exceptions).

  La notion de redéfinition est importante en POO (cf. liaison dynamique).

1. Mon parti pris : redéfinition = cas particulier du masquage. D'autres sources restreignent, au contraire, la définition de « masquage » aux cas où il n'y a pas de redéfinition (« masquage simple »).
La JLS dit que les méthodes d'instance sont redéfines et jamais qu'elles sont masquées... mais ne dit pas non plus que le terme est inapproprié.

Héritage : ajout, redéfinition, masquage

Remarques diverses (1)

- L'accès à toute définition visible (masquée ou pas) de la superclasse est toujours possible via le mot-clé **`super`**. Par ex. : **`super`**`.toString()`.
- Les définitions masquées ne sont pas effacées. En particulier, un attribut masqué contient une valeur indépendante de la valeur de l'attribut qui le masque.

```
class A { int x; }
class B extends A { int x; } // le x de A est masqué par celui-ci
...
B b = new B(); // <- mais cet objet contient bien deux int
```

- De même, les définitions non visibles des superclasses restent « portées » par les instances de la sous-classe, même si elles ne sont pas accessibles directement.

```
class A { private int x; } // x privé, pas hérité par classe B
class B extends A { int y; }
...
B b = new B(); // <- mais cet objet contient aussi deux int
```

Héritage : ajout, redéfinition, masquage

Remarques diverses (2)

- Les membres non hérités ne peuvent pas être masqués ou redéfinis, mais rien n'empêche de définir à nouveau un membre de même nom (= ajout).

```
class A { private int x; }
class B extends A { int x; } // autorisé !
// et tant qu'on y est :
B b = new B(); // là  encore, cet objet contient deux int !
```

Héritage : ajout, redéfinition, masquage

Exemple


```
class GrandParent {
    protected static int a, b; // visibilité protected, assure que l'héritage se fait bien
    protected static void g() {}
    protected void f() {}
}

class Parent extends GrandParent {
    protected static int a; // masque le a hérité de GrandParent (tjs accessible via super.a)
    // masque g() hérité de GrandParent (tjs appelable via super.g()).
    protected static void g() {}
    // redéfinit f() hérité de GrandParent (tjs appelable via super.f()).
    @Override protected void f() {}
}

class Enfant extends Parent { @Override protected void f() {} }
```

- La classe `Enfant` hérite `a`, `g` et `f` de `Parent` et `b` de `GrandParent` via `Parent`.
- `a` et `g` de `GrandParent` masqués mais accessibles via préfixe `GrandParent.`.
- `f` de `Parent` héritée mais redéfinie dans `Enfant`. Appel de la version de `Parent` avec `super.f()`.
- `f` de `GrandParent` masquée par celle de `Parent` mais peut être appelée sur un récepteur de classe `GrandParent`. **Remarque :** ~~`super.super`~~ n'existe pas.

Héritage : ajout, redéfinition, masquage

Cas tordu

Un ajout simple dans un contexte peut provoquer un masquage dans un autre :

```java
package bbb;
public class B extends aaa.A {
    public void f() { System.out.println("B"); }    // avec @Override, ça ne compilerait pas.
    // En effet, bbb.B ne voit pas la f de aaa.A. C'est donc un ajout de nouvelle méthode !
}
```

```java
package aaa;
public class A {
    void f() { System.out.println("A"); } // méthode package-private, invisible dans bbb
    public static void main(String[] args) {
        bbb.B b = new bbb.B();
        b.f(); // contexte : récepteur de type B, ici f de bbb.B masque f de aaa.A
        ((A) b).f(); // contexte : récepteur de type A, f de aaa.A ni masquée ni redéfinie
    }
}
```

Ici, une méthode d'instance en masque une autre sans la redéfinir.

→ Contradiction apparente avec ce qui avait été dit.

En réalité masquage implique redéfinition seulement s'il y a masquage dans le contexte où est définie la méthode masquante.

Liaisons statique et dynamique

Non, mais sérieusement, pourquoi distinguer la redéfinition du masquage simple ?

À la compilation : dans tous les cas, chaque occurrence de nom de méthode est traduite comme référence vers une méthode existant dans le contexte d'appel.

À l'exécution :

- Autres membres que méthodes d'instance : la méthode trouvée à la compilation sera effectivement appelée.
  → Mécanisme de **liaison statique** (ou précoce).
- Méthodes d'instance [1] : une méthode **redéfinissant** la méthode trouvée à la compilation sera recherchée, depuis le contexte de la classe de l'objet récepteur.
  → Mécanisme de **liaison dynamique** (ou tardive).
  Le résultat de cette recherche peut être différent à chaque exécution.
  Ce mécanisme permet au polymorphisme par sous-typage de fonctionner.

1. Sauf méthodes privées et sauf appel avec préfixe "`super.`" → liaison statique.

Liaisons statique et dynamique

Exemple


```java
class A {
   public A() {
     f();
     g();
   }
   static void f() {
     System.out.println("A::f");
   }
   void g() {
     System.out.println("A::g");
   }
}
```

```java
class B extends A {
   public B() {
     super();
   }
   static void f() { // masquage simple
     System.out.println("B::f");
   }
   @Override
   void g() { // redéfinition
     System.out.println("B::g");
   }
}
```

Si on fait `new B();`, alors on verra s'afficher

```
A::f
B::g
```

Liaison statique

Comment le compilateur traduit-il les noms de membres apparaissant dans un code source ?

Principe de la liaison statique : dès la compilation, on décide quelle définition sera effectivement utilisée pour l'exécution (c.-à-d. **toutes** les exécutions).

Pour l'explication, nous distinguons cependant :

- d'abord le cas simple (tout membre sauf méthode)
- ensuite le cas moins simple des méthodes (possible surcharge)

Attention : seules les méthodes d'instance peuvent être liées dynamiquement (et le sont habituellement [1]) ; pour tous les autres membres elle est toujours statique.

1. Les méthodes d'instance peuvent parfois être sujets à une liaison uniquement statique : méthodes `private` ; ainsi que toute méthode lorsqu'elle est appelée avec préfixe `super.`.

Liaison statique

Principe : cas sans surcharge (membres non-méthode)

Pour trouver la bonne définition d'un membre (non méthode) de nom `m`, à un point donné du programme.

- Si *m* membre statique, soit *C* le contexte [1] d'appel de *m*. Sinon, soit *C* la classe de l'objet sur lequel on appelle *m*.
- On cherche dans le corps de *C* une définition visible et compatible (même catégorie de membre, même "staticité", même type ou sous-type...), puis dans les types parents de C (superclasse et interfaces implémentées), puis les parents des parents (et ainsi de suite).
  On utilise la première définition qui convient.

Pour les méthodes : même principe, mais on garde toutes les méthodes de signature compatible avec l'appel, puis on applique la résolution de la surcharge. Ce qui donne...

1. le plus souvent classe ou interface, en toute généralité une définition de type

Liaison statique

Principe : cas des méthodes

Quelle définition de `f` utiliser, quand on appelle `f(x1, x2, ...)` [1] ?

1. `C` := contexte de l'appel de la méthode (classe ou interface).
2. Soit M_f := { méthodes de nom « `f` » dans `C`, compatibles avec `(x1, x2, ...)`}.
3. Pour tout supertype direct `S` de `C`, M_f += { méthodes de nom « `f` » dans `S`, compatibles avec `x1, x2, ...`, non masquées [2] par autre méthode dans M_f}.
4. On répète (3) avec les supertypes des supertypes, et ainsi de suite. [3].
5. On résout la surcharge parmi M_f (i.e. : on prend la signature la plus spécifique).

Le compilateur ajoute au code-octet l'instruction **invokestatic** [4] avec pour paramètre une référence vers la méthode trouvée.

1. Avec `f` habituellement statique, mais pas toujours cf. précédemment.
2. À cause de la surcharge il peut exister des méthodes de même nom non masquées
3. Jusqu'aux racines du graphe de sous-typage.
4. Pour les méthodes statiques. Pour les méthodes d'instance **`private`** ou **`super`**, c'est **invokespecial**.

Liaison dynamique

Principe

Lors de l'appel `x.f(y)`, quelle définition de `f` choisir ?

→ Principe de la liaison dynamique :

1. à la compilation : la même recherche que pour la liaison statique est exécutée (recherche depuis le **type statique** `S` de `x`), mais le compilateur ajoute au code-octet l'instruction **invokevirtual** (si `S` est une classe) ou **invokeinterface** (si `S` est une interface) au lieu de **invokestatic**.
2. à l'exécution : quand la JVM lit **invokevirtual** ou **invokeinterface**, une **redéfinition** de la méthode trouvée en (1) est recherchée dans la classe `C` de l'objet référencé (= **type dynamique** de `x` [1]), puis récursivement dans ses superclasses successives, puis dans les interfaces implémentées (méthode **`default`**) [2] [3] .

1. Le type dynamique de `y` n'est jamais pris en compte (java est *single dispatch*).
2. En fait, seuls les supertypes de `C` sous-types de `S` peuvent contenir des redéfinitions.
3. Comme on se limite aux redéfinitions valides, les éventuelles surcharges ajoutées dans `C` par rapport à `S`, sont ignorées à cette étape. Cf. exemples.

Liaison dynamique

Table virtuelle (implémentation optimisée de la liaison dynamique)

En pratique, pour chaque classe C, la JVM établit une fois pour toutes une **table virtuelle**, associant à chaque méthode d'instance (nom et signature), un pointeur [1] vers le code qui doit être exécuté quand un appel est effectué sur une instance directe de C.

Ainsi, à chaque appel, la liaison dynamique se fait **en temps constant**.

Le calcul de cette table prend en compte les méthodes héritées, redéfinies et ajoutées :

1. la table virtuelle de C est initialisée comme copie de celle de sa superclasse ;
2. y sont ajoutées des entrées pour les méthodes déclarées dans les interfaces implémentées par C ; [2]
3. les redéfinitions de C écrasent les entrées correspondantes déjà existantes ; [3]
4. les ajouts de C sont ajoutés à la fin de la table.

1. Pointeur `null` si la méthode est abstraite.
2. Contenant `null` ou bien pointeur vers le code de la méthode `default`, le cas échéant.
3. Elles existent forcément, sinon ce ne sont pas des redéfinitions !

Liaison dynamique

Cas simple avec redéfinition

- Classe dérivée : certaines méthodes peuvent être redéfinies

```
class A { void f() { System.out.println("classe A"); } }
class B extends A { void f() { System.out.println("classe B"); } }
public class Test {
  public static void main(String args[]) {
    B b = new B();
    b.f(); // <-- affiche "classe B"
  }
}
```

- mais aussi...

```
public class Test {
  public static void main(String args[]) {
    A b = new B(); // <-- maintenant variable b de type A
    b.f(); // <-- affiche "classe B" quand-même
  }
}
```

Liaison dynamique

Cas compliqué, question ?

Imaginons le cas suivant, avec redéfinition et surcharge :

```
class Y1 {}

class Y2 extends Y1 {}

class X1 { void f(Y1 y) { System.out.print("X1 et Y1 ; "); } }

class X2 extends X1 {
    void f(Y1 y) { System.out.print("X2 et Y1 ; "); }
    void f(Y2 y) { System.out.print("X2 et Y2 ; "); }
}

class X3 extends X2 { void f(Y2 y) { System.out.print("X3 et Y2 ; "); } }

public class Liaisons {
    public static void main(String args[]) {
        X3 x = new X3(); Y2 y = new Y2();
        // notez tous les upcastings explicites ci-dessous (servent-ils vraiment à rien ?)
        ((X1) x).f((Y1) y);      ((X1) x).f(y);
        ((X2) x).f((Y1) y);      ((X2) x).f(y);
        x.f((Y1) y);             x.f(y);
    }
}
```

Qu'est-ce qui s'affiche ?

Liaison dynamique

Cas compliqué, réponse !


```
class Y1 {}
class Y2 extends Y1 {}
class X1 { void f(Y1 y) { System.out.print("X1 et Y1 ; "); } }
class X2 extends X1 {
    void f(Y1 y) { System.out.print("X2 et Y1 ; "); }
    void f(Y2 y) { System.out.print("X2 et Y2 ; "); }
}
class X3 extends X2 { void f(Y2 y) { System.out.print("X3 et Y2 ; "); } }

public class Liaisons {
    public static void main(String args[]) {
        X3 x = new X3(); Y2 y = new Y2();
        // notez tous les upcastings explicites ci-dessous (servent-ils vraiment à rien ?)
        ((X1) x).f((Y1) y);     ((X1) x).f(y);     ((X2) x).f((Y1) y);
        ((X2) x).f(y);          x.f((Y1) y);       x.f(y);
    }
}
```

Affiche : `X2 et Y1 ; X2 et Y1 ; X2 et Y1 ; X3 et Y2 ; X2 et Y1 ; X3 et Y2 ;`

- Pour les instructions commençant par `((X1)x).` : la phase statique cherche les signatures dans `X1` → les surcharges prenant `Y2` sont ignorées à l'exécution.
- Les instructions commentçant par `((X2)x).` se comportent comme celles commençant par `x.` : les mêmes signatures sont connues dans `X2` et `X3`.

Liaison dynamique

Retour sur redéfinition et surcharge

Attention aux “redéfinitions ratées” : ça peut compiler mais...

- si on se trompe dans le type ou le nombre de paramètres, ce n’est pas une redéfinition [1], mais un ajout de méthode surchargée. Erreur typique :

```
public class Object { // la ``vraie'', c.-à -d. java.lang.Object
    ...
    public boolean equals(Object obj) { return this == obj; }
    ...
}

class C /* sous-entendu : extends Object */ {
    public boolean equals (C obj) { return ....; } // <- c'est une surcharge, pas
        une redéfinition !
}
```

- **Recommandé :** placer l’annotation `@Override` devant une définition de méthode pour demander au compilateur de générer une erreur si ce n’est pas une redéfinition.

  Exemple : `@Override public boolean equals(Object obj){ ... }`

1. même pas un masquage

Méthodes et classes finales

On peut déclarer une méthode avec modificateur **`final`** [1]. Exemple :

```
class Employee {
    private String name;
    . . .
    public final String getName() { return name; }
    . . .
}
```

$\Longrightarrow$ ici, **`final`** empêche une sous-classe de `Employee` de redéfinir `getName()`. [2]

Aussi possible :

```
final class Employee {  . . . }
```

$\Longrightarrow$ ici, **`final`** interdit d'étendre la classe `Employee`

1. **Attention :** une variable peut aussi être déclarée avec le mot-clé **`final`**. Sa signification est alors différente : il interdit juste toute nouvelle affectation de la variable après son initialisation.
2. Ainsi, pour résumer, on a le droit de redéfinir les méthodes héritées non **`static`** et non **`final`**.

Méthodes et classes abstraites

- **Méthode abstraite** : méthode déclarée sans être définie.
  Pour déclarer une méthode comme abstraite, faire précéder sa déclaration du mot-clé **`abstract`**, et ne pas écrire son corps (reste la signature suivie de « **;** »).
- **Classe abstraite** : classe déclarée comme **non directement instanciable**.
  Elle se déclare en faisant précéder sa déclaration du modificateur **`abstract`** :

```
abstract class A {
    int f(int x) { return 0; }
    abstract int g(int x);   // <- oh, une méthode abstraite !
}
```

- **Le lien entre les 2 :** une méthode abstraite ne peut être pas déclarée dans un type directement instanciable → seulement dans interfaces et classes abstraites.
  Interprétation : tout objet instancié doit connaître une implémentation pour chacune de ses méthodes.
- Une méthode abstraite a vocation à être redéfinie dans une sous-classe.
  Conséquence : ~~**`abstract static`**~~, ~~**`abstract final`**~~ et ~~**`abstract private`**~~ sont des non-sens !

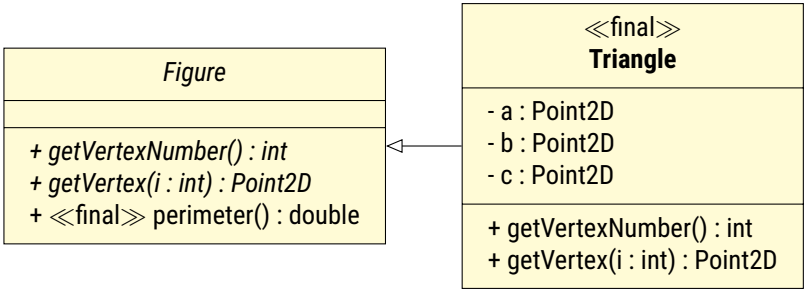
Méthodes et classes abstraites

Exemple


```java
abstract class Figure {
    Point2D centre; String nom; // autres attributs éventuellement
    public abstract int getVertexNumber();
    public abstract Point2D getVertex(int i);
    public final double perimeter() {
        double peri = 0;
        Point2D courant = getVertex(0);
        for (int i=1; i < getVertexNumber(); i++) {
            Point2D suivant = getVertex(i);
            peri += courant.distance(suivant);
            courant = suivant;
        }
        return peri + courant.distance(getVertex(0));
    }
}

final class Triangle extends Figure {
    private Point2D a, b, c;
    @Override public int getVertexNumber() {
        return 3;
    }
    @Override public Point2D getVertex(int i) {
        switch(i) {
            case 0: return a;
            case 1: return b;
            case 2: return c;
            default: throw new NoSuchElementException();
        }
    }
}
```

Méthodes et classes finales ou abstraites

Notation UML



Remarquez l'italique pour les méthodes et classes abstraites. En revanche, **`final`** n'a pas de typographie particulière [1].

1. **`final`** n'est pas un concept de la spécification d'UML, mais heureusement, UML autorise à ajouter des informations supplémentaires en tant que « stéréotypes », écrits entre doubles chevrons.

abstract et final

le bon usage pour les classes (1)

- `abstract` et `final` contraignent la façon dont une classe s'utilise.
- Pourquoi contraindre ? → empêcher une utilisation incorrecte non prévue (cf. suite).
  Plus précisément :
  - `final`, en figeant les méthodes (une ou toutes) d'une classe, permet d'assurer des propriétés qui resteront vraies pour toutes les instances de la classe.
  - `abstract` (appliqué à une classe [1]) empêche l'instanciation directe d'une classe qui serait une implémentation incomplète.

  Dans les deux cas, on interdit la possibilité d'instances absurdes (respectivement incohérents ou incomplets) de la classe marquée.

1. `abstract`, appliqué à une méthode, n'est une contrainte que dans la mesure où cela force à marquer aussi `abstract` la classe la contenant.

abstract et final

le bon usage pour les classes (2)

Constat : une classe non finale correspond à une implémentation complétable.

Idéologie : si c'est complétable c'est que c'est donc probablement incomplet. [1]

Si cela est vrai, alors une classe ni finale ni abstraite est louche !

Comme ~~`abstract final`~~ est exclus d'office, toute classe devrait alors être soit (juste) `abstract` soit (juste) `final`.

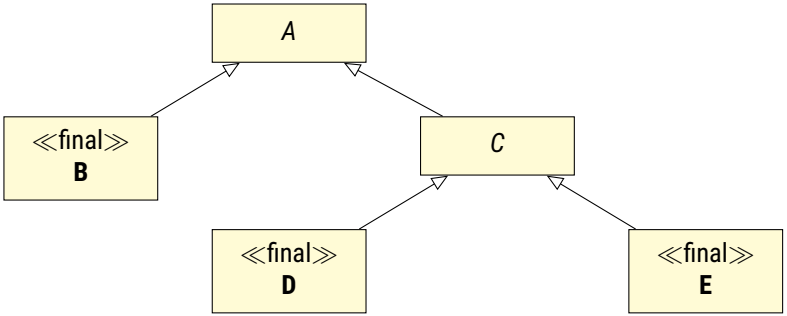
	pas `abstract`	`abstract`
pas `final`	louche (« *code smell* »)	OK
`final`	OK	ne compile pas

1. Ce n'est pas toujours vrai : certaines classes proposent un comportement par défaut tout à fait valable, tout en laissant la porte ouverte à des modifications (cf. composants Swing).

abstract et final

le bon usage pour les classes : illustration en UML

En UML, une bonne structure d'héritage selon l'idéologie ressemble à cela : [1]



1. Rappel : les classes dont le nom est en italique sont abstraites.

abstract et final

le bon usage pour les classes : exemple

Exemple (à ne pas faire !) :

```
class Personne {
    public String getNom() { return null; } // mauvaise implémentation par défaut
}

class PersonneImpl extends Personne {
    private String nom;
    @Override public String getNom() { return nom; }
}
```

Mieux :

```
abstract class Personne {
    public abstract String getNom();
}

final class PersonneImpl extends Personne {
    private String nom;
    @Override public String getNom() { return nom; }
}
```

abstract et final

le bon usage pour les méthodes : contre-exemple

À ne pas faire non plus :

```
class Personne {
    private String prenom, nom;
    public String getPrenom() { return prenom; } // il faudrait final
    public String getNom() { return nom; } // là aussi
    public String getNomComplet() {
        return getPrenom() + "␣" + getNom(); // appel à méthodes redéfinissables —> danger !!!
    }
}
```

Sans `final`, `Personne` est une **classe de base fragile**. Quelqu'un pourrait écrire :

```
class Personne2 extends Personne {
    @Override public String getPrenom() { return getNomComplet().split("␣")[0] }
    @Override public String getNom() { return getNomComplet().split("␣")[1] }
}
```

... puis exécuter `new Personne2(...).getNom()`, qui appelle `getNomComplet()`, qui appelle `getPrenom()` et `getNom()`, qui appellent `getNomComplet()` qui appelle...

Récursion non bornée ! → `StackOverflowError`.

abstract et final

le bon usage pour les méthodes

Quand on programme une classe extensible :

- Si possible, éviter tout appel, depuis une autre méthode de la classe [1], de méthode redéfinissable (= non `final` = « ouverte »).
- À défaut le signaler dans la documentation.
- Objectif : éviter des erreurs bêtes dans les futures extensions.
  Par exemple : appels mutuellement récursifs non voulus.
- La documentation devra donner une spécification des méthodes redéfinissables assurant de conserver un comportement globalement correct.

1. Cela vaut aussi pour les appels de méthodes depuis une méthode `default` dans une interface.

abstract et final

L'héritage casse-t-il l'encapsulation ?

On entend souvent dire « L'héritage casse l'encapsulation. ».

Signification : pour qu'une classe soit étendue correctement, documenter ses membres `public` ne suffit pas [1] ; certains points d'implémentation doivent aussi l'être.

→ Cela contredit l'idée que l'implémentation d'une classe devrait être une « boîte noire ».

À défaut de pouvoir faire cet effort de documentation pour une classe, il est plus raisonnable d'interdire d'hériter de celle-ci (→ `final class`).

EJ3, Item 19 : « *Design and document for inheritance or else prohibit it* »

1. De toute évidence, il faut au moins documenter les membres `protected`.

abstract et final

le bon usage : résumé

Une stratégie simple et extrême :

- Déclarer `final` toute classe destinée à être instanciée.
  ⇔ feuilles de l'arbre d'héritage.
- Déclarer `abstract` toute classe destinée à être étendue [1].
  ⇔ nœuds internes de l'arbre d'héritage.
- Dans ce dernier cas, déclarer en `private` ou `final` tous les membres qui peuvent l'être, afin d'empêcher que les extensions cassent les contrats déjà implémentés.
- Écrire la spécification de toute méthode redéfinissable (telle que, si elle est respectée, les contrats soient alors aussi respectés).

1. Voire, si la classe n'a pas d'attribut d'instance, déclarer plutôt une interface !

Application : les types scellés

Définition et intérêt

- **Type scellé** : type dont l'ensemble des sous-types directs est fixé à la compilation de celui-ci.
- **Utilité :**
  - Les fameuses listes de `else if (... instanceof ...){ ... }` deviennent plus acceptables car l'exhaustivité est garantie. Cela est utile quand la liaison dynamique ne peut pas être utilisée [1].
  - Prouver un contrat pour un type scellé revient à le prouver pour un ensemble fini et connu de sous-types directs.
    Si les sous-types sont eux-mêmes finaux ou récursivement scellés, c'est encore plus facile.

1. Notamment :
   - besoin d'écrire une méthode dont les comportements varient en fonction des types de plusieurs paramètres (impossible : la liaison dynamique est *single disptach*) ;
   - besoin d'ajouter un comportement à un type fourni par un tiers (impossible d'y ajouter une méthode).

Application : les types scellés

Les types scellés de Java

Exemples de types scellés :

- Les types primitifs : liste finie et fixe.
- Les classes `final` : n'ayant pas de sous-type du tout, elles sont scellées.
- Les `enum` : une classe d'énumération est par définition un type fini.

Y a-t-il d'autres possibilités ?

1. Depuis les classes qui sont imbriquées dans la même classe englobante de premier niveau.

Application : les types scellés

Les types scellés de Java

Exemples de types scellés :

- Les types primitifs : liste finie et fixe.
- Les classes **`final`** : n'ayant pas de sous-type du tout, elles sont scellées.
- Les **`enum`** : une classe d'énumération est par définition un type fini.

Y a-t-il d'autres possibilités ? → Oui !

- (trivialement) Les types locaux et types imbriqués privés (y compris interfaces).
- Les classes à constructeurs tous privés (une telle classe est extensible seulement depuis son groupe d'imbrication [1], c.-à-d. là où un constructeur est visible).
- Nous verrons : les **`record`**, qui sont des classes **`final`** particulières.
- ... et surtout les classes et interfaces avec le mot-clé **`sealed`**..

1. Depuis les classes qui sont imbriquées dans la même classe englobante de premier niveau.

Application : les types scellés

Les types scellés de Java

Théorème : en Java, les types scellés sont exactement ceux listés précédemment.

Preuve :

- $\Leftarrow$ Pour les raison déjà expliquées, chaque cas contient uniquement des types scellés.
- $\Rightarrow$ Réciproquement : les seuls cas hors de cette liste sont
  - les interfaces
    - de premier niveau ou membre non **`private`**
    - et sans le modificateur **`sealed`**

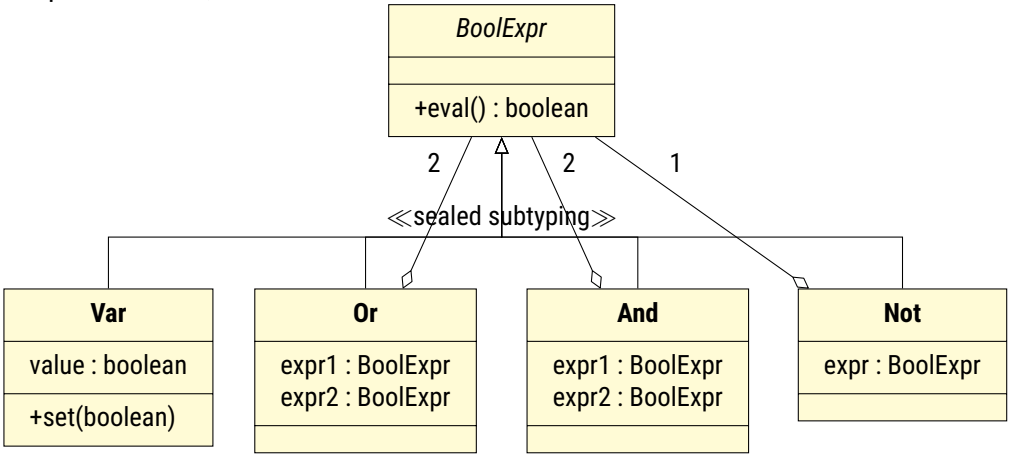
    $\rightarrow$ peuvent être implémentées et étendues sans restriction (au moins dans le *package*)
  - et les classes
    - de premier niveau ou membre non **`private`**
    - qui de plus sont ni **`sealed`** ni **`final`**
    - et possèdent un constructeur non privé

    $\rightarrow$ Java autorise à les étendre depuis un autre fichier (au moins dans le *package*)

Application : les types scellés

Exemple typique : type algébrique

Ce qu'on souhaite, décrit en UML :



Application : les types scellés

Exemple typique : type algébrique


```java
public abstract class BoolExpr { // classe scellée (et abstraite !)
    private BoolExpr() {} // constructeur privé !

    public abstract boolean eval();

    public static final class Var extends BoolExpr {
        private boolean value;
        public Var(boolean value) { this.value = value; } // super() est accessible car Var est imbriquée
        public void set(boolean newVal) { this.value = newVal; }
        @Override public boolean eval() { return value; }
    }

    public static final class Not extends BoolExpr {
        private final BoolExpr expr;
        public Not(BoolExpr expr) { this.expr = expr; } // même remarque
        @Override public boolean eval() { return !expr.eval(); }
    }

    public static final class And extends BoolExpr {
        private final BoolExpr expr1, expr2;
        public And(BoolExpr expr1, BoolExpr expr2) { this.expr1 = expr2; this.expr2 = expr2; } // idem
        @Override public boolean eval() { return expr1.eval() && expr2.eval(); }
    }

    public static final class Or extends BoolExpr {
        private final BoolExpr expr1, expr2;
        public Or(BoolExpr expr1, BoolExpr expr2) { this.expr1 = expr2; this.expr2 = expr2; } // idem
        @Override public boolean eval() { return expr1.eval() || expr2.eval(); }
    }
}
```

Application : les types scellés

Exemple typique : type algébrique, avec pattern matching


```
public abstract class BoolExpr {
    private BoolExpr() {}

    public boolean eval() { // nouveau pattern matching --> pas de cast
        if (this instanceof Var varExpr) return varExpr.get();
        else if (this instanceof Not notExpr) return !notExpr.expr.eval();
        else if (this instanceof And andExpr) return andExpr.expr1.eval() && andExpr.expr2.eval();
        else if (this instanceof Or orExpr) return orExpr.expr1.eval() || orExpr.expr2.eval();
        else { assert false : "Cannot happen : the pattern matching is exhaustive!"; return false; }
    }

    public static final class Var extends BoolExpr {
        private boolean value;
        public Var(boolean value) { this.value = value; }
        public void set(boolean newVal) { this.value = newVal; }
    }
    public static final class Not extends BoolExpr {
        private final BoolExpr expr;
        public Not(BoolExpr expr) { this.expr = expr; }
    }
    public static final class And extends BoolExpr {
        private final BoolExpr expr1, expr2;
        public And(BoolExpr expr1, BoolExpr expr2) { this.expr1 = expr1; this.expr2 = expr2; }
    }
    public static final class Or extends BoolExpr {
        private final BoolExpr expr1, expr2;
        public Or(BoolExpr expr1, BoolExpr expr2) { this.expr1 = expr1; this.expr2 = expr2; }
    }
}
```

Application : les types scellés

Le mot-clé `sealed`

Java 17 a introduit le mot-clé **`sealed`** (ainsi que **`permits`** et **`non-sealed`**)

- pour éviter de faire ce bricolage à la main
- et introduire un peu plus de souplesse :
  - **`sealed class`** et **`sealed interface`** sont possibles ;
  - et on peut séparer les déclarations en plusieurs fichiers grâce à **`permits`**.

La sous-typabilité des sous-types directs d'une classe scellée doit être écrite explicitement, en les déclarant obligatoirement avec un de ces modificateurs :

- **`final`** → interdire les sous-sous-types,
- **`sealed`** → propager le scellage aux sous-sous-types
- ou **`non-sealed`** → lever toute restriction sur les sous-sous-types
  (c'est comme s'il n'y avait pas de modificateur, mais ce choix est rendu explicite).

Application : les types scellés

Exemple typique : type algébrique, version Java moderne avec `sealed`


```java
public sealed interface BoolExpr { // sealed, pas besoin de constructeur privé ! (et interface autorisée)
    boolean eval();

    final class Var implements BoolExpr {
        private boolean value;
        public Var(boolean value) { this.value = value; }
        public void set(boolean newVal) { this.value = newVal; }
        @Override public boolean eval() { return value; }
    }
    final class Not implements BoolExpr {
        private final BoolExpr expr;
        public Not(BoolExpr expr) { this.expr = expr; }
        @Override public boolean eval() { return !expr.eval(); }
    }
    final class And implements BoolExpr {
        private final BoolExpr expr1, expr2;
        public And(BoolExpr expr1, BoolExpr expr2) { this.expr1 = expr1; this.expr2 = expr2; }
        @Override public boolean eval() { return expr1.eval() && expr2.eval(); }
    }
    final class Or implements BoolExpr {
        private final BoolExpr expr1, expr2;
        public Or(BoolExpr expr1, BoolExpr expr2) { this.expr1 = expr1; this.expr2 = expr2; }
        @Override public boolean eval() { return expr1.eval() || expr2.eval(); }
    }
}
```

Application : les types scellés

Exemple typique : type algébrique, version Java moderne éclatée avec `sealed` et `permits`


```java
public sealed interface BoolExpr
    permits Var, Not, And, Or { // permits dit quelles classes ont le droit d'implémenter
    boolean eval();
}
```

Sous-classes dans d'autres fichiers :

```java
public final class Var implements BoolExpr {
    private boolean value;
    public Var(boolean value) { this.value = value; }
    public void set(boolean newVal) { this.value = newVal; }
    @Override public boolean eval() { return value; }
}
```

```java
public final class Not implements BoolExpr {
    public final BoolExpr expr;
    public Not(BoolExpr expr) { this.expr = expr; }
    @Override public boolean eval() { return !expr.eval(); }
}
```

Et ainsi de suite.

Application : les types scellés

À propos de `permits`

La clause **`permits`** rend techniquement possible de modifier les sous-types d'un type scellé sans le recompiler, puisqu'en Java, chaque fichier est compilé séparément.

$\rightarrow$ cela affaiblit l'idée initiale que les sous-types d'un type scellé sont fixés à la compilation de celui-ci.

Avec **`permits`** seule la liste des noms de ces sous-types est fixée.

Application : les types scellés

Exemple typique : type algébrique, avec pattern-matching switch (attention : Java 17-19 preview seulement)


```
public sealed interface BoolExpr {
    default boolean eval() {
        return switch(this) {
            case Var varExpr -> varExpr.value;
            case Not notExpr -> !notExpr.expr.eval();
            case And andExpr -> andExpr.expr1.eval() && andExpr.expr2.eval();
            case Or orExpr -> orExpr.expr1.eval() || orExpr.expr2.eval();
        }; // liste exhaustive, pas besoin de cas default !
    }

    final class Var implements BoolExpr {
        private final boolean value;
        public Var(boolean value) { this.value = value; }
    }
    final class Not implements BoolExpr {
        private final BoolExpr expr;
        public Not(BoolExpr expr) { this.expr = expr; }
    }
    final class And implements BoolExpr {
        private final BoolExpr expr1, expr2;
        public And(BoolExpr expr1, BoolExpr expr2) { this.expr1 = expr1; this.expr2 = expr2; }
    }
    final class Or implements BoolExpr {
        private final BoolExpr expr1, expr2;
        public Or(BoolExpr expr1, BoolExpr expr2) { this.expr1 = expr1; this.expr2 = expr2; }
    }
}
```

Override ou switch+pattern matching ?

2 styles opposés ont été présentés dans les exemples :

(1) méthode `eval` abstraite + redéfinition pour chaque sous-type (liaison dynamique)

(2) une seule définition de `eval`, dans `BoolExpr`, avec un gros **`switch`** [1]

Quand choisir quel style ?

1. Voire un **`if`**/**`else if`**/.../**`else`**... avec **`instanceof`** ce qui est moins bien.
2. On retombe dans le cas par défaut qui consiste souvent à lever une exception.

Override ou switch+pattern matching ?

2 styles opposés ont été présentés dans les exemples :

(1) méthode `eval` abstraite + redéfinition pour chaque sous-type (liaison dynamique)

(2) une seule définition de `eval`, dans `BoolExpr`, avec un gros **`switch`** [1]

Quand choisir quel style ? Généralement, (1) est conseillé :

- (2) → risque d'oubli de traiter le cas d'un nouveau sous-type qu'on ajouterait. [2]
- Dans (1), chaque sous-type ajouté s'occupe de sa propre implémentation → architecture extensible (y compris par autre développeur).

1. Voire un **`if`**/**`else if`**/.../**`else`**... avec **`instanceof`** ce qui est moins bien.
2. On retombe dans le cas par défaut qui consiste souvent à lever une exception.

Override ou switch+pattern matching ?

2 styles opposés ont été présentés dans les exemples :

(1) méthode `eval` abstraite + redéfinition pour chaque sous-type (liaison dynamique)

(2) une seule définition de `eval`, dans `BoolExpr`, avec un gros **`switch`** [1]

Quand choisir quel style ? Généralement, (1) est conseillé :

- (2) → risque d'oubli de traiter le cas d'un nouveau sous-type qu'on ajouterait. [2]
- Dans (1), chaque sous-type ajouté s'occupe de sa propre implémentation → architecture extensible (y compris par autre développeur).

Mais pour une hiérarchie scellée, (2) se défend :

- Exhaustivité du **`switch`** vérifiée par compilateur. De plus, seul le développeur de `BoolExpr` peut ajouter un autre sous-type. Donc (2) n'a pas d'inconvénient !
- (2) a l'avantage de présenter toute la logique a un emplacement unique.

1. Voire un **`if`**/**`else if`**/.../**`else`**... avec **`instanceof`** ce qui est moins bien.
2. On retombe dans le cas par défaut qui consiste souvent à lever une exception.

Types finis

Pour quoi faire ?

Un **type fini** est un type ayant un ensemble fini d'instances, toutes définies statiquement dès l'écriture du type, sans possibilité d'en créer de nouvelles lors de l'exécution. [1]

Certaines variables ont, en effet, une valeur qui doit rester dans un ensemble fini, prédéfini :

- les 7 jours de la semaine
- les 4 points cardinaux
- les 3 (ou 4 ou plus) états de la matière
- les *n* états d'un automate fini (dans protocole ou processus industriel, par exemple)
- les 3 mousquetaires, les 7 nains, les 9 nazgûls...

→ Situation intéressante car, théoriquement, nombre fini de cas à tester/vérifier.

1. C'est donc un type scellé (très contraint) : clairement, si un type n'est pas scellé, il ne peut pas être fini.

Pourquoi définir un type fini plutôt que réutiliser un type existant ?

- Typiquement, types de Java trop grands [1]. Si utilisés pour représenter un ensemble fini, difficile voire impossible de prouver que les variables ne prennent pas des valeurs absurdes.
- Même si on l'a prouvé sur papier, le programe peut comporter des typos (ex : `"lnudi"` au lieu de `"lundi"`), que le compilateur ne les verra pas.

Avec un type fini, le compilateur garantit que la variable reste dans le bon ensemble. [2]

1. Soit très grands (p. ex., il y a 2^{32} **`int`**s), soit quasi-infinis (il ne peut pas exister plus de 2^{32} références en même temps, mais à l'exécution, un objet peut être détruit et un autre recréé à la même adresse...).
2. Il pourrait aussi théoriquement vérifier l'exhaustivité des cas d'un **`switch`** (sans **`default`**) ou d'un **`if`** / **`else if`** (sans **`else`** seul) : ça existe dans d'autres langages, mais `javac` ne le fait pas [3]. Intérêt : éviter des **`default`** et des **`else`** que l'on sait inatteignables.

Écrire un type fini

Comment faire ?

- **Mauvaise idée :** réserver un nombre fini de constantes dans un type existant (ça ne résout pas les problèmes évoqués précédemment).

  Remarque : c'est ce que fait la construction **enum** du langage C. Les constantes déclarées sont en effet des **int**, et le type créé est un *alias* de **int**.
- On a déjà vu qu'il fallait créer un nouveau type.
- Il faut qu'il soit impossible d'en créer des instances en dehors de sa déclaration...
- ... qu'elles soient directes (appel de son constructeur) ou indirectes (via extension).
- **Bonne idée :** implémenter le type fini comme classe à constructeurs privés et créer les instances du type fini comme constantes statiques de la classe :

```
public class Piece { // peut être final... mais le constructeur privé suffit
    private Piece() {}
    public static final Piece PILE = new Piece(), FACE = new Piece();
}
```

→ les **enum** de Java sont du sucre syntaxique pour écrire cela (+ méthodes utiles).

Classes d'énumération

```
public enum ETAT { SOLIDE, LIQUIDE, GAZ, PLASMA }
```

Une **classe d'énumération** (ou juste énumération) est une classe particulière, déclarée par un bloc synaxique **`enum`**, dans lequel est donnée la liste (exhaustive et définitive) des instances (= « constantes » de l'**`enum`**.).

Elle définit un **type énuméré**, qui est un type :

- fini : c'est la raison d'être de cette construction ;
- pour lequel l'opérateur « == » teste bien l'égalité sémantique [1]
  (toutes les instances représentent des valeurs différentes) ;
- utilisable en argument d'un bloc **`switch`** ;
- et dont l'ensemble des instances s'itère facilement :
  `for (MonEnum val: MonEnum.values()){...}`

1. Pour les enums, identité et égalité sont synonymes.

Exemple simple :

```java
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}

class Test {
    public static void main(String[] args) {
        for (Day d : Day.values()) {
            // Remarque : cette syntaxe de switch a été introduite dans Java 12.
            // Si vous étiez habitué aux ":" avec des breaks, familiarisez-vous avec le
                 nouveau style !
            // (plus sûr : pas de risque d'oublier un break)
            switch (d) {
                case SUNDAY, SATURDAY -> System.out.println(d + ": sleep");
                default -> System.out.println(d + ": work");
            }
        }
    }
}
```

Énumération = classe particulière

Il est possible d'écrire une classe équivalente sans utiliser le mot-clé **enum** [1].

L'exemple précédent pourrait (presque [2]) s'écrire :

```
public final class Day extends Enum<Day> {
    public static final Day SUNDAY = new Day("SUNDAY", 0),
            MONDAY = new Day("MONDAY", 1), TUESDAY = new Day("TUESDAY", 2),
            WEDNESDAY = new Day("WEDNESDAY", 3), THURSDAY = new Day("THURSDAY", 4),
            FRIDAY = new Day("FRIDAY", 5), SATURDAY = new Day("SATURDAY", 6);

    private Day(String name, int ordinal) {
        super(name, ordinal);
    }

    // plus méthodes statiques valueOf() et values()
}
```

`Enum<E>` est la superclasse directe de toutes les classes déclarées avec un bloc **enum**. Elle contient les fonctionnalités communes à toutes les énumérations.

1. Puisque c'est du sucre syntaxique !
2. En réalité, ceci ne compile pas : `javac` n'autorise pas le programmeur à étendre la classe `Enum` à la main. Cela est réservé aux vraies **enum**. Si on voulait vraiment toutes les fonctionnalités des **enum**, il faudrait réécrire les méthodes de la classe `Enum`.

Énumérations

Extensions

On peut donc y ajouter des membres, en particulier des méthodes :

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;
    public boolean isWorkDay() {
        // le switch nouveau s'utilise aussi comme expression !
        return switch (this) {
            case SUNDAY, SATURDAY -> false;
            default -> true;
        }
    }
    public static void main(String[] args) {
        for (Day d : Day.values()) {
            System.out.println(d + ": " + (d.isWorkDay() ? "work" : "sleep"));
        }
    }
}
```

Énumérations

Extensions

On peut ajouter des constructeurs (privés seulement).
Chaque constante de l'enum doit être suivie des arguments de l'un des constructeurs.
Attention : si constructeur personnalisé, pas de constructeur par défaut !

```
public enum Day {
    // remarquer les constantes sans argument : c'est juste un raccourci pour ()
    SUNDAY(false), MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY(false);
    final boolean isWorkDay;
    private Day(boolean work) {
        isWorkDay = work;
    }
    private Day() { // constructeur sans paramètre -> permet de déclarer les constantes
         d'enum sans argument
        isWorkDay = true;
    }
    public static void main(String[] args) {
        for (Day d : Day.values()) {
            System.out.println(d + ": " + (d.isWorkDay ? "work" : "sleep"));
        }
    }
}
```

Énumérations

Extensions

Chaque déclaration de constante énumérée peut être suivie d'un corps de classe, afin d'ajouter des membres ou de redéfinir des méthodes juste pour cette constante.

```
public enum Day {
    SUNDAY { @Override public boolean isWorkDay() { return false; } },
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    SATURDAY { @Override public boolean isWorkDay() { return false; } };

    public boolean isWorkDay() { return true; }

    public static void main(String[] args) {
        for (Day d : Day.values())
            System.out.println(d + ": " + (d.isWorkDay() ? "work" : "sleep"));
    }
}
```

Dans ce cas, la constante est l'instance unique d'une sous-classe [1] de l'**`enum`**.

Remarque : comme d'habitude, toute construction basée sur des `@Override` et la liaison dynamique est à préférer à un **`switch`** (quand c'est possible et que ça a du sens).

Énumérations

Types énumérés et sous-typage

- Tous les types énumérés étendent la classe `Enum` [1].
  Donc une énumération ne peut étendre aucune autre classe.
- En revanche rien n'interdit d'écrire `enum Truc implements Machin { ... }`.
- Les types enum sont des classes à constructeur(s) privé(s). [2]
  Ainsi aucune instance autre que les constantes déclarées dans le bloc `enum` ne pourra jamais exister [3].
- On ne peut donc pas non plus étendre un type énuméré [4].

1. Version exacte : l'énumération `E` étend `Enum<E>`. Voir la généricité.
2. Elles sont mêmes `final` si aucune des constantes énumérées n'est muni d'un corps de classe.
3. Ainsi, toutes les instances d'une `enum` sont connues dès la compilation.
4. On ne peut pas l'étendre « à la main », mais des sous-classes (singletons) sont compilées pour les constantes de l'enum qui sont munies d'un corps de classe.

Énumérations

Méthodes

Toute énumération `E` a les méthodes d'instance suivantes, héritées de la classe `Enum` :

- `int compareTo(E o)` (de l'interface `Comparable`, implémentée par la classe `Enum`) : compare deux éléments de `E` (en fonction de leur ordre de déclaration).
- `String toString()` : retourne le nom de la constante (une chaîne dont le texte est le nom de l'identificateur de la constante d'enum)
- `int ordinal()` : retourne le numéro de la constante dans l'ordre de déclaration dans l'enum.

Par ailleurs, tout type énuméré `E` dispose des deux méthodes statiques suivantes :

- `static E valueOf(String name)` : retourne la constante d'enum dont l'identificateur est égal au contenu de la chaîne `name`
- `static E[] values()` : retourne un tableau contenant les constantes de l'enum dans l'ordre dans lequel elles ont été déclarées.

Énumérations

Quand les utiliser

- **Évidemment :** pour implémenter un type fini (cf. intro de ce cours). Remarquez au passage toutes les erreurs potentielles si on utilisait, à la place d'une `enum` :
  - des `int` : tentation d'utiliser directement des littéraux numériques (`1`, `0`, `-42`) peu parlants au lieu des constantes (par flemme). Risque très fort d'utiliser ainsi des valeurs sans signification associée.
  - des `String` sous forme littérale : risque fort de faire une typo en tappant la chaîne entre guillemets.
- **Cas particulier :** quand une classe ne doit contenir qu'une seule instance (singleton) → le plus sûr pour garantir qu'une classe est un singleton c'est d'écrire une enum à 1 élément.

```
enum MaClasseSingleton /* insérer implements Machin */{
    INSTANCE; // <--- l'instance unique !
    /* insérer ici tous les membres utiles */
}
```

Tout cela peut être fait sans les `enum`s mais c'est fastidieux et risque d'être mal fait.

Énumérations

Bien les utiliser

- Les **`enum`**s sont bien pensés et robustes. Il est assez difficile de mal les utiliser.
- **Piège possible :** compter sur les ordinaux (**`int`** retourné par `ordinal()`) ou l'ordre relatif des constantes d'une **`enum`** $\rightarrow$ fragilité en cas de mise à jour de la dépendance fournissant l'**`enum`**.

Bonne pratique pour utiliser une enum fournie par un tiers : (EJ3 Item 35) ne compter ni sur le fait qu'une constante possède un ordinal donné, ni sur l'ordre relatif des ordinaux (= ordre des constantes dans tableau `values()`).

Énumérations et collections (1)

Il existe des implémentations de collections optimisées pour les énumérations.

- `EnumSet<E extends Enum<E>>`, qui implémente `Set<E>` : on représente un ensemble de valeurs de l'énumération `E` par un champ de bits (le bit n°*i* vaut 0 si la constante d'ordinal *i* est dans l'ensemble, 1 sinon). Cette représentation est très concise et très rapide.

  Création via méthodes statiques
  `Set<DAY> weekend = EnumSet.of(Day.SATURDAY, Day.SUNDAY)`, voire
  `Set<Day> week = EnumSet.allOf(Day.class)`.

  L'usage d'`EnumSet` est à préférer à l'usage direct des champs de bits [1] (EJ3 Item 36). On gagne en clarté et en sécurité.

1. Vous savez, ces entiers qu'on manipule bit à bit via les opérateurs <<, >>, |, & et ~ et dont les programmeurs en C sont si friands...

Énumérations et collections (2)

- `EnumMap<K extends Enum<K>, V>` qui implémente `Map<K,V>` : une `Map` représentée (en interne) par un tableau dont la case d'indice *i* référence la valeur dont la clé est la constante d'ordinal *i* de l'enum `K`.

  Construire un `EnumMap` :
  `Map<Day, Activite> edt = new EnumMap<>(Day.class);`.

  `EnumMap` est à préférer à tout tableau ou toute liste où l'on utiliserait les ordinaux des constantes d'une **`enum`** en tant qu'indices (EJ3 Item 37).

Énumérations

Exemple du type algébrique, continué avec enum

Ajoutons un nouveau cas pour les 2 constantes Vrai et Faux :

```
public sealed interface BoolExpr {
    boolean eval();
    enum Cst implements BoolExpr { // cas supplémentaire : les constantes Vrai/Faux
        TRUE {@Override public boolean eval() { return true; } },
        FALSE {@Override public boolean eval() {return false; }}
    } // pas de default nécessaire, car les cas du switch sont exhaustifs
    final class Var implements BoolExpr {
        private boolean value;
        public Var(boolean value) { this.value = value; }
        public void set(boolean newVal) { this.value = newVal; }
        @Override public boolean eval() { return value; }
    }
    final class Not implements BoolExpr {
        private final BoolExpr expr;
        public Not(BoolExpr expr) { this.expr = expr; }
        @Override public boolean eval() { return !expr.eval(); }
    }
   final class And implements BoolExpr {
        private final BoolExpr expr1, expr2;
        public And(BoolExpr expr1, BoolExpr expr2) { this.expr1 = expr1; this.expr2 = expr2; }
        @Override public boolean eval() { return expr1.eval() && expr2.eval(); }
    }
    final class Or implements BoolExpr {
        private final BoolExpr expr1, expr2;
        public Or(BoolExpr expr1, BoolExpr expr2) { this.expr1 = expr1; this.expr2 = expr2; }
        @Override public boolean eval() { return expr1.eval() || expr2.eval(); }
    }
}
```

Énumérations

Exemple du type algébrique, continué avec enum et switch expression exhaustive


```java
public sealed interface BoolExpr {
    boolean eval();
    enum Cst implements BoolExpr {
        TRUE, FALSE;
        @Override public boolean eval() {
            return  switch(this) { // un switch expression, pourquoi pas ?
                case TRUE -> true;
                case FALSE -> false;
            }; // le cas default n'est pas nécessaire (liste exhaustive des cas de l'enum);
        }
    }
    final class Var implements BoolExpr {
        private boolean value;
        public Var(boolean value) { this.value = value; }
        public void set(boolean newVal) { this.value = newVal; }
        @Override public boolean eval() { return value; }
    }
    final class Not implements BoolExpr {
        private final BoolExpr expr;
        public Not(BoolExpr expr) { this.expr = expr; }
        @Override public boolean eval() { return !expr.eval(); }
    }
    // et ainsi de suite pour And et Or...
}
```

Remarque : cette hiérarchie scellée est valide car cette **enum** est implicitement **final** .

Enregistrements (records)

Un peu sur le modèle des `enum`, Java 16 a introduit le concept de **`record`** :

```
record Complex(double real, double imaginary) { }
```

Ceci est du sucre syntaxique pour :

```
final class Complex extends Record { // Record, classe réservée (comme Enum)
    private final double real, imaginary;
    public Complex(double real, double imaginary) {
        this.real = real; this.imaginary = imaginary;
    }
    public double real() { return real; }
    public double imaginary() { return imaginary; }
    public String toString() {  // "Complex[real=...,imaginary=...]"
        return "Complex[real=" + real + ", imaginary=" + imaginary + "]";
    }
    public String equals(Object other) {
        if (other instanceof Complex c)
        return real == c.real && imaginary == c.imaginary;
        else return false;
    }
    // + redéfinition de hashCode() : calculé à  partir de real et imaginary
}
```

Enregistrements (records)

Comme les **`enum`**, les **`record`** :

- Ne peuvent pas hériter d'une autre classe (héritent de `Record`).
- Ne peuvent pas être hérités (**`final`**).

On peut aussi ajouter des membres (méthodes, attributs, types imbriqués) et des constructeurs, mais pas de nouveaux attributs non statiques.

Les **`record`** sont ainsi **des types garantis immuables** [1].

1. Au premier niveau, c'est-à-dire que rien n'empêche les attributs de référencer des objets modifiables.

Enregistrements (records)

Exemple du type algébrique, continué


```java
public sealed interface BoolExpr {
    boolean eval();

    enum Cst implements BoolExpr {
        TRUE, FALSE;
        @Override public boolean eval() {
            return  switch(this) {
                case TRUE -> true;
                case FALSE -> false;
            };
        }
    }
    final class Var implements BoolExpr { // classe mutable, record pas possible
        private boolean value;
        public Var(boolean value) { this.value = value; }
        public void set(boolean newVal) { this.value = newVal; }
        @Override public boolean eval() { return value; }
    }
    // Tous les cas immuables convertis en record !
    record Not(BoolExpr expr) implements BoolExpr {
        @Override public boolean eval() { return !expr.eval(); }
    }
    record And(BoolExpr expr1, BoolExpr expr2) implements BoolExpr {
        @Override public boolean eval() { return expr1.eval() && expr2.eval(); }
    }
    record Or(BoolExpr expr1, BoolExpr expr2) implements BoolExpr {
        @Override public boolean eval() { return expr1.eval() || expr2.eval(); }
    }
}
```

Enregistrements (records)

Exemple du type algébrique, continué, version pattern matching (Java 19 preview)


```java
public sealed interface BoolExpr {
    default boolean eval() {
        return switch(this) {
            case Cst cstExpr -> switch(cstExpr) {
                case TRUE -> true;
                case FALSE -> false;
            };
            case Var varExpr -> varExpr.value;
            case Not(BoolExpr expr) -> !expr.eval();
            case And(BoolExpr expr1, BoolExpr expr2) -> expr1.eval() && expr2.eval();
            case Or(BoolExpr expr1, BoolExpr expr2) -> expr1.eval() || expr2.eval();
        };
    }

    enum Cst implements BoolExpr { TRUE, FALSE }

    final class Var implements BoolExpr {
        private boolean value;
        public Var(boolean value) { this.value = value; }
        public void set(boolean newVal) { this.value = newVal; }
    }
    // Tous les cas immuables convertis en record !
    record Not(BoolExpr expr) implements BoolExpr { }
    record And(BoolExpr expr1, BoolExpr expr2) implements BoolExpr { }
    record Or(BoolExpr expr1, BoolExpr expr2) implements BoolExpr { }
}
```

Remarque : cette hiérarchie scellée est valide car les **enum** et **record** sont **final** .

On a coutûme de dire que :

- la classe B hérite de la classe A seulement si un B **est un** A.
- on compose [1] A dans B quand un B **possède un** A.

Mais « **est un** » peut être interprété de plusieurs façons :

- une instance de A **peut être utilisée à la place d'** une instance de B $\leftrightarrow$ sous-typage.
- une instance de A **est faite comme** une instance de B [2] $\leftrightarrow$ héritage.

(Comme l'héritage implique le sous-typage, la seconde interprétation est plus « forte ».)

1. C'est-à-dire qu'on met dans A un attribut d'instance de type B. On reparle de composition juste après.
2. Mêmes champs en mémoire, appel du constructeur parent ; même code sauf si redéfini.

Autour de l'héritage

Le bon usage de l'héritage de classe (2)

On peut donc envisager de déclarer une classe `B` sous-classe d'une classe `A` existante par une classe `B` lorsque :

- `B` doit pouvoir être utilisée à la place de `A`,
- l'implémentation de `B` semble pouvoir se baser sur celle de `A`,
- et `A` est faite telle que l'héritage est possible.
  C.-à-d. : ni `A` ni les méthodes à redéfinir ne sont **`final`** et le code à ajouter ou modifier est en accord avec les instructions données dans la documentation de `A`. [1]

Mais...

1. Faute de documentation, évitez les constructions fragiles, comme par exemple, appeler une méthode héritée `f` depuis une méthode redéfnie `g` de `B`. En effet : sauf indication contraire, `f` est susceptible d'appeler `g` → risque de `StackOverFlowError`.

... ce n'est pas parce qu'on peut le faire que c'est une bonne idée !

- Si la classe B hérite de A, elle récupère toutes les fonctionnalités héritées de A, y compris celles qui n'auraient pas de rapport avec l'objectif de B. [1]

  (C'est le principe-même du sous-typage. Mais la vraie question est : est-ce mon intention de créer un sous-type ? Cette classe sera-t-elle utilisée dans un contexte polymorphe ?)
- Les instances de B contiennent tous les champs de A (y compris privés), même devenus inutiles → « surpoids » et risque d'incohérences.
- Étendre une classe qui n'était pas conçue pour cela expose à des comportements inattendus [2] (non documentés par son auteur... qui n'avait pas prévu ça !).

1. Et si ce n'est pas le cas maintenant, quid de la prochaîne version de A ?
2. Cf. cas de la « classe de base fragile » vu précédemment.

Pour des objectifs simples, préférer des techniques alternatives :

- créer du sous-typage → implémentation d'interface [1] [2].
  Une interface ne craint pas le syndrôme de la « classe de base fragile ». [3]
- réutiliser des fonctionnalités déjà programmées → **composition** [4] [5] (utiliser un objet auxiliaire possédant les fonctionnalités voulues pour les ajouter à votre classe).
  Ici aussi, on ne risque pas de « perturber » des fonctionnalités déjà programmées.

1. Obligatoire et assumé dans des langages comme Rust, où l'héritage ne crée pas de sous-typage.
2. EJ3 20 : « *Prefer interfaces to abstract classes* »
3. Faux en cas de méthodes `default` → même besoin de documentation que pour l'héritage de classe.
4. EJ3 18 : « *Favor composition over inheritance* »
5. On a l'habitude de parler de composition dans ce contexte. Mais souvent, une agrégation simple peut rendre le même service.

Autour de l'héritage

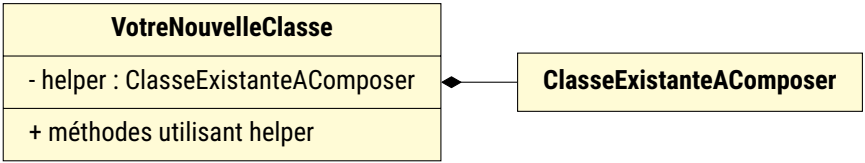
Alternative : ce qu'on entend par « composition »

Composition/Agrégation : utilisation d'un objet à l'intérieur d'un autre pour réutiliser les fonctionalités codées dans le premier. Exemple :

```
class Vendeur {
    private double marge;
    public Vendeur(double marge) { this.marge = marge; }
    public double vend(Bien b) { return b.getPrixRevient() * (1. + marge); }
}

class Boutique {
    private Vendeur vendeur;
    private final List<Bien> stock = new ArrayList<>();
    private double caisse = 0.;

    public Boutique(Vendeur vendeur) { this.vendeur = vendeur; }

    public void vend(Bien b) {
        if (stock.contains(b)) { stock.remove(b); caise += vendeur.vend(b); }
    }
}
```

`Boutique` réutilise des fonctionalités de `Vendeur` sans en être un sous-type.

Ces mêmes fonctionalités pourraient aussi être réutilisées par une autre classe `SuperMarche`.



Notez le losange plein.

UML distingue la composition de l'agrégation (losange vide). La différence est subtile :

- composition : l'objet du côté du losange est considéré comme propriétaire de l'autre objet, dont le cycle de vie est lié à celui du premier.
- agrégation : simple utilisation d'un objet par un autre sans que ce dernier ne soit propriétaire de l'autre.

En Java, la composition se traduit par l'absence de référence externe vers l'objet utilisé (le ramasse-miette peut le détruire dès que son propriétaire est détruit).

Objectif : ne pas hériter d'un « bagage » inutile

Exemple problématique

Mathématiquement les entiers sont des rationnels particuliers. Mais comment le coder ?

Pas terrible :

```
public class Rationnel {
    private final int numerateur, denominateur;
    public Rationnel(int p, int q) { numerateur = p; denominateur = q; }
    // + getteurs et opérations
}
public class Entier extends Rationnel { public Entier (int n) { super(n, 1); } }
```

Ici, toute instance d'entiers contient 2 champs (certes non visibles) : numérateur et dénominateur. Or 1 seul `int` aurait dû suffire.

→ utilisation trop importante de mémoire (pas très grave)

→ risque d'incohérence à cause de la redondance (plus grave)

Autre problème : la classe `Rationnel` visant à être immuable (attributs `final`) serait typiquement `final` (pour empêcher des sous classes avec attributs modifiables).

Objectif : ne pas hériter d'un « bagage » inutile

Modélisation à l'aide d'interfaces

Mieux :

```
public interface Rationnel { int getNumer(); int getDenom(); /* + opérations */ }
public interface Entier extends Rationnel {
    int getValeur();
    default int getNumer() { return getValeur(); }
    default int getDenom() { return 1; }
}
public final class RationnelImmuable implements Rationnel {
    private final int numerateur, denominateur;
    public RationnelImmuable(int p, int q) { numerateur = p; denominateur = q; }
    // + getteurs et opérations
}
public final class EntierImmuable implements Entier {
    private final intValue;
    public EntierImmuable (int n) { intValue = n; }
    @Override public int getValeur() { return intValue; }
}
```

Ainsi : types en version immuable (via les classes) et en version à mutabilité non précisée (via les interfaces), le tout sans trainer de « bagage » inutile.

Objectif : ne pas hériter d'un « bagage » inutile

Modélisation à l'aide d'interfaces... et de records !

Ces classes immuables sont moralement des records...

```
public interface Rationnel { int numerateur(); int denominateur(); /* + opér. */ }
public interface Entier extends Rationnel {
    int valeur();
    default int numerateur() { return valeur(); } // comme l'attribut du record
    default int denominateur() { return 1; } // idem
}
public record RationnelImmuable(int numerateur, int denominateur) implements
     Rationnel {
    // + opérations
}
public record EntierImmuable(int valeur) implements Entier { }
```

Inconvénient : getteurs automatiques de même nom que les attributs
$\rightarrow$ soit il faut même le nom pour les attribut des records et les méthodes des interfaces, soit il faut définir des synonymes.

Objectif : ne pas hériter d'un « bagage » inutile

Modélisation à l'aide de classes scellées

Dans la solution précédente, nous avons perdu le sous-typage entre les types immuables.

Cela peut encore être amélioré : en rendant privées les implémentations immuables et en scellant tous les types publics de cette hiérarchie.

```
public class Arithmetique {
    private Arithmetique() {} // classe—outil non instantiable
    // hiérarchie publique (classes à constructeurs privés —> scellées)
    public static abstract class Rationnel { private Rationnel() {} /* +get. abstraits */ }
    public static abstract class Entier extends Rationnel { private Entier() {}  /* + getteur abstrait */}
    // implémentations immuables privées
    private static final class RationnelImpl extends Rationnel {
        final int numerateur, denominateur;
        RationnelImpl(int p, int q) { numerateur = p; denominateur = q; }
        // + implémentations getteurs
    }
    private static final class EntierImpl extends Entier {
        final int valeur;
        EntierImpl(int valeur) { this.valeur = valeur; }
        // + implémentation getteur
    }
    // fabriques statiques
    public static Rationnel rationnel(int p, int q) { return new RationnelImpl(p, q); }
    public static Entier entier(int n) { return new EntierImpl(n); }
}
```

→ types publics avec bon sous-typage et scellage garantissant l'immuabilité.

Objectif : ne pas hériter d'un « bagage » inutile

Modélisation à l'aide de classes scellées

Que l'on peut maintenant écrire plus succinctement :

```
public class Arithmetique {
    private Arithmetique() {} // classe—outil non instantiable
    public sealed interface Rationnel { int denominateur(); int numerateur(); }
    public sealed interface Entier extends Rationnel { int valeur(); }
    private record RationnelImpl(int numerateur, int denominateur) implements Rationnel { }
    private record EntierImpl(int valeur) implements Entier {
        @Override public int numerateur() { return valeur;}
        @Override public int denominateur() { return 1;}
    }
    public static Rationnel rationnel(int p, int q) { return new RationnelImpl(p, q); }
    public static Entier entier(int n) { return new EntierImpl(n); }
}
```

Remarque : ici, la classe-outil reste nécessaire à l'encapsulation. En effet : les `record` ne peuvent pas étendre une classe, donc il fallait des `sealed interface` mais les interfaces n'acceptent pas les classes membre privées.

La seule solution était de co-imbriquer les interfaces et leurs implémentations.

... sinon il fallait faire sans `record` → classes abstraites scellées publiques avec implémentations par classes membres privées.

Objectif : ne pas laisser l'héritage casser l'encapsulation

Exemple problématique


```java
/**
*   ReelPositif représente un réel positif modifiable.
*   Contrat : getValeur et racine retournent toujours un réel positif.
*/
class ReelPositif {
    double valeur;
    public ReelPositif(double valeur) { setValeur(valeur) ; }
    public getValeur() { return valeur; } // on veut retour >= 0
    public void setValeur(double valeur) {
        if (valeur < 0) throw new IllegalArgumentException(); //crash
        this.valeur = valeur;
    }
    public double racine() { return Math.sqrt(valeur); }
}

class ReelPositifArrondi extends ReelPositif{
    public ReelPositifArrondi(double valeur) { super(valeur) ; }
    public void setValeur(double valeur) { this.valeur = Math.floor(valeur); }
}

public class Test {
    public static void main(String[] args) {
        ReelPositif x = new ReelArrondi(— Math.PI);
        System.out.println(x.racine()); // ouch! (affiche "NaN")
    }
}
```

Objectif : ne pas laisser l'héritage casser l'encapsulation

1. L'évidente : rendre `valeur` privé pour forcer l'accès via `getValeur` et `setValeur`.
   Point faible : ne résiste toujours pas à certaines extensions

```
class ReelPositifArrondi extends ReelPositif{
    double valeur2; // et hop, on remplace l'attribut de la superclasse
    public ReelPositifArrondi(double valeur) { this(valeur); }
    public void getValeur() { return valeur2; }
    public void setValeur(double valeur) { this.valeur2 = Math.floor(valeur); }
}
```

   Ici, `racine` peut toujours retourner `NaN`. Pourquoi ?

2. La solution la plus précise : rendre `valeur` privé **et** et passer `getValeur` en `final`. Cette solution garantit que le contrat sera respecté par toute classe dérivée.
   Point fort : on restreint le strict nécessaire pour assurer le contrat.
   Point faible : il faut réfléchir, sinon on a vite fait de manquer une faille.

Objectif : ne pas laisser l'héritage casser l'encapsulation

3. La sûre, simple mais rigide : `valeur` → **`private`**, `ReelPositif` → **`final`**.

   Point fort : sans faille et très facile

   Point faible : on ne peut pas créer de sous-classe `ReelPositifArrondi`, mais on peut contourner grâce à la composition (on perd le sous-typage) :

```
class ReelPositifArrondi {
    private ReelPositif valeur;
    public ReelPositifArrondi(double valeur) { this.valeur = new
        ReelPositif(Math.floor(valeur)); }
    public void getValeur() { return valeur.getValeur(); }
    public void setValeur(double valeur) {
        this.valeur.setValeur(Math.floor(valeur)); }
}
```

   Pour retrouver le polymorphisme : écrire une interface commune à implémenter (argument supplémentaire pour toujours programmer à l'interface).

   → on a alors mis en œuvre le patron de conception « **décorateur** » (GoF).

Objectif : ne pas laisser l'héritage casser l'encapsulation

Le patron décorateur (1)

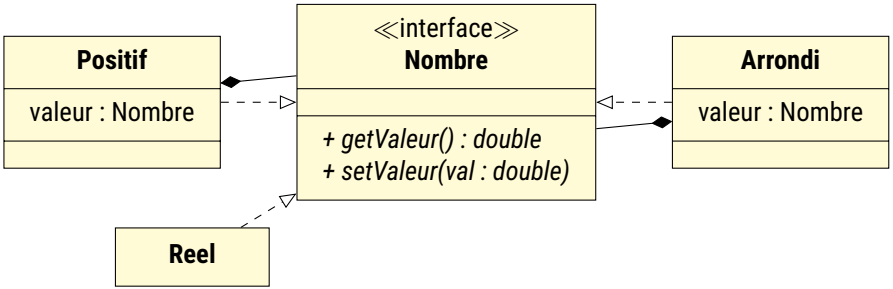

```java
interface Nombre {
    double getValeur();
    void setValeur(double valeur);
}

final class Reel implements Nombre {
    private double valeur;
    public Reel(double valeur) { this.valeur = valeur };
    @Override public double getValeur() { return valeur; }
    @Override public void setValeur(double valeur) { this.valeur = valeur; }
}

final class Arrondi implements Nombre {
    private final Nombre valeur;
    public Arrondi(Nombre valeur) { this.valeur = valeur; }
    @Override public double getValeur() { return Math.floor(valeur.getValeur()); }
    @Override public void setValeur(double valeur) { this.valeur.setValeur(valeur); }
}

final class Positif implements Nombre {
    private final Nombre valeur;
    public Positif(Nombre valeur) { this.valeur = valeur; }
    @Override public double getValeur() { return Math.abs(valeur.getValeur()); }
    @Override public void setValeur(double valeur) { this.valeur.setValeur(valeur); }
}
```

Objectif : ne pas laisser l'héritage casser l'encapsulation

Le patron décorateur (2)

Principe du patron décorateur : on implémente un type en utilisant/composant un objet qui est déjà instance de ce type, mais en lui ajoutant de nouvelles responsabilités.

L'intérêt : on peut décorer plusieurs fois un même objet avec des décorateurs différents.



Dans l'exemple, les décorateurs sont les classes `Positif` et `Arrondi`. Pour obtenir un réel positif arrondi, on écrit juste : `new Arrondi(new Positif(new Reel(42)))`. On n'a pas eu besoin de créer la classe `ReelPositifArrondi`.

Comment « hériter » de plusieurs classes ?

- Le patron décorateur permet, via la composition, d'ajouter/modifier plusieurs fois du comportement en réutilisant plusieurs clases existantes.
- Mais, ce patron est limité à créer des objets d'interface constante [1].
- Pour obtenir à la fois le bénéfice de la réutilisation d'implémentation et d'un type enrichi (plus de méthodes), il faut s'y prendre autrement.
- Le besoin décrit serait pourvu si la clause `extends` admettait plusieurs superclasses. Malheureusement, Java ne permet pas l'héritage multiple.
- À la place, il faut donc « bricoler » avec la composition et l'implémentation d'interfaces → **patron délégation** [2].

1. L'ajout de méthodes n'est pas une fonctionalité de ce patron de conception : en effet, seules les méthodes ajoutées par le dernier décorateur seront utilisables dans l'objet final.
2. Patron décrit et nommé par les auteurs du langage Kotlin, pas par le « Gang of Four », bien qu'il ressemble à d'autres patrons comme décorateur ou adaptateur.

Objectif : simuler un héritage multiple

Via les interfaces et la composition → patron « délégation »

Supposons que vous ayez 2 interfaces avec leurs implémentations respectives : [1]

```
interface AvecPropA { void setA(int newA); int getA(); }
interface AvecPropB { void setB(int newB); int getB(); }

class PossedePropA implements AvecPropA { int a; /* +methodes setA et getA... */ }
class PossedePropB implements AvecPropB { int b; /* +methodes setB et getB... */ }
```

On peut alors écrire une classe ayant les 2 propriétés de la façon suivante :

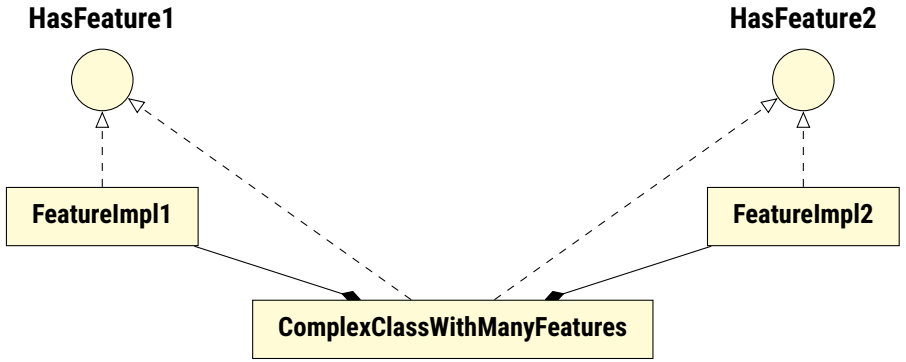
```
class PossedePropAetB implements AvecPropA, AvecPropB {
    PossedePropA aProxy; PossedePropB bProxy;
    void setA(int newA) { aProxy.setA(newA); }
    int getA() { return aProxy.getA(); }
    void setB(int newB) { bProxy.setB(newB); }
    int getB() { return bProxy.getB(); }
}
```

Si les classes auxiliaires sont fournies par un tiers et n'implémentent pas d'interface, on peut créer les interfaces manquantes et les implémenter dans `PossedePropAetB`. [2]

1. Supposées moins triviales que dans l'exemple (sinon c'est le marteau-pilon pour écraser une mouche !).
2. On revient au patron adaptateur déjà introduit dans ce cours.

Patron « délégation » en UML

Diagramme à 2 interfaces déléguées (mais ça pourrait être 1, 3 ou autant qu'on veut) :



Les implémentations dans `ComplexClassWithManyFeatures` des méthodes de `HasFeatureX` consistent en un simple appel vers la méthode de `FeatureImplX` de même nom.

Généricité

Exemple (de l'API) :

```
public interface Comparator<T> {
  public int compare(T o1, T o2);
  public boolean equals(Object o);
}
```

→ Que veut dire ce « <T> » ?
→ `Comparator` est une interface **générique** : un type paramétrable par un autre type. [1]

Types génériques du JDK :

- Les collections de Java ≥ 5 (interfaces et classes génériques).
  Ce fait seul suffit à justifier l'intérêt des génériques et leur introduction dans Java 5.
- Les interfaces fonctionnelles de Java ≥ 8 (pour les lambda expressions).
- `Optionnal`, `Stream`, `Future`, `CompletableFuture`, `ForkJoinTask`, ...

1. Ou « constructeur de type ». Mais cette terminologie est rarement utilisée en Java.

Généricité

Pourquoi ? (1)

La généricité est un procédé permettant d'augmenter la réutilisabilité du code de façon maîtrisée [1] grâce à des relations fines entre les types utilisés.

Sur un exemple :

```
class Boite { // non polymorphe
  public int x;
  void echange(Boite autre) {
    int ech = x; x = autre.x; autre.x = ech;
  }
}
```

Inconvénient : définition qui ne marche que pour les boîtes à entiers.

Réutilisabilité : proche de zéro !

1. Par opposition au polymorphisme par sous-typage, où, par exemple, pour les arguments d'appel de méthode, tout sous-type fait l'affaire indépendamment des autres types utilisés en argument.

Généricité

Pourquoi ? (2)

Première solution : boîte universelle (polymorphisme par sous-typage)

```
class Boite { // très (trop ?) polymorphe
  public Object x; // contient des Object, supertype de tous les objets
  void echange(Boite autre) {
    Object ech = x; x = autre.x; autre.x = ech;
  }
}
```

Réutilisabilité : semble totale (on peut tout mettre dans la boîte).

Inconvénient : on ne sait pas (avant l'exécution [1]) quel type contient une telle boîte → difficile d'utiliser la valeur stockée (il faut tester et *caster*).

1. En fait, programmer des classes comme cette version de `Boite` revient à abandonner le bénéfice du typage statique (pourtant une des forces de Java).

Cas d'utilisation problématique :

```
Boite b1 = new Boite(), b2 = new Boite(); b1.x = 6; b2.x = "toto";
System.out.println(7 * (Integer) b1.x); // <- là  c'est ok
b1.echange(b2);
System.out.println(7 * (Integer) b1.x); // <- ClassCastException !!
```

En fait on aurait dû tester le type à l'exécution :

```
if (b.x instanceof Integer) System.out.println(7 * (Integer) b.x);
```

... mais on préfèrerait vraiment que le code soit garanti par le compilateur [1].

1. Remarque : dans cet exemple, probablement l'IDE (à défaut de javac) signalera que la conversion est hasardeuse.
Normalement, on aura donc pensé à mettre `instanceof`. Il n'en reste pas moins que c'est un test à l'exécution qu'on aimerait éviter (en plus d'être une lourdeur à l'écriture du programme).

La bonne solution : boîte générique (→ **polymorphisme générique**)

```
class Boite<C> {
    public C x;
    void echange(Boite<C> autre) { C ech = x; x = autre.x; autre.x = ech; }
}
... // plus loin :
        Boite<Integer> b1 = new Boite<>(); Boite<String> b2 = new Boite<>();
        b1.x = 6; b2.x = "toto";
        System.out.println(7 * b1.x); // <- là  c'est toujours ok (et sans cast, SVP !)
        // b1.echange(b2); // <- ici erreur à  la compilation ! (ouf !)
        System.out.println(7 * b1.x);
```

La généricité consiste à introduire des types dépendants d'un paramètre de type.

La concrétisation du paramètre est vérifiée dès dès de la compilation [1] et uniquement à la compilation. Celle-ci est oubliée aussitôt [2] (**effacement de type** / *type erasure*).

1. Or le plus tôt on détecte une erreur, le mieux c'est !
2. Conséquence : les objets de classe générique ne savent pas avec quel paramètre ils ont été instanciés.

Types génériques et types paramétrés

Usage de base (1)

- **Type générique** : type (classe ou interface) dont la définition fait intervenir un **paramètre de type** (dans les exemples, c'était `T` et `C`).
- **À la définition** du type générique, le paramètre introduit dans son en-tête peut ensuite être utilisé dans son corps comme si c'était un vrai nom de type.

```
class Triplet<T,U,V> {          // introduction de T, U et V
    T elt1; U elt2; V elt3;       // utilisation de T, U et V
    public Triplet(T e1, U e2, V e3) { elt1 = e1; elt2 = e2; elt3 = e3; }
}
```

Attention : `T`, `U`, `V`, ne sont utilisables qu'en contexte non statique : en effet, ils représentent des types choisis pour chaque instance de `Triplet` ⇒ il faut donc être dans le contexte d'une instance pour qu'ils aient du sens.

Types génériques et types paramétrés

Usage de base (2)

- **À l'usage**, le type générique sert de constructeur de type : on remplace le paramètre par un type concret et on obtient un **type paramétré**.

  Exemple : `List` est un type générique, `List<String>` un des types paramétrés que `List` permet de construire.

- Le type concret substituant le paramètre doit être un type **référence** : ~~`Triplet<int, boolean, char>`~~ est interdit [1] !

1. Pour l'instant. Il semble qu'il soit prévu de permettre cela dans une prochaine version de Java.

Types génériques et types paramétrés

Usage de base (3)

- Utilisation de classe générique par instanciation directe :

```java
// à  partir de Java 5 :
Triplet<String, String, Integer> t1 =
    new Triplet<String, String, Integer>("Marcel", "Durand", 23);

// à  partir de Java 7 :
Triplet<String, String, Integer> t2 = new Triplet<>("Marcel", "Durand", 23);

// à  partir de Java 10 (si t3 est une variable locale) :
var t3 = new Triplet<String, String, Integer>("Marcel", "Durand", 23);
```

Le type de `t1`, `t2` et `t3` est le type paramétré `Triplet<String, String, Integer>`.

Types génériques et types paramétrés

Usage de base (4)

- Utilisation de classe générique par extension non générique (**spécialisation**) :

```
class TroisChars extends Triplet<Char, Char, Char> {
  public TroisChars(Char x, Char y, Char z) { super(x,y,z); }
}
```

`TroisChars` étend la classe paramétrée `Triplet<Char, Char, Char>`.

- Variante, spécialisation partielle :

```
class DeuxCharsEtAutre<T> extends Triplet<Char, Char, T> {
  public DeuxCharsEtAutre(Char x, Char y, T z) { super(x,y,z); }
}
```

La classe générique `DeuxCharsEtAutre<T>` étend la classe générique partiellement paramétrée `Triplet<Char, Char, T>`.

Parallèle entre type générique et méthode

La déclaration et l'utilisation des types génériques rappellent celles des méthodes.

Similitudes :

- introduction des paramètres (de type ou de valeur) dans l'en-tête de la déclaration ;
- utilisation des noms des paramètres dans le corps de la déclaration seulement ;
- pour utiliser le type générique ou appeler la méthode, on passe des concrétisations des paramètres.

Principales différences :

- Les paramètres des génériques représentent des types alors que ceux des méthodes représentent des valeurs.
- Pour les paramètres de type, le « remplacement » [1] a lieu à la compilation.
  Pour les paramètres des méthodes, remplacement par une valeur à l'exécution.

1. Rappel : remplacement oublié, effacé, aussitôt que la vérification du bon typage a été faite.

Types bruts

- Un nom de type générique seul, sans paramètre (comme « `Triplet` »), est aussi un type légal, appelé un **type brut** (***raw type***).
  Son utilisation est **fortement déconseillée**, mais elle est permise pour assurer la compatibilité ascendante [1].
- Un type brut est supertype direct [2] de tout type paramétré correspondant (ex : `Triplet` est supertype direct de `Triplet<Number, Object, String>`).
- Pour faciliter l'écriture, le *downcast* implicite [3] est malgré tout possible :

```
List l1 = new ArrayList(); // déclaration de l1 avec raw type
List<Integer> l2 = l1; // downcast implicite de l1 vers type paramétré
```

compile avec l'avertissement `unchecked conversion` sur la deuxième ligne.

1. Un source Java < 5 compile avec `javac` $\geq$ 5. Or certains types sont devenus génériques entre temps.
2. C'est une des règles de sous-typage relatives aux génériques, omises dans le début de ce cours.
3. Je crois que c'est l'unique occurrence de *downcast* implicite en Java.

Déclaration de paramètre de type niveau méthode

- Il est aussi possible d'introduire un paramètre de type dans la signature d'une méthode (possible aussi dans une classe non générique) :

```
static <E> List<E> inverseListe(List<E> l) { ... ; E x = get(0); ...; }
```

- Dans l'exemple ci-dessus, on garantit que la liste retournée par `inverseListe()` a le même type d'éléments que celle donnée en paramètre.
- **Usages possibles :**
  - contraindre plusieurs types apparaissant dans la signature de la méthode à être le même type, sans pour autant dire lequel ;
  - introduire localement un nom de type utilisable dans le corps de la méthode (type non défini, mais dont les contraintes sont connues, ex : type intersection, voir plus loin).

Remarque : il est donc possible d'écrire une méthode statique générique et son corps (contexte statique) pourra utiliser le paramètre introduit, contrairement aux paramètres de type introduits au niveau de la classe ou de l'interface.

Bornes de paramètres de type (1)

- Pour limiter les concrétisations autorisées, un paramètre de type admet des **bornes supérieures** [1] (se comportant comme supertypes du paramètre) :

```
class Calculator<Data extends Number>
```

Ici, `Data` devra être concrétisé par un sous-type de `Number` : une instance de `Calculator` travaillera sur nécessairement avec un certain sous-type de `Number`, celui choisi à son instanciation.

1. On verra dans la suite que les bornes inférieures existes aussi, mais elles ne s'appliquent qu'aux *wildcards* (et non aux paramètres de type).

Bornes de paramètres de type (2)

- Pour définir des bornes supérieures multiples (p. ex. pour implémenter de multiples interfaces), les supertypes sont séparés par le symbole « & » :

```
class RunWithPriorityList<T extends Comparable<T> & Runnable> implements List<T>
```

« `Comparable<T> & Runnable` » est un **type intersection**[1], il est sous-type direct de `Comparable<T>` et de `Runnable`.

Ainsi, `T` est sous-type de l'intersection (et donc de de `Comparable<T>` et de `Runnable`).

1. Remarque : c'est le seul contexte où on peut écrire un type intersection (type non dénotable). Ainsi, il n'est pas possible de déclarer explicitement une variable de type intersection. Implicitement, à l'aide d'une méthode générique et du mot-clé **`var`**, cela est cependant possible :
```
public static <T extends A & B > T intersectionFactory(...){ ... }
```
plus loin :
```
var x = intersectionFactory(...); //x est de type A & B
```
La technique assez « tirée par les cheveux » et d'utilité toute relative... .

Bornes de paramètres de type (3)

On peut prolonger l'analogie avec les méthodes et leurs paramètres : en effet, les paramètres des méthodes sont eux-mêmes « bornés » par les types déclarés dans la signature.