

Compléments en Programmation Orientée Objet

Aldric Degorre

Version 2022.01.01 du 28 septembre 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 Bigeon¹ (jeudi 16h15), Wael Boutglay² (mercredi 16h15), Wieslaw Zielonka³ (mardi 8h30, jeudi 10h45), Aldric Degorre⁴ (mardi 14h00)

1. bigeon@irif.fr

2. boutglay@irif.fr

3. zielonka@irif.fr

4. adegorre@irif.fr

- Utilisez les machines de TP de l'UFR...
- ... ou bien votre portable avec
 - le JDK 17 ou plus
 - votre IDE favori¹ (Eclipse, IntelliJ, NetBeans, VSCode, vi, emacs...)

Si vous utilisez les machines de l'UFR, assurez-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).

En fait, pouvez avoir besoin de 2 comptes :

- le compte U-Paris pour accéder à moodle¹ (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>

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

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

- 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.

- Vous connaissez la syntaxe de Java.
- Vous savez même écrire des programmes¹ 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 PL4, n'est-ce pas ?
- Vous pensez savoir programmer ... **vraiment ?**

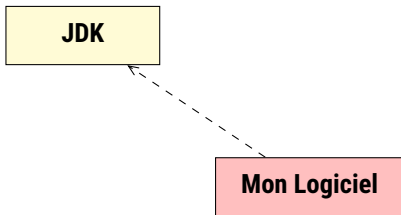
1. même dans le style objet!

Sauriez-vous encore ?

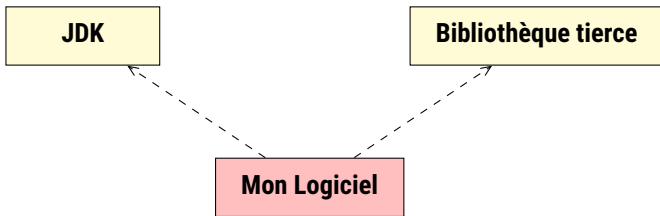
- Supprimer ce fameux bug que vous n'aviez pas eu le temps de corriger avant la deadline de PI4 l'année dernière ?
- Remplacer une des dépendances de votre projet par une nouvelle bibliothèque, sans tout modifier et ajouter 42 nouveaux bugs ?
- Ajouter une extension que vous n'aviez pas non plus eu le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

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

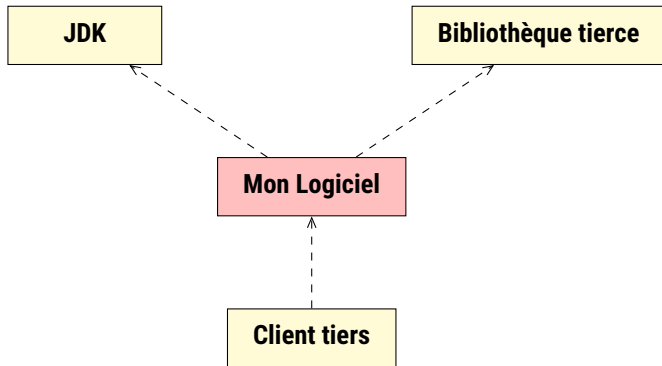
(Ou plus prosaïquement, de le publier sur GitHub pour y faire participer la communauté ?)



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



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



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

1. mais rarement en L2...

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 compatibles¹**
- **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* ?

→ 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 pratiques²

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.

Objectif : explorer les concepts de la programmation orientée objet (P00) au travers du langage Java et enseigner les principes d'une programmation fiable, pérenne et évolutive.

Contexte :

- Ce cours fait suite à P00-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 P00 (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 → économie du ré-apprentissage d'une syntaxe → 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 langages³ 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!

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 (1967¹), Smalltalk (1980²).
- 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 a commencé en 1971.

- **Principe de la P00** : des messages¹ s'échangent entre des objets qui les traitent pour faire progresser le programme.
- → **P00 = 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 objet² caché au monde extérieur → **encapsulation**.

Pour résumer : la P00 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 P00 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.

P00 → (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 fonctionnelle).

- **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 2^{er} ou 3^{ème} langage le plus populaire. ³

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

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

Versions « récentes » de Java :

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

Ce cours utilise Java 17, mais :

- la plupart de ce qui y est dit vaut aussi pour les versions antérieures;
- on ne s'interdit pas de s'interdire certaines nouveautés pour voir comment on aurait 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 » (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 C¹,
 - 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 éclipsées¹ par Flash² puis Javascript.
- Java n'est pas adapté à la programmation système³.
→ C, C++ et Rust plus adaptés.
- Idem pour le temps réel⁴.
- Idem pour l'embarqué⁵ (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.

- 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 commentaires¹ 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".

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

Langage de programmation → 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).

Habitudes dictées par :

- ❶ le compilateur (la syntaxe de Java¹)
 - ❷ le guide² de style qui a été publié par Sun en même temps que le langage Java (→ conventions à vocation universelle pour tout programmeur Java)
 - ❸ les directives de son entreprise/organisation
 - ❹ 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.

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

- ... de classes, interfaces, énumérations et annotations¹ → `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 mots². Exemple : `com.masociete.bibliothequetruc`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`

- **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.

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!

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'impératif¹, si la méthode sert à effectuer une action avec effet de bord²
`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

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

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

- On limite le nombre de caractères par ligne de code. Raisons :
 - certains programmeurs préfèrent désactiver le retour à la ligne automatique¹ ;
 - 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 colonnes²). De nos jours (écrans larges, haute résolution), 100-120 est plus raisonnable³.
- 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 (→ 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 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 !

- **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.

- On essaye de privilégier les retours à la ligne en des points du programme “hauts” dans l’arbre syntaxique (→ minimise la taille de l’indentation).

P. ex., dans “ $(x + 2) * (3 - 9/2)$ ”, on préférera couper à côté de “*” →

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

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

- 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 responsibility principle*/principe de responsabilité unique.

- Pour une méthode, la taille est le nombre de lignes.
- Principe de responsabilité unique¹ : une méthode est censée effectuer une tâche précise et compréhensible.
→ 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 critère² : on ne peut pas bien comprendre une méthode si on ne peut pas la parcourir en un simple coup d'œil
→ faire en sorte qu'elle tienne en un écran (~ 30-40 lignes max.)
 - En général, suivre les directives du projet.

1. Oui, là aussi!

2. qui n'engage que moi!

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`).

- 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/consistant (avec vous-même... et avec l’organisation dans laquelle vous travaillez).

- 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).

- 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);
```

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.

- 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...

- 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.

- 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.

Objet java = entité...

- caractérisée par un enregistrement contigu de données typées (**attributs**¹)
- accessible via un pointeur² 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. \mapsto <code>Personne.class</code>
<code>int age</code>	42
<code>String nom</code>	réf. \mapsto chaîne <code>"Dupont"</code>
<code>String prenom</code>	réf. \mapsto chaîne <code>"Toto"</code>
...	
<code>boolean marie</code>	<code>true</code>

-
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.

À service égal¹, les objets-**Personne** pourraient aussi être représentés ainsi :

	classe :	↦ Personne.class
int age	42	
Ident ident		
...		
boolean marie	true	

	classe :	↦ Ident.class
String nom	↦ "Dupont"	
String prenom	↦ "Toto"	

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 !

- **Objet** → **graphe d'objet** = un certain nombre d'enregistrements, se référéncant les uns les autres, tels que tout est accessible depuis un enregistrement principal ¹.
- C'est donc un graphe orienté connexe dont les nœuds sont des enregistrements et les arcs les référencements par pointeur.
- Les informations stockées dans ce graphe permettent aux services (méthodes de l'enregistrement principal) prévus par le type (interface) de l'objet de fonctionner.

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.

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.

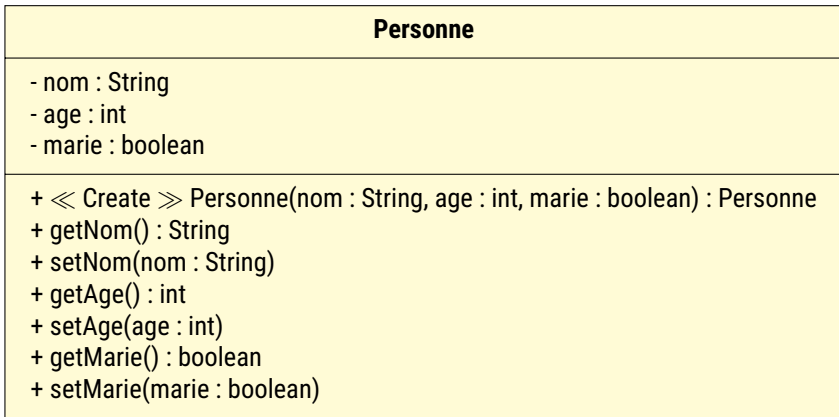
1. Dans le cas de Java. Dans d'autres langages comme C++, un objet tiers peut être carrément inclus dans l'objet principal. Quand c'est le cas, il s'agit nécessairement de composition et non pas d'une aggrégation simple.

- **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; }  
}
```



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.

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 instanciable¹
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`.

- 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 données² 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'occurrence : les attributs d'instance déclarés dans cette classe.

3. Ainsi, on note que le type défini par une classe est un type référence.

Constructeur : fonction¹ 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(... paramètres... );
```

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 défaut³.

-
1. Ou bien `super(params);` si utilisation d'un constructeur de la superclasse.
 2. Les attributs restent à leur valeur par défaut (0, `false` ou `null`), ou bien à celle donnée par leur initialiseur, s'il y en a un.
 3. Mais rien n'empêche d'écrire, en plus, à la main, un constructeur sans paramètre.

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

Plusieurs catégories de membres : attributs, méthodes et types membres².

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".

```
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;  
        }  
    }  
}
```

Contexte (associé à tout point du code source) :

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

Désigner un membre `m` déjà défini quelque part :

- écrire soit juste `m` (**nom simple**), soit `chemin.m` (**nom qualifié**)
- "`chemin`" donne le contexte auquel appartient le membre `m` :
 - pour un membre statique : la classe³ (ou interface ou autre...) où il est défini
 - pour un membre d'instance : une instance de la classe où il est défini
- "`chemin.`" est facultatif si `chemin ==` contexte local.

1. typiquement, remplacer "définition" par "corps de méthode"

2. L'objet qui, à cet endroit, serait référencé par `this`.

3. Et pour désigner une classe d'un autre paquetage : `chemin = paquetage.NomDeClasse`.

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 !

	statique (ou "de classe")	non statique (ou " d'instance ")
attribut	donnée <u>globale</u> ¹ , <u>commune</u> à <u>toutes les instances</u> de la classe.	donnée <u>propre</u> ² à <u>chaque instance</u> (nouvel exemplaire de cette variable alloué et initialisé à chaque instantiation).
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.

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); }  
}
```


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 redéfinition de `f` sera appelée. `f` est sujette à la liaison dynamique.

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'instances¹;
- obligation de retourner une instance directe de la classe.

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

```
public abstract class C { // ou bien interface
    ...
    // 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.

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 clients¹ (= en réduisant leur **couplage**) facilite son déboguage et son évolution future.²

- empêche les clients d'accéder à un objet de façon incorrecte ou non prévue. Ainsi,
 - la correction d'un programme est plus facile à vérifier (moins d'interactions à vérifier);
 - plus généralement, seuls les **invariants de classe**³ ne faisant pas intervenir d'attributs non privés peuvent être prouvés.

→ L'encapsulation rend donc aussi la classe plus fiable.

-
1. **Clients** d'une classe : les classes qui utilisent cette classe.
 2. En effet : on peut modifier la classe sans modifier ses clients.
 3. Différence avec l'item du dessus : les invariants de classe doivent rester vrais dans tout contexte d'utilisation de la classe, pas seulement dans le programme courant.

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

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

Pas bien :

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

Toute autre classe peut interférer en modifiant directement les valeurs de `a` ou `b`

→ on ne peut rien prouver à propos de `FiboGen`!

Bien :

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

Seule la méthode `next` peut modifier directement les valeurs de `a` ou `b`

→ s'il y a un bug, il est causé par l'exécution de `next`, pas de celle d'un code extérieur!

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

Pas bien :

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

Toute autre classe peut interférer en modifiant directement les valeurs de `a` ou `b`

→ on ne peut rien prouver à propos de `FiboGen`!

Bien : (ou presque)

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

Seule la méthode `next` peut modifier directement les valeurs de `a` ou `b`

→ s'il y a un bug, il est causé par l'exécution de `next`, pas de celle d'un code extérieur!¹

1. Or un bug peut se manifester si on exécute `next` plusieurs fois simultanément (sur plusieurs *threads*).

- 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).

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			
<i>package-private</i>	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  
}
```

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

Ici, 2 niveaux seulement : **public** ou *package-private*.

Visibilité	paquetage	partout
<i>package-private</i>	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.

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

Ainsi il faut bien réfléchir à ce que l'on souhaite exposer.⁵

1. Application Programming Interface

2. cf. Javadoc

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 existant¹ → garantie faible.
- **protected** → de même et, en +, toute sous-classe, n'importe où, voit la définition.
- **Aucun niveau ne garantit la confidentialité des données.**

Constantes : lisibles directement dans le fichier **.class**.

Variables : lisibles, via réflexion, par tout programme s'exécutant sur la même JVM.

Si la sécurité importe : bloquer la réflexion².

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

1. Même à une dépendance tierce, même sans 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!

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

Syntaxe du fichier `module-info.java` :

```
module nom_du_module {  
    requires nom_d_un_module_dont_on_depends;  
    exports nom_d_un_package_defini_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`.

- Pour les classes publiques, il est recommandé¹ 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** à 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.

- 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 public¹.
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()`

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

```
public final class Person {  
  
    // propriété "age"  
  
    // attribut de base (qui doit rester positif)  
    private int age;  
  
    // getteur, accesseur en lecture  
    public int getAge() {  
        return age;  
    }  
  
    // setteur, écriture contrôlée  
    public void setAge(int a) {  
        if (a >= 0) age = a;  
    }  
}
```


Exemple : propriété en lecture seule avec évaluation paresseuse.

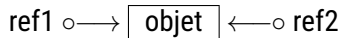
```
public final class Entier {  
    public Entier(int valeur) { this.valeur = valeur; }  
  
    private final int valeur;  
  
    // propriété ``diviseurs`` :  
    private List<Integer> diviseurs;  
  
    public List<Integer> getDiviseurs() {  
        if (diviseurs == null) diviseurs =  
            Collections.unmodifiableList(Utils.factorise(valeur)); // <- calcul  
            coûteux, à n'effectuer que si nécessaire  
        return diviseurs;  
    }  
}
```

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

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.

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?

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** = 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 immuable², 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, récurivement : on copie le graphe de l'objet³).

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 s'agit de savoir en quoi consiste le graphe de l'objet, sinon la notion de copie profonde reste ambiguë.

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

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

... 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 ≥ 14);
- plus généralement, toute classe¹ dont la documentation dit qu'elle l'est.

Les 8 types primitifs² se comportent aussi comme des types immuables³.

1. Voir `sealed interface` (Java ≥ 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.

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

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

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

1. Faisant intervenir un autre *thread*, cf. chapitre sur la programmation 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.

L'impossibilité d'*alias* extérieur au *frame*¹ d'une méthode est aussi intéressante, car elle autorise la JVM à optimiser en allouant l'objet directement en pile plutôt que dans le tas.

En effet : comme l'objet n'est pas référencé en dehors de l'appel courant, il peut être détruit sans risque au retour de la méthode.

La recherche de la possibilité qu'une exécution crée des *alias* externes (à une classe ou une méthode) s'appelle l'**escape analysis**².

1. *frame* = zone de mémoire dans la pile, dédiée au stockage des informations locales pour un appel de méthode donné.

2. Traduction proposée : **analyse d'échappement**?

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 :

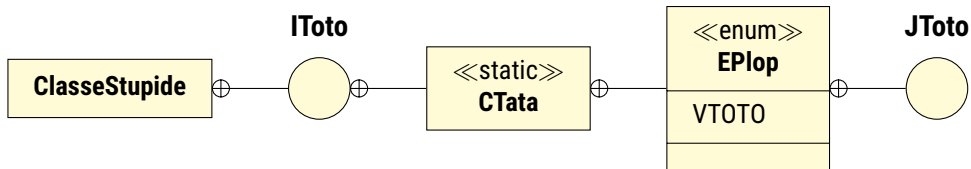
- dans 1., mauvaise idée si plusieurs des contextes partageant la référence pensent être les seuls à pouvoir modifier l'objet référencé;
- dans 2., risque de modifications concurrentes dans un programme *multi-thread* → précautions à prendre.

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



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

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)  
    }  
}
```

- 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...).

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.

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**<_>(_)".

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>`).

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).

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 locales¹).

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

La définition de classe est une expression dont la valeur est une instance¹ 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 */ ;  
    }  
}
```

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 \implies 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.

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.

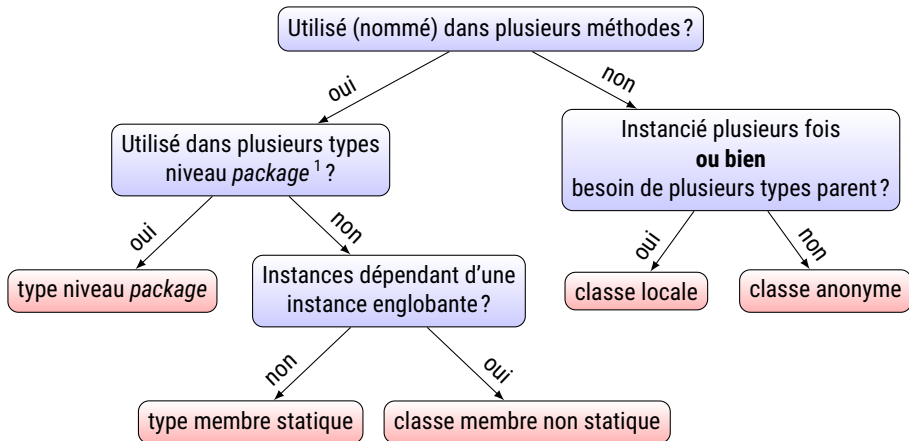
```
class/interface/enum TypeEnglobant {  
    static int x = 4;  
    static class/interface/enum TypeMembre { static int y = x; }  
    static int z = TypeMembre.y;  
}
```

Le contexte interne du type imbriqué contient toutes les définitions du contexte externe. Ainsi, sont accessibles directement (sans chemin¹) :

- 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 locales².

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...



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.*

- 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

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.

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.

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'objet².

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.

- Lors de l'appel d'une méthode¹ :
 - 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 méthode², le *program counter* du *thread* prend la valeur du pointeur de retour; le cas échéant³, 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).

- **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**.

- 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 différencié¹ dans code polymorphe).
- sous-typage, permettant le polymorphisme : une méthode déclarée pour argument de type **T** est callable 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)

Pour des raisons liées à la mémoire et au polymorphisme, 2 catégories¹ 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 (≤ 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.

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~~ = ~~"true"~~ ne compile pas, alors que **int** ~~String~~ = 12, oui!).

1. 1 **mot** = 32 bits

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.

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'identité¹ $x == z$ 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.

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

Ainsi :

- pour les types primitifs¹ → 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

- 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 spécifique² (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.

- 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 attribué un type ne sont pas les mêmes selon le moment où cette vérification est faite.

Typage statique → concerne les expressions du programme

Typage dynamique → concerne les données existant à l'exécution.

Où 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 ».

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**.

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.

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.)

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 instantiation.

- **Définition** : le type A est **sous-type** de B ($A <: B$) (ou bien B **supertype** de A ($B >: A$)) si toute entité¹ 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.

- **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 ensemble¹ : il est aussi muni d'opérations, d'une structure, de contrats², ...

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).

- **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.

Pourquoi le sous-typage structurel est insuffisant ?

Exemple :

- Dans le cours précédent, les instances de la classe *FiboGen* génèrent la suite de Fibonacci.
- Contrat possible¹ : « 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-classe²) de *Fibogen* dont la méthode *next* retournerait toujours 0.
→ 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.

- **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.).

→ 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.

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**).

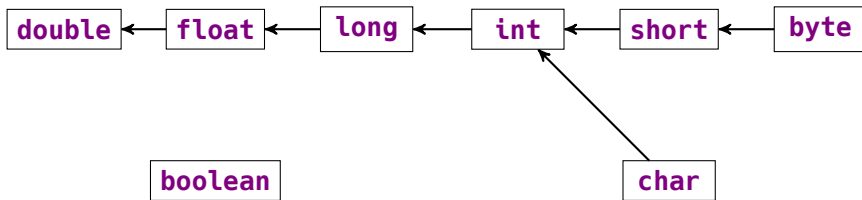
Mais la définition de A ne passe la compilation que si certaines contraintes structurelles¹ sont vérifiées, concernant les redéfinitions de méthodes.

- 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.

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.

Types référence :

$A <: B$ ssi B est `Object`¹ ou s'il existe des types $A_0(=A), A_1, \dots, A_n(=B)$ 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 $a <: b$;

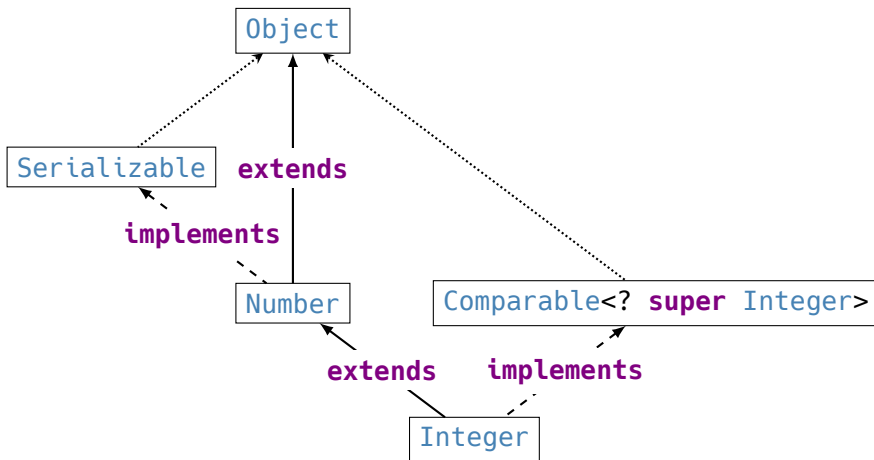
1. C'est vrai même si A est une interface, alors même qu'aucune interface n'hérite de `Object`.

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.

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 ambiguë²).

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.

Pourquoi ce remplacement ne gêne pas l'exécution :

- les objets sont utilisables sans modification comme instances de tous leurs supertypes¹ (**sous-typage inclusif**). P. ex. : `Object o = "toto"` fonctionne.
- Java² 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.

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* = 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.

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).

- 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.

- 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 fonction¹ 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.

- 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 $T' \leq T$.
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`.

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`;

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.

- 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

- 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).

- 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.

- 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 ≤ 24 bits vers **float** et **double** ;
 - *upcasting* d'entier ≤ 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) → certains **int** ne sont pas représentables en **float**.

2. Selon implémentation, mais pas de garantie. Cherchez à quoi sert ce mot-clé!

- Pour *upcasting* d'entier de ≤ 32 bits vers ≤ 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 littéral², 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**, **i2i**
 - *upcasting* avec perte : **f2d** (sans **strictfp**), **i2f**, **i2d**, **i2f**
 - *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.

Et concrètement, que font les conversions ?

- *Upcasting* d'entier ≤ 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**, **i2i**) : 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.

- `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`.

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

- 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**.

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...

- L'opérateur + fonctionne avec différents types de nombres. C'est une forme de polymorphisme résolue à la compilation¹.

```
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`!

- **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 recompilation¹, 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 = 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 classe² 2 méthodes ayant même nom et même signature (ou 2 constructeurs de même signature).

→ 2 entités surchargées ont forcément une signature différente³.

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 !

- Une signature (p_1, \dots, p_n) **subsume** une signature (q_1, \dots, 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 $f(e_1, e_2, \dots, e_n)$ dans un code source, la **signature d'appel** est le n -uplet de types (t_1, t_2, \dots, t_n) tel que t_i est le type de l'expression e_i (tel que détecté par le compilateur).

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 règles¹ pour éliminer d'autres candidates;
- 5 s'il reste plusieurs candidates à ce stade, renvoyer une erreur (appel ambigu);
- 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.

```
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"  
    }  
}
```

```
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).

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-typage¹ (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.

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).

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

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

- constructeurs interdits;
- tous les membres implicitement² **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.

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 directement² 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()` { ... }.

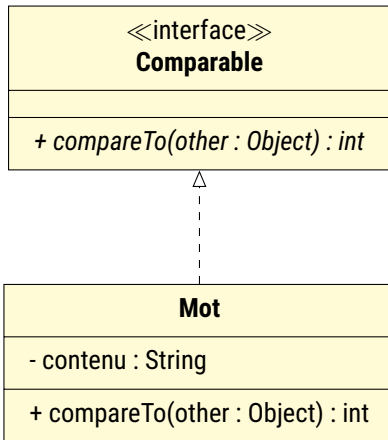
```
public interface Comparable { int compareTo(Object other); }

class Mot implements Comparable {
    private String contenu;

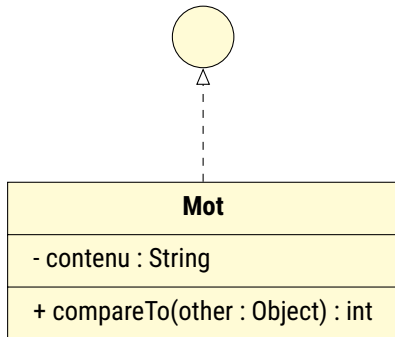
    public int compareTo(Object other) {
        return ((Mot) autreMot).contenu.length() - contenu.length();
    }
}
```

- 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 { ... }.

```
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 []  
    }  
}
```



ou version "abrégée" :

Comparable

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

- **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 → 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).


```
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.

Une classe peut hériter de plusieurs implémentations d'une même méthode, via les interfaces qu'elle implémente (méthodes **default**, Java ≥ 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.

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.

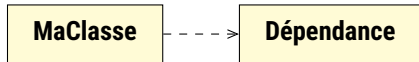
Java n'oblige pas à lever l'ambiguïté dans ce cas.

```
interface I {  
    default void f() { System.out.println("I"); }  
}  
  
class B {  
    public void f() { System.out.println("B"); }  
}  
  
class A extends B implements I {}
```

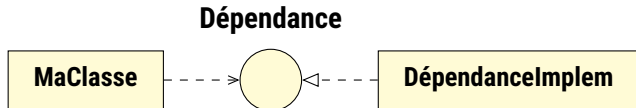
Ce programme compile et `new A().f();` affiche `B`.

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.

- 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 !

Pourquoi programmer à l'interface :

Une classe qui mentionne par son nom une autre classe contient une dépendance statique¹ 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 ».
→ 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.

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());
```

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

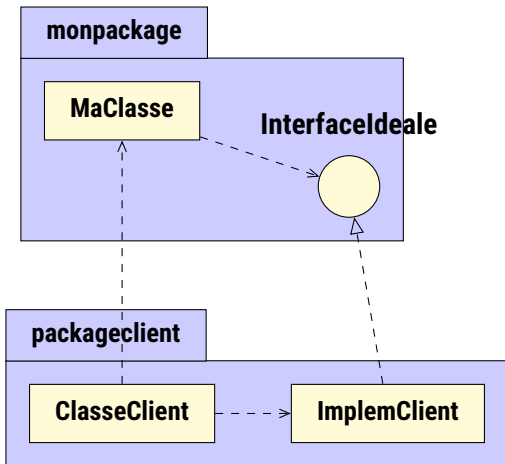
```
public class MyClass {  
    private final Supplier<Dep> factory;  
    public MyClass(Supplier<Dep> factory) { this.factory = factory; }  
    /* plus loin */ Dep uneInstanceDeDep = factory.get();  
}
```

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

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

- **Quand ?** quand on programme une bibliothèque dépendant 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 interface³ (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)



- **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"*.

- **Quand ?**

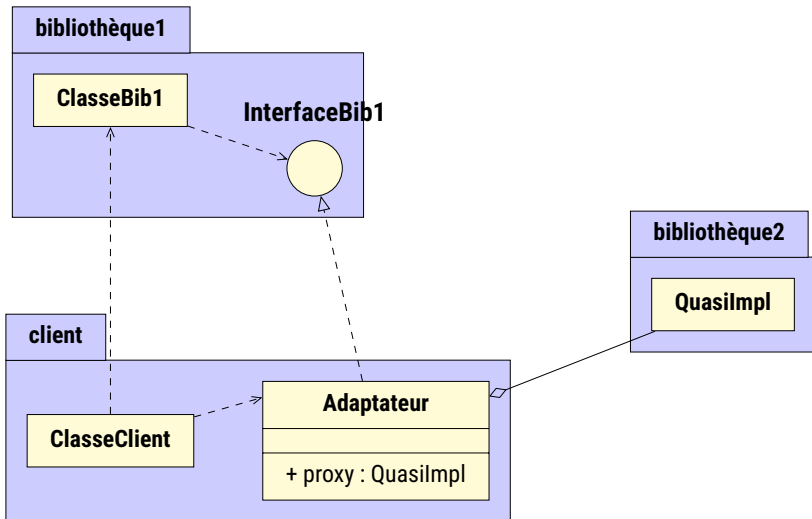
- vous voulez utiliser une bibliothèque dont les méthodes ont des paramètres typés par une certaine interface **I**.
- mais vous ne disposez pas de classe implémentant **I**
- cependant, une autre bibliothèque vous fournit une classe **C** contenant la même fonctionnalité que **I** (ou presque)

- **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 implémentées¹ par des appels de méthodes sur **proxy**.

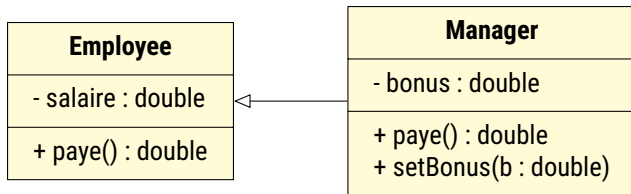
1. De préférence très simplement et brièvement...



- L'**héritage** est un mécanisme pour définir une nouvelle classe¹ **B** à partir d'une classe existante **A** : **B** récupère les caractéristiques² 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 ».

```
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 flèche : trait plein et tête en triangle côté superclasse, pour signifier "hérite de".
La méthode redéfinie (`paye`) apparaît à nouveau dans la sous-classe.

- D'après le folklore : « **piliers** » de la P00 = encapsulation, polymorphisme et héritage.
- **Mais ce dernier principe ne définit pas du tout la P00!**¹
- Héritage : mécanisme de réutilisation de code très pratique, mais non fondamental.
- \exists L00 sans héritage : les premières versions de Smalltalk ; le langage Go. La P00 moderne incite aussi à préférer la composition à l'héritage.²

Avertissement : l'héritage, mal utilisé, est souvent source de rigidité ou de fragilité³. 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 : P00 = 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* »

En POO, en théorie :

- implémentation d'interface \leftrightarrow spécification d'un supertype
- héritage/extension \leftrightarrow 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.

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.

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**

- 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 (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 : $A \prec A_1 \prec \dots \prec A_n \prec B^1$.
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 implique² 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 $\Rightarrow \prec \subset \prec$: 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.

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 classes¹.

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.

`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.

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`.
- 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()`.

Dans une sous-classe :

- On **hérite** des membres visibles¹ de la superclasse directe².
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 identique³, 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.

...

- 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 redéfinies et jamais qu'elles sont masquées... mais ne dit pas non plus que le terme est inapproprié.

- L'accès à toute définition visible (masquée ou pas) de la surperclasse 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
```

- 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 !
```

```
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 callable via super.g()).  
    protected static void g() {}  
    // redéfinit f() hérité de GrandParent (tjs callable 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.

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

```
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 !
}
```

```
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.

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'instance¹ : 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.


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

```
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
```

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** ..

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 contexte¹ 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

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 $M_f := \{ \text{méthodes de nom « f » dans C, compatibles avec (x1, x2, ...)} \}$.
- 3 Pour tout supertype direct **S** de **C**, $M_f += \{ \text{méthodes de nom « f » dans S, compatibles avec x1, x2, ..., non masquées}^2 \text{ par autre méthode dans } M_f \}$.
- 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**.

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.

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 pointeur¹ 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 !

- 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  
    }  
}
```

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 ?


```
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 commençant par `((X2)x).` se comportent comme celles commençant par `x.` : les mêmes signatures sont connues dans `X2` et `X3`.

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 redéfinition¹, 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