# Compléments en Programmation Orientée Objet

Aldric Degorre

Version `2020.11.00` du 27 novembre 2020

**Chargés de TP en 2020 :**
Isabelle Fagnot [1] (lundi et mardi), Yan Jurski [2] (jeudi) et Aldric Degorre [3] (vendredi).

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

---

1. `fagnot@irif.fr`
2. `jurski@irif.fr`
3. `adegorre@irif.fr`

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

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

1. même dans le style objet !

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

Sauriez-vous encore ?

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

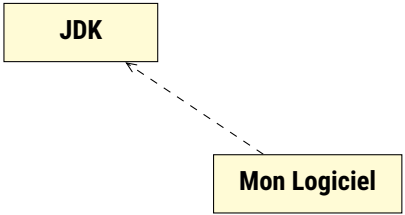
*Et auriez-vous l'audace d'imprimer le code et de l'afficher dans votre salon ?*

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

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

## La vraie vie ?



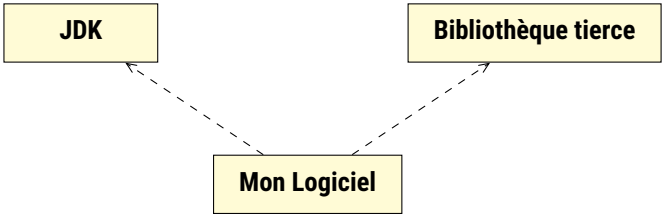



Jusqu'à présent, vos projets ressemblaient à ça (un logiciel exécutable qui ne dépend que du JDK).

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

## La vraie vie ?



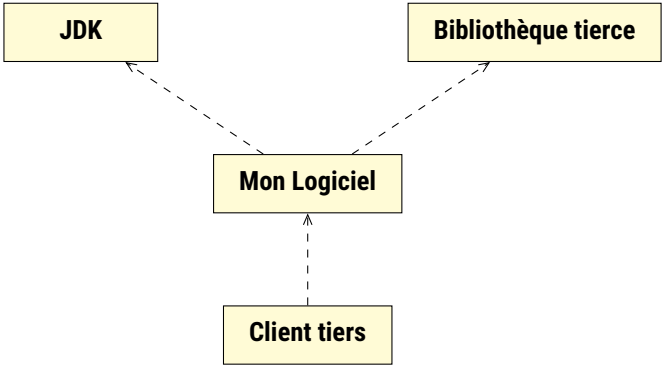



Ou parfois, à ça (dépendance à une ou des bibliothèques tierces).

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

## La vraie vie ?






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

1. mais rarement en L2...

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

## La vraie vie ?

De plus, tous ces logiciels évoluent au cours du temps (versions successives).

**Comment s'assurer alors :**

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

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

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

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

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

La programmation orientée objet (à condition qu'elle soit réellement pratiquée) permet une telle hygiène.

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.

# À propos de ce cours

## Objectifs, contexte



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

**Contexte :**

- Ce cours fait suite au cours de POO-IG de L2.
- Il précède les cours LOA de M1 (langage C++) et POCA de M2 (langage Scala).

**Contenu :**

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

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

# À propos de ce cours

## Pourquoi Java ?

**Pourquoi un autre “cours de Java” ?**

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

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

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

# Organisation du cours

- **Volume horaire :**
  - 2h de CM toutes les 2 semaines,
  - 2h de TP chaque semaine.
- **En ligne :** 1 cours sur Moodle UP [1] pour télécharger tous les documents (ce cours, les fiches de TP, ...) et déposer vos travaux (dont les projets).
  **Vérifiez que vous êtes bien inscrit.** (ou contactez-moi au plus vite !)

**Assurez-vous d'avoir vos identifiants pour vos comptes à l'UFR d'Informatique en arrivant au premier TP !** [2]

Restez vigilants, d'autres ressources seront mises en places s'il faut passer ce cours ou ses TP en mode distanciel.

---

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

# Modalités de contrôle



(Tout peut encore changer, notamment à cause de l'évolution de la situation sanitaire, auquel cas je vous en avertirai aussi vite que possible.)

- En première session :
  Un projet de programmation $\rightarrow$ 40% de la note,
  Un contrôle final écrit $\rightarrow$ 30% de la note.
  Le reste (QCMs en amphi, devoir ou note de TP) $\rightarrow$ 30% de la note,
- En session de rattrapage : un unique examen (100%).

(Spoiler ?) ... ah et oui, les *threads* et la programmation concurrente sont toujours « tombés » à l'examen.

# À propos de ces transparents

- À la fois de support de présentation et support de révision
  **Tous ne sera pas présenté en amphi → récupérez le cours complet sur Moodle !**
- Ce document va évoluer
  - Chaque semaine : ajout du cours de la semaine (voire de la semaine d'après).
  - N'importe quand : corrections d'erreurs, ajouts de précisions, d'explications, d'exemples (selon besoins, n'hésitez pas à demander !)
- Ainsi, versions numérotées avec numéros de la forme : `aaaa.ss.rr` :
  - `aaaa` est l'année (2019, pour vous),
  - `ss` est le numéro de semaine du dernier cours ajouté (i.e. : 0, 2, 4, 6, 8 et 10 ; le cours de semaine $s$ est appliqué dans les TP des semaines $s+1$ et $s+2$)
  - et `rr` est le numéro de révision (0 pour la première version publique du cours de la semaine, puis 1 à la première correction, etc.).

  Exemple : la version de ce cours est "`2020.11.00`".
- Pour faciliter la (re-)lecture, un code de couleurs est mis en place...

# Page “normale”

Les pages de cette couleur sont les pages “normales” (ce qui ne veut rien dire mais...). Il peut s’agir :

- de celles qui ne tombent dans aucune des autres catégories
- de celles qui devraient appartenir à plusieurs catégories
- et enfin de celles qui n’ont pas encore été catégorisées (oubli, hésitation...)

# Révision

Les pages de cette couleur sont celles dont le contenu constituent une révision du cours POO-IG de L2.

Vérifiez que vous maîtrisez bien les notions qui y sont abordées.

# Exemple/illustration

Les pages de cette couleur sont celles qui contiennent un exemple ou une illustration et un éventuel commentaire.

Dès lors qu'on généralise sur l'exemple, ce sera une diapo "normale".

# Détail/développement

Les pages de cette couleur contiennent des détails techniques qui apportent peu à la compréhension profonde des notions en cours de présentation... mais sont indispensables pour les utiliser concrètement.

Probablement ces pages ne seront que très rapidement montrées lors du cours magistral, mais il sera indispensable de leur consacrer du temps en seconde lecture.

# Réflexion préliminaire/analyse

Les pages de cette couleur correspondent à des discussions servant à amener la notion qui est sur le point d'être abordée.

On y décrit un problème et ses enjeux... mais pas encore la solution apportée par Java.

Le contenu est un raisonnement qu'il faut comprendre, mais pas apprendre par cœur.

# Synthèse/résumé/à retenir

Les pages de cette couleur contiennent une synthèse des dernières pages, c'est-à-dire soit un résumé, soit le résultat principal à retenir.

# Discussion/ouverture

Les pages de cette couleur contiennent une discussion, un commentaire, des remarques, ou bien une ouverture relative à ce qui vient d'être abordé.

Ces pages doivent servir à faire réfléchir, mais ne sont en aucun cas à apprendre par cœur.

# Approfondissement



Les pages de cette couleur contiennent un approfondissement du cours. Cela veut dire que le contenu est intéressant à apprendre, dès lors qu'on comprend le reste du cours, mais n'est pas prioritaire (sur les pages "normales").

# Autres “codes”

## Marquage du texte

- “**Blah :**” : titre de paragraphe
- “important” : mot ou passage important
- “**très important**” : mot ou passage très important
- “**concept**” : concept clé (en général défini explicitement ou implicitement dans le transparent... sinon, il faudra s’assurer de connaître ou trouver la définition)
- “*foreign words*” : passage en langue étrangère
- “`void duCodeJavaEnLigne( ){}`” : code java intégré au texte
- 

```
    void duCodeJavaNonIntegre( ) {
        System.out.println("Java c'est cool !");
    }
```

Code non intégré au texte.

# Autres “codes”

## Références bibliographiques

- JLS : *Java language specification* (Oracle)
- JVMS : *Java virtual machine specification* (Oracle)
- EJ3 : *Effective Java 3rd edition* (Joshua Bloch)
- JCiP :*Java Concurrency in Practice* (Brian Goetz)
- GoF : *Design Patterns : Elements of Reusable Object-Oriented Software* (Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides a.k.a. ”the Gang of Four”)
- DPDP : *Design Principles and Design Patterns* (Robert C. Martin)
  (C’est juste un article, mais riche de principes utiles.)

# La programmation orientée objet

## Historique



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

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

# La programmation orientée objet

## Principes



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

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

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

# La programmation orientée objet

## Avantages et raisons du succès de la POO

La POO permet de découper un programme en composants

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

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

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

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

En 2020, Java est donc « dans la force de l'âge » (24 ans) [2], à comparer avec :

- même génération : Haskell : 30 ans, Python : 29 ans, JavaScript, OCaml : 24 ans
- anciens : C++ : 35 ans, C : 42 ans, COBOL : 61 ans, Lisp : 63 ans, Fortran : 66 ans...
- modernes : Scala : 16 ans, Go : 11 ans, Rust : 10 ans, Swift : 6 ans, Kotlin : 4 ans, ...

Depuis plusieurs années, Java est le 1er ou 2ème langage le plus populaire. [3]

1. Auteurs principaux : James Gosling et Patrick Naughton.
2. Je considère la version 1.0 de chaque langage. Mais sur le web, Java a déjà fêté ses 25 ans cet été !
3. Dans la plupart des classements principauxs : TIOBE, RedMonk, PYPL, ...
Selon les métriques, l'autre langage le plus populaire est C, Javascript ou Python.

Versions « récentes » de Java :

- **03/2014 :** Java SE 8, la version long terme précédente ;[1]
- **09/2018 :** Java SE 11, la version long terme actuelle ;[2]
- **03/2019 :** Java SE 12, la dernière version
- **15/09/2019 :** Java SE 15, la prochaine version, imminente !
- **09/2021 :** Java SE 17, la prochaine version long terme.

Ce cours utilise Java 11, mais :

- la plupart de ce qui y est dit vaut aussi pour les versions antérieures ;
- les nouveautés de Java 12 à 15 ne sont pas censurées.

1. Utilisée pour POOIG et CPOO5 jusque l'an passé.
2. Depuis Java 9, une version "normale" sort tous les 6 mois et une à "support long terme", tous les 3 ans.

# Java

## Caractérisitiques principales

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

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

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

**Domaines d'utilisation :**

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

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

**Mais :**

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

---

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

# Compilation

**Compilation** : traduction code source (lisible par l'humain) $\rightarrow$ code « machine »

Une seule fois pour une version donnée du code source.

**Cas classique :** machine = architecture physique (ex : processeur x86-64, ARM, etc.)

**Cas de Java :** machine = JVM, la **machine virtuelle** [1] de Java.

Dans les deux cas : transformation d'un langage évolué, plus ou moins haut-niveau, vers une séquence d'instructions (appartenant à un **jeu d'insructions** de petite taille).

1. programme qui exécute des programmes écrits en **code-octet**. Ce programme est lui-même exécuté sur une **machine hôte** (machine physique ou bien autre machine virtuelle...).

# Compilation

## Autour de la JVM

- « JVM » est le standard de machine virtuelle pour Java publié [1] par *Oracle*.
- L'implémentation par *Oracle* est appelée *HotSpot* (plusieurs distributions [2]).
- Implémentation tierce compatible avec HotSpot : *OpenJ9* (fondation *Eclipse*).
- VMs incompatibles : notamment *Dalvik* (sous Android).
- On peut même compiler Java vers d'autres 'cibles' : code natif (cf GCJ, cf ART dans Android $\geq 5$, jaotc/Graal dans Java $\geq 9$ , ...), voire vers autre langage (ex : Javascript, via GWT).
- Vice-versa, d'autres langages que Java sont compilés pour la JVM : Scala, Groovy, Clojure, Gosu, Ceylon, Kotlin...

1. Spécifié par la "JVMS" : `https://docs.oracle.com/javase/specs/jvms/se11/html/index.html`
2. L'officielle d'Oracle, ainsi que toutes les dérivées d'OpenJDK : AdoptOpenJDK, Amazon Corretto, Microsoft Azul Zulu, BellSoft Liberica, SAP SapMachine, ...

# Exécution

**Exécution** : à chaque fois qu'on veut utiliser le programme. La machine traite une à une les opérations élémentaires dictées par le code compilé.

**Dans le cas classique :** le processeur physique (CPU), lit les instructions une par une et active à chaque fois le circuit dédié correspondant.

**Dans le cas de Java :** la JVM traduit le code-octet à la volée en instructions natives pour le CPU, qui les exécute immédiatement.

*code source Java* $\xrightarrow[\textit{compilation}]{\texttt{javac}}$ *code-octet* $\xrightarrow[\textit{exécution}]{\texttt{java (JVM)}}$ *instructions CPU "réelles"*

# Compilation ?

Est-ce que le bytecode est vraiment juste interprété par la JVM ?

En réalité, plusieurs stratégies :

- Simple interprétation du code-octet au fur et à mesure de son exécution.
- JIT, « Just In Time Compilation » : pendant l'exécution du programme, la VM traduit (une bonne fois pour toutes) en code natif optimisé les morceaux de code qui s'exécutent souvent
- AOT, « Ahead Of Time Compilation » : "on" compile tout ou partie du code-octet vers des instructions natives avant son exécution

La JVM HotSpot (JVM par défaut depuis Java 3), fait du JIT. Depuis $\sim$ Java 9, Oracle expérimente AOT via GraalVM.

# Le « programme » Java

## Architecture du code source



**Le code source** : fichier `.java` ∈ paquetage ∈ module ∈ projet

- la base : fichiers « source » `.java` et éventuellement ressources diverses (images, sons, polices de caractères, etc.) ;
- fichiers regroupés en **paquetages** (matérialisés par des sous-répertoires) ;
- si on utilise JPMS (Java ≥ 9), paquetages regroupés en **modules** (décrits dans les fichiers `module-info.java`) [1] ;
- paquetages (et modules) souvent regroupés en « **projets** » [2].
  Un tel projet est typiquement un répertoire muni d'un ou plusieurs fichiers de configuration (propriétaires à l'outil utilisé).

1. Dans IntellIJ IDEA, ces modules correspondent désormais aux modules du projet, subdivision déjà proposée par cet IDE, avant Java 9.
2. C.-à-d. un ensemble de packages ou modules partageant une configuration commune dans un IDE (comme Eclipse, NetBeans, IntellIJ IDEA, ...) ou dans un moteur de production (make, ant, maven, gradle, ...). Dans Eclipse, en plus, un « espace de travail » regroupe les projets apparaissant dans une même fenêtre.

# Le « programme » Java

## Architecture d'un programme compilé/distribuable

**Le code compilé** :

- organisé de façon similaire au code source.
- Mais, à chaque un fichier `.java` correspond (au moins) un fichier `.class`.
- Un programme compilé est distribuable via une ou des archives `.jar`[1].
- Si on utilise JPMS, il y a exactement un fichier `.jar` par module.

1. C'est en réalité un fichier `.zip` avec quelques méta-données supplémentaires.

# Question(s) de style

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

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

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

# Question de goût ?

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

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

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

# Une hiérarchie de normes

Habitudes dictées par :

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

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

» et enfin... le bon sens ! [3]

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

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

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



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

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

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

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

# Nommer les entités

## Codage, et langue

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

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

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

# Nommer les entités

## Nature grammaticale (1)



Nature grammaticale des identifiants :

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

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

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

# Nommer les entités

## Nature grammaticale (2)

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

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

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

# Nommer les entités

## Concision versus information

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

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

# Nombre de caractères par ligne

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

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

# Indentation



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

**Exemple :**

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

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

# Où couper les lignes

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

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

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

# Taille des classes

## Quelle est la bonne taille pour une classe ?

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

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

# Taille des méthodes

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

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

# Nombre de paramètres des méthodes

Autre critère : le nombre de paramètres.

Trop de paramètres ($>4$) implique :

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

Il est souvent possible de réduire le nombre de paramètres

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

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

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

# Et ainsi de suite

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

# Commentaires

## Plusieurs sortes de commentaires (1)



- En ligne :

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

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

- en bloc :

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

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

# Commentaires

## Plusieurs sortes de commentaires (2)



- en bloc JavaDoc :

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

# Commentaires

## La JavaDoc



**À propos de la JavaDoc :**

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

# Patrons de conception (1)

ou *design patterns*



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

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

# Patrons de conception (2)

ou *design patterns*

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

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

# Objets

## Vision haut niveau





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

... oui mais concrètement ?

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

# Objets

## Vision bas niveau (en Java)



**Object** = entité...

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

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

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

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

# Objets

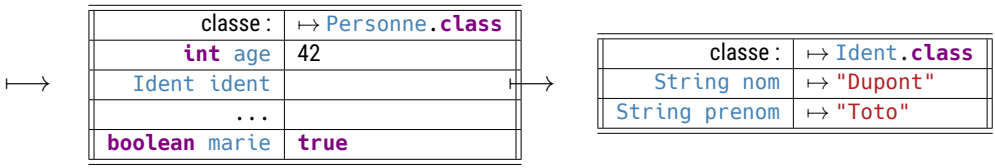
## Quelle bonne vision bas niveau ?



Cette dernière vision est en fait réductrice.

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

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




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

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

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

# Objets

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



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

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

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

# Objets

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



**Question :** où arrêter le graphe d'un objet ?

- Est-ce que les éléments d'une liste font partie de l'objet-liste ?
- En exagérant un peu, un programme ne contient en réalité qu'un seul objet ! [1]
- Clairement, le graphe d'un objet ne doit pas contenir *tous* les enregistrements accessibles depuis l'enregistrement principal. Mais où s'arrêter et sur quel critère ?

**Cela n'est pas anodin :**

- Que veut dire « copier » un objet ? (Quelle « profondeur » pour la copie ?)
- Si on parle d'un objet non modifiable, qu'est-ce qui n'est pas modifiable ?
- Est-ce qu'une collection non modifiable peut contenir des éléments modifiables ?

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

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

# Classes

## Langages à prototypes, langages à classes



- **Besoin :** créer de nombreux objets similaires (même interface, même schéma de données).
- **2 solutions → 2 familles de LOO :**
  - **LOO à classes** (Java et la plupart des LOO) : les objets sont instanciés à partir de la description donnée par une **classe** ;
  - **LOO à prototypes** (toutes les variantes d'ECMAScript dont JavaScript ; Self, Lisaac, ...) : les objets sont obtenus par extension d'un objet existant (le prototype).

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

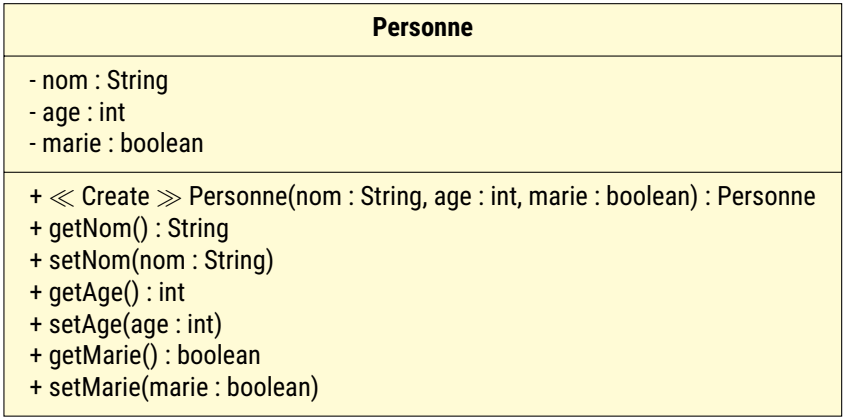
Pour l'objet juste donné en exemple, la classe `Personne` pourrait être :

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

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

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

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

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

# Classes

## Exemple (le même en UML)





# Classes

## Points de vue pertinents pour Java

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

**Autres points de vue :**

Classe =

- ensemble cohérent de définitions (champs, méthodes, types auxiliaires, ...), en principe relatives à un même type de données
- conteneur permettant l'encapsulation (= limite de visibilité des membres privés). [2]

1. "similaires" = utilisables de la même façon (même type) et aussi structurés de la même façon.
2. Remarque : en Java, l'encapsulation se fait par rapport à la classe et au paquetage et non par rapport à l'objet. En Scala, p. ex., un attribut peut avoir une visibilité limitée à l'objet qui le contient.

# Classes

## Autres points de vue (non-OO) :

Classe =

- sous-division syntaxique du programme
- espace de noms (définitions de nom identique possibles si dans classes différentes)
- parfois, juste une bibliothèque de fonctions statiques, non instanciable [1]
  exemples de classes non instanciables du JDK : `System`, `Arrays`, `Math`, ...

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

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

# Instanciation

## De la classe à l'objet

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

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

# Instanciation

## Constructeurs (1)



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

- **Déclaration :**

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

**NB :** même nom que la classe, pas de type de retour, ni de `return` dans son corps.

- Typiquement, "`// instructions`" = initialisation des attributs de l'instance.
- Appel toujours précédé du mot-clé `new` :

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

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

1. En toute rigueur, un constructeur n'est pas une méthode. Notons tout de même les similarités dans les syntaxes de déclaration et d'appel et dans la sémantique (exécution d'un bloc de code).

**Il est possible de :**

- définir plusieurs constructeurs (tous le même nom → cf. surcharge) ;
- définir un constructeur secondaire à l'aide d'un autre constructeur déjà défini : sa première instruction doit alors être `this(paramsAutreConstructeur);` [1] ;
- ne pas écrire de constructeur :
  - Si on ne le fait pas, le compilateur ajoute un **constructeur par défaut** sans paramètre. [2].
  - Si on a écrit un constructeur, alors il n'y a pas de constructeur par défaut [3].

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

# Membres d'une classe

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

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

Un membre `m` peut être

- soit non statique ou **d'instance** (relatif à une instance de `C`)
  Utilisable en écrivant « `m` » n'importe où où un `this` (**récepteur** implicite) de type `C` existe et ailleurs en écrivant « `recepteurDeTypeC.m` ».
- soit **statique** (relatif à la classe `C`) → mot-clé `static` dans déclaration.
  Utilisable sans préfixe dans le corps de `C` et ailleurs en écrivant « `C.m` ».

Les **membres d'un objet** donné sont les membres non statiques de la classe de l'objet.

1. D'après la JLS 8.2, les constructeurs ne sont pas des membres. Néanmoins, sont déclarés à l'intérieur d'une classe et acceptent, comme les membres, les modificateurs de visibilité (`private`, `public`, ...).
2. Souvent abusivement appelés "classes internes".

# Membres d'une classe

## Exemple



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

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

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

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

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

# Membres d'une classe

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



**Contexte** (associé à tout point du code source) :

- dans une définition [1] statique : contexte = la classe contenant la définition ;
- dans une définition non-statique : contexte = l'objet "courant", le **récepteur** [2].

Désigner un membre `m` déjà défini quelque part :

- écrire soit juste `m` (**nom simple**), soit `chemin.m` (**nom qualifié**)
- "`chemin`" donne le contexte auquel appartient le membre `m` :
  - pour un membre statique : la classe [3] (ou interface ou autre...) où il est défini
  - pour un membre d'instance : une instance de la classe où il est défini
- "`chemin.`" est facultatif si `chemin` == contexte local.

1. typiquement, remplacer "définition" par "corps de méthode"
2. L'objet qui, à cet endroit, serait référencé par `this`.
3. Et pour désigner une classe d'un autre paquetage : `chemin` = `paquetage.NomDeClasse`.

# Membres statiques et membres non statiques

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

**Membre statique** = lié à (la durée de vie et au contexte d') une classe [1].

| | statique (ou "de classe") | non statique (ou "**d'instance**") |
|---|---|---|
| attribut | donnée globale [2], commune à toutes les instances de la classe. | donnée propre [3] à chaque instance (nouvel exemplaire de cette variable alloué et initialisé à chaque instanciation). |
| méthode | "fonction", comme celles des langages impératifs. | message à instance concernée : le **récepteur** de la méthode (`this`). |
| type membre | juste une classe/interface définie à l'intérieur d'une autre classe (à des fins d'encapsulation). | comme statique, mais instances contenant une référence vers instance de la classe englobante. |

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

# Membres statiques et membres non-statiques

## Zoom sur le cas des attributs



Qu'affiche le programme suivant ?

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

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

# Membres statiques et membres non-statiques

## Zoom sur le cas des attributs



### Qu'affiche le programme suivant ?

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

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

### Réponse :

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

# Membres statiques et membres non-statiques

## Zoom sur le cas des méthodes



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

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

**Mais différences essentielles :**

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

# Fabriques statiques

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



**Problème, les limitations des constructeurs :**

- même nom pour tous, qui ne renseigne pas sur l'usage fait des paramètres ;
- impossibilité d'avoir 2 constructeurs avec la même signature ;
- si appel à constructeur auxiliaire, nécessairement en première instruction ;
- obligation de retourner une nouvelle instance → pas de contrôle d'instances [1] ;
- obligation de retourner une instance directe de la classe.

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

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

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

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

# Encapsulation

Concept essentiel de la POO



## L'**encapsulation**

- = restriction de l'accès depuis l'extérieur aux choix d'implémentation internes.
- **bonne pratique** favorisant la pérennité d'une classe.
  Minimiser la « surface » qu'une classe expose à ses clients [1] (= en réduisant leur **couplage**) facilite son déboguage et son évolution future. [2]
- empêche les clients d'accéder à un objet de façon incorrecte ou non prévue. Ainsi,
  - la correction d'un programme est plus facile à vérifier (moins d'intéractions à vérifier) ;
  - plus généralement, seuls les **invariants de classe** [3] ne faisant pas intervenir d'attributs non privés peuvent être prouvés.

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

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

# Encapsulation

## Exemple

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

**Pas bien :**

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

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

$\rightarrow$ on ne peut rien prouver !

# Encapsulation

## Exemple

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

**Pas bien :**

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

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

→ on ne peut rien prouver !

**Bien :** (ou presque)

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

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

→ s'il y a un bug, c'est dans la méthode `next` et pas ailleurs ![1]

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

# Encapsulation

## Remarque

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

# Encapsulation

## Niveaux de visibilité : private, public, etc.



L'encapsulation est mise en œuvre via les **modificateurs de visibilité** des membres.

4 niveaux de visibilité en faisant précéder leurs déclarations de `private`, `protected` ou `public` ou d'aucun de ces mots (→ visibilité *package-private*).

| Visibilité | classe | paquetage | sous-classes [1] | partout |
|---|---|---|---|---|
| `private` | X | | | |
| *package-private* | X | X | | |
| `protected` | X | X | X | |
| `public` | X | X | X | X |

**Exemple :**

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

1. voir héritage

# Encapsulation

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



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

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

| Visibilité | paquetage | partout |
|---|---|---|
| *package-private* | X | |
| **`public`** | X | X |

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

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

# Encapsulation

## Niveaux de visibilité et documentation



- Toute déclaration de membre non `private` est susceptible d'être utilisée par un autre programmeur dès lors que vous publiez votre classe.
- Elle fait partie de l'API [1] de la classe.
- → vous devez donc **la documenter** [2] (EJ3 Item 56)
- → et vous vous engagez **à ne pas modifier** [3] sa spécification [4] dans le futur, sous peine de "casser" tous les clients de votre classe.

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

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

# Encapsulation

## Niveaux de visibilité : avertissements



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

- *package-private* → on peut, par inadvertance, créer une classe dans un paquetage déjà existant [1] → garantie faible.
- `protected` → de même et, en +, toute sous-classe, n'importe où, voit la définiton.
- **Aucun niveau ne garantit la confidentialité des données.**
  Constantes : lisibles directement dans le fichier `.class`.
  Variables : lisibles, via réflexion, par tout programme s'exécutant sur la même JVM.
  Si la sécurité importe : bloquer la réflexion [2].

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

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

# Encapsulation

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



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

Syntaxe du fichier `module-info.java` :

```
module nom_du_module {
    requires nom_d_un_module_dont_on_depend;
    exports nom_d_un_package_defini_ici;
}
```

Ce sujet sera développé en TP.

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

# Encapsulation

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



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

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

# Encapsulation

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

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

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

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

- L'utilisation d'accesseurs laisse la **possibilité de changer ultérieurement l'implémentation** de la propriété, sans changer son mode d'accès public [1].
  Ainsi, quand cela sera fait, il ne sera **pas nécessaire de modifier les autres classes** qui accèdent à la propriété.

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

# Encapsulation

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



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

```
public final class Person {

    // propriété "age"

    // attribut de base (qui doit rester positif)
    private int age;

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

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

# Encapsulation

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



**Exemple :** propriété en lecture seule avec évaluation paresseuse.

```
public final class Entier {
    public Entier(int valeur) { this.valeur = valeur; }

    private final int valeur;

    // propriété ``diviseurs'' :
    private List<Integer> diviseurs;

    public List<Integer> getDiviseurs() {
        if (diviseurs == null) diviseurs =
             Collections.unmodifiableList(Outils.factorise(valeur)); // <- calcul
             coûteux, à  n'effectuer que si nécessaire
        return diviseurs;
    }
}
```

# Encapsulation

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



**Comportements envisageables pour `get` et `set` :**

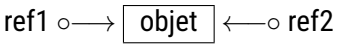
- contrôle de validité avant modification ;
- initialisation paresseuse : la valeur de la propriété n'est calculée que lors du premier accès (et non dès la construction de l'objet) ;
- consignation dans un journal pour déboguage ou surveillance ;
- observabilité : le setteur notifie les objets observateurs lors des modifications ;
- vétoabilité : le setteur n'a d'effet que si aucun objet (dans une liste connue de "véto-eurs") ne s'oppose au changement ;
- ...

# Encapsulation

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



***Aliasing*** = existence de références multiples vers un même objet.




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

À éviter :




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

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

# Encapsulation

## *Aliasing* : exercice



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

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

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

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

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

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

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

# Encapsulation

## *Aliasing* : comment l'empêcher.



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

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

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

# Copie défensive

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



- **Copie défensive** = copie profonde réalisée pour éviter des *alias* indésirables.
- **Copie profonde :** technique consistant à obtenir une copie d'un objet « égale » [1] à son original au moment de la copie, mais dont les évolutions futures seront indépendantes.
- **2 cas, en fonction du genre de valeur à copier :**
  - Si type primitif ou immuable [2], pas d'évolutions futures $\rightarrow$ une copie directe suffit.
  - Si type mutable $\rightarrow$ on crée un nouvel objet dont les attributs contiennent des copies profondes des attributs de l'original (et ainsi de suite, récursivement : on copie le graphe de l'objet [3]).

---

1. La relation d'égalité est celle donnée par la méthode `equals`.
2. Type **immuable** (*immutable*) : type (en fait toujours une classe) dont toutes les instances sont des objets non modifiables.
C'est une propriété souvent recherchée, notamment en programmation concurrente.
Contraire : **mutable** (*mutable*).
3. Il savoir en quoi consiste le graphe de l'objet, sinon la notion de copie profonde reste ambiguë.

# Copie défensive

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



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

**Remarque :** il est impossible [1] de faire une copie profonde d'une classe mutable dont on n'est pas l'auteur si ses attributs sont privés et l'auteur n'a pas prévu lui-même la copie.

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

# Immuabilité...

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

Sont notamment immuables :

- la classe `String` ;
- toutes les *primitive wrapper classes* : `Boolean`, `Char`, `Byte`, `Short`, `Integer`, `Long`, `Float` et `Double` ;
- d'autres sous-classes de `Number` : `BigInteger` et `BigDecimal` ;
- plus généralement, toute classe [1] dont la documentation dit qu'elle l'est.

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

1. Les types définis par les interfaces ne peuvent pas être garantis immuables.
2. Fonctionnellement. Pour d'autres aspects, comme la performance, le comportement est différent.
3. Mais cette distinction n'a pas de sens pour des valeurs directes.

# Encapsulation

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



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

- on ne peut pas prouver d'invariant de `C` faisant intervenir `a`, notamment, la classe `C` n'est pas immuable (certaines instances pourraient être modifiées par un tiers) ;
- on ne peut empêcher les modifications concurrentes [1] de l'objet *alias*é, dont le résultat est notoirement imprévisible. [2]

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

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

# Encapsulation

## *Aliasing* : remarque sans rapport avec l'encapsulation

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

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

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

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

# Encapsulation

## *Aliasing*, est-ce toujours « mal » ?



Pour conclure sur l'*aliasing*.

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

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

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

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

# Types imbriqués

## Qu'est-ce que c'est ?



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

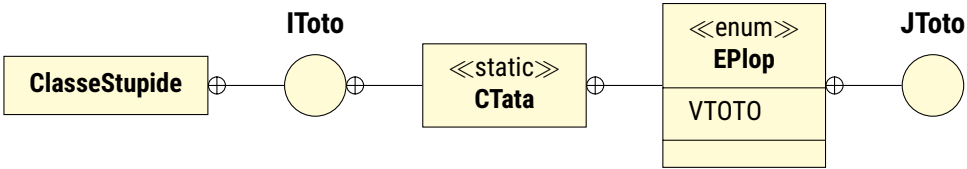
```
public class ClasseStupide {
    public static interface IToto {
        public static class CTata {
            public enum EPlop {
                VTOTO;
                public interface JToto { }
            }
        }
    }
}
```

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

# Types imbriqués

## En UML






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

# Types imbriqués

## Pour quoi faire ?



L'imbrication permet l'encapsulation des définitions de type et de leur contenu :

```
class A {
    static class AA { static int x; } // définition de x à  l'intérieur de AA
    private static class AB { } // comme pour tout membre, la visibilité peut être
         modifiée (ici private, mais public et protected sont aussi possibles)

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

}

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

# Types imbriqués

## Pour quoi faire ?



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

L’exemple ci-dessous compile :

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

    static class TIB {
        private static void fIB() { }
    }

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

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

# Plusieurs sortes de types imbriqués

## Définitions et classification



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

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

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

# Types imbriqués

## Exemple de type membre statique



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

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

    ...

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

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

# Types imbriqués

## Exemple de classe membre non statique



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

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

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

# Types imbriqués

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



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

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

```
class CE {
    int x = 1;
    class CI {
        int y = 2;
        void f() { System.out.println(CE.this.x + " " + this.y); }
    }
}
// alors new CE().new CI().f(); affichera "1 2"
```

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

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

# Types imbriqués

## Exemple de classe locale



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

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

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

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

# Types imbriqués

## Exemple de classe anonyme



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

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

1. La seule instance.

# Classes anonymes

**Classe anonyme** =

- cas particulier de classe locale avec syntaxe allégée
  → comme classes locales, accès aux déclarations du bloc [1] ;
- déclaration “en ligne” : c’est syntaxiquement une expression, qui s’évalue comme une instance de la classe déclarée ;
- déclaration de classe sans donner de nom ⟹ instanciable une seule fois
  → c’est une classe singleton ;
- autre restriction : un seul supertype direct [2] (dans l’exemple : `Iterator`).

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

**Syntaxe encore plus concise : lambda-expressions** (cf. chapitre dédié), par ex.
`x -> System.out.println(x)`.

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

# Classes anonymes

## Astuce avec Java $\geq$ 10



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

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

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

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

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

# Types imbriqués

## Accessibilité du contexte englobant



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

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

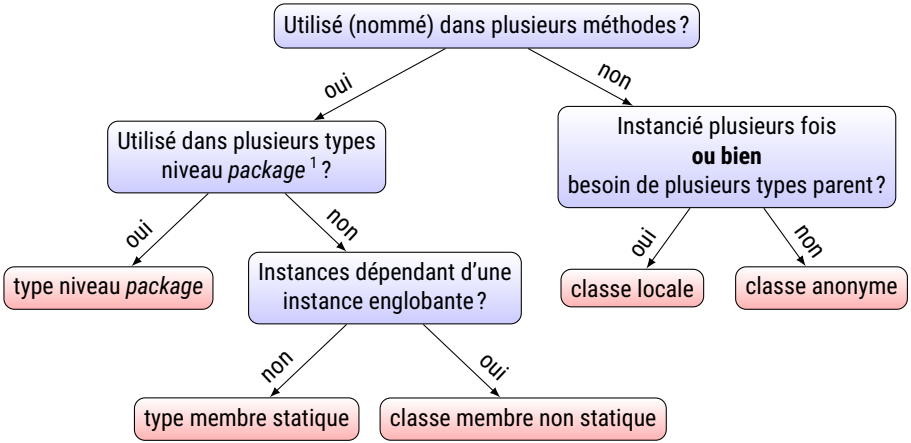
- dans tous les cas : les membres statiques du type englobant ;
- pour les classes membres non statiques et classes locales dans bloc non statique : tous les membres non statiques du type englobant ;
- pour les classes locales : les définitions locales [2].

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

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

# Plusieurs sortes de types imbriqués

## D'accord, mais lequel choisir ? [2]




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

# Types imbriqués

## Remarques et limitations diverses



- Dans une interface englobante, les types membres sont [1] **`public`** et **`static`**.
- Dans les classes locales (et anonymes), on peut utiliser les variables locales du bloc seulement si elles sont **effectivement finales** (c.à.d. déclarées **`final`**, ou bien jamais modifiées).

  Explication : l'instance de la classe locale peut "survivre" à l'exécution du bloc. Donc elle doit contenir une copie des variables locales utilisées. Or les 2 copies doivent rester cohérentes → modifications interdites.

  Une alternative non retenue : stocker les variables partagées dans des objets dans le tas, dont les références seraient dans la pile. On pourrait aisément programmer ce genre de comportement au besoin.
- Les classes internes [2] ne peuvent pas contenir de membres statiques (à part attributs **`final`**).

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

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

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

# Zones de la mémoire

## Vue d'ensemble

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

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

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

# Zones de la mémoire

## Le tas/*heap* et autres zones

**Tas :**

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

# Zones de la mémoire

## La pile/*stack* (1)

**Pile :**

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

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

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

# Zones de la mémoire

## La pile/*stack* (2)



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

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

# Types de données

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

# Système de types de Java

## Caractéristiques principales

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

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

# Types de données en Java

## Deux catégories

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

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

Il existe quelques autres différences, abordées dans ce cours.

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

# Types de données en Java

## Zoom sur les types primitifs



**Les 8 types primitifs :**

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

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

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

1. 1 **mot** = 32 bits

# Références et valeurs directes

## Interprétations



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

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

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

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

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

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

# Références et valeurs directes

## Conséquences sur l'affectation et l'égalité

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

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

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

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

# Références et valeurs directes

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

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

Ainsi :

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

**Conséquence :**

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

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

# Références et valeurs directes

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



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

---

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

# Typages dynamique et statique

## Quand vérifier la cohérence des données ?

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

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

  Typage statique $\rightarrow$ concerne les expressions du programme

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

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

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

# Stades de vérification et entités typables en Java

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

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

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

# Stades de vérification et entités typables en Java

## À la compilation : les expressions



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

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

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

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

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

# Stades de vérification et entités typables en Java

## À l'exécution : les objets



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

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

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

# Relation entre type statique et type dynamique

Pour une variable ou expression :

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

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

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

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

# Notion de sous-typage

## Définition abstraite



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

$\rightarrow$ plusieurs interprétations possibles (mais contraintes par notre définition de « type »).

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

# Notion de sous-typage

## Interprétations



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

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

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

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

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

# Notion de sous-typage

## Interprétations



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

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

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

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

# Notion de sous-typage

## Interprétations



Pourquoi le sous-typage structurel est insuffisant ?

**Exemple :**

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

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

# Notion de sous-typage

## Interprétations

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

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

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

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

# Notion de sous-typage

## Interprétations



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

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

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

# Notion de sous-typage

## Interprétations : en pratique

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

- Pour les 8 types primitifs, il y a une relation de sous-typage pré-définie.
- Pour les types référence, le sous-typage est nominal : *A* n'est sous-type de *B* que si *A* est déclaré comme tel (`implements` ou `extends`).

  Mais la définition de *A* ne passe la compilation que si certaintes contraintes structurelles [1] sont vérifiées, concernant les redéfinitions de méthodes.
- Types primitifs et types référence forment deux systèmes déconnectés. Aucun type référence n'est sous-type ou supertype d'un type primitif.

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

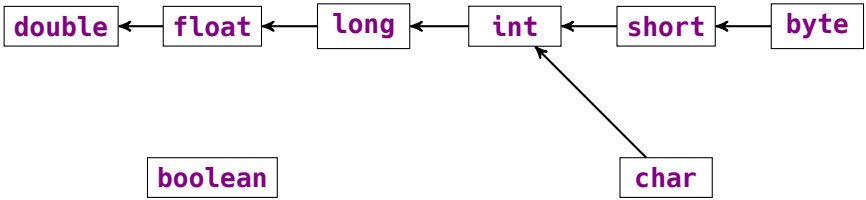
# Notion de sous-typage pour Java

## Relation de sous-typage en Java (1)



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

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




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

# Notion de sous-typage pour Java

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



**Types référence :**

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

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

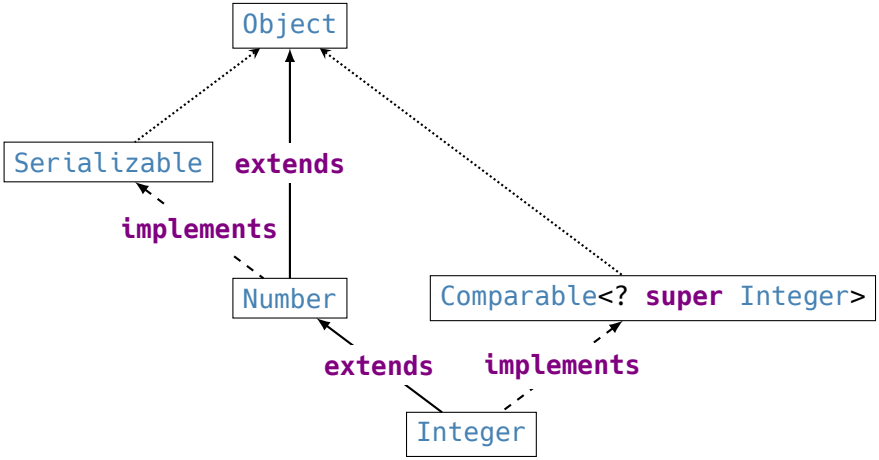
1. C'est vrai même si $A$ est une interface, alors même qu'aucune interface n'hérite de `Object`.
2. Pour être exhaustif, il manque les règles de sous-typage pour les types génériques.
3. Les types tableau sont des classes (très) particulières, implémentant les interfaces `Cloneable` et `Serializable`.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

# Notion de sous-typage pour Java

## Relation pour les types références, diagramme



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



# Notion de sous-typage pour Java

## Principe fondamental

### Principe fondamental

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

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

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

La correction du programme résultant n'est pas garantie !

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

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

# Notion de sous-typage pour Java

## Principe fondamental – explications



Pourquoi ce remplacement ne gène pas l'exécution :

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

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

# Notion de sous-typage pour Java

**Corollaires :**

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

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

# Transtypage (*type casting*)

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

Plusieurs mécanismes [1] [2] :

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

---

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

# Transtypage (*type casting*)

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

Parfois, conséquences à l'exécution :

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

# Transtypage (*type casting*)

## Autres termes employés (notamment dans la JLS)



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

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

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

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

# Transtypage (*type casting*)

## Autres termes employés (notamment dans la JLS)



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

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

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

# Transtypage, cas 1 : *upcasting*

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

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

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

# Transtypage, cas 2 : *downcasting*



***Downcasting* :**

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

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

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

# Transtypage, cas 2 : *downcasting*

## Avertissement



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

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

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

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

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

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

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

## Types “emballés”



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

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

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

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

## Conversions automatiques, depuis Java 5



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

Exemple :

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

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

# Transtypage : à propos du “*cast*”

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



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

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

# Subtilités du transtypage

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

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

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

# Subtilités du transtypage

## Cas des types primitfs : points d'implémentation

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

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

# Subtilités du transtypage

## Cas des types primitfs : points d'implémentation



Et concrètement, que font les conversions ?

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

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

# Subtilités du transtypage

## Cas des types primitfs : exemples

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

# Subtilités du transtypage

## Cas des types référence



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

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

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

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

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

# Subtilités du transtypage

## Cas des types référence

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

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

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

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

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

# Polymorphisme

## Un principe fondamental de la POO



### Definition (Polymorphisme)

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

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

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

# Polymorphisme

## Exemples et mécanismes divers



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

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

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

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

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

# Formes de polymorphisme

## Les 3 formes de polymorphisme en Java

- **polymorphisme *ad hoc*** (via la surcharge) : le même code recompilé dans différents contextes peut fonctionner pour des types différents.
  Attention : résolution à la compilation → après celle-ci, type concret fixé.
  Donc pas de réutilisation du code compilé → forme très faible de polymorphisme.

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

- **polymorphisme paramétré** :
  Concerne le code utilisant les **type génériques** (ou paramétrés, cf. généricité).
  Le même code peut fonctionner, sans recompilation [1], quelle que soit la concrétisation des paramètres.
  Ce polymorphisme permet d'exprimer des relations fines entre les types.

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

# Surcharge

## Définition

**Surcharge** = situation où existent plusieurs définitions (au choix)

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

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

**Remarques :**

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

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

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

# Surcharge

## Résolution : définitions préalables

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

# Surcharge

## Résolution (simplifiée... )

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

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

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

# Surcharge

## Exemples



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

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

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

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

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

# Surcharge

## Pourquoi elle ne permet que du polymorphisme “faible”



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

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

# Surcharge

## Pourquoi elle ne permet que du polymorphisme “faible”



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

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

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

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

# Interfaces

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

**Plusieurs façons de voir la notion d'interface :**

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

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

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

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

# Interfaces : syntaxe de la déclaration

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

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

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

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

# Interfaces : syntaxe de la déclaration

## Pourquoi ces contraintes ?



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

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

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

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

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

# Interfaces : usage (1)



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

class Mot implements Comparable {
  private String contenu;

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

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

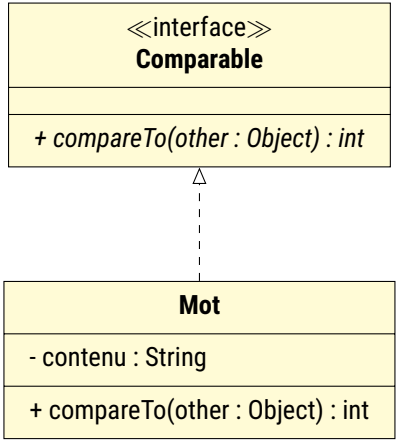
# Interfaces : usage (2)

## une méthode de tri polymorphe

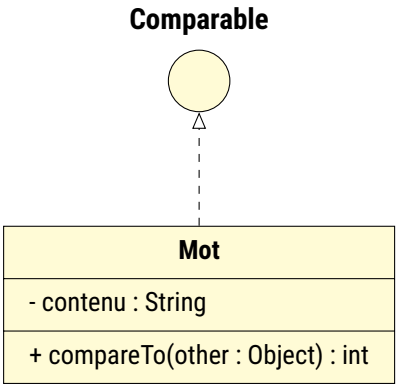


```
public class Tri {
    static void trie(Comparable [] tab) {
    /* ... algorithme de tri
        utilisant tab[i].compareTo(tab[j])
        ...
    */
    }

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

# Interfaces : notation UML




ou version ”abrégée” :




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

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

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

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

# Interfaces : méthodes par défaut

## Exemple



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

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

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

# Héritage d'implémentations multiples

## À cause des méthodes par défaut des interfaces



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

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

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

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

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

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

# Héritage d'implémentations multiples

## À cause des méthodes par défaut des interfaces



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

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

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

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

class A extends B implements I {}
```

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

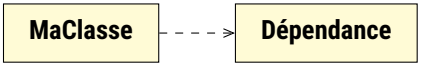
# Du bon usage des interfaces

## Programmez à l'interface (1)

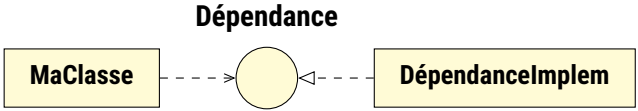


Évitez d'écrire, dans votre programme, le nom [1] des classes des objets qu'il utilise.

Cela veut dire, évitez :




et préférez :




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

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

# Du bon usage des interfaces

## Programmez à l'interface (2)



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

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

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

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

# Du bon usage des interfaces

## Programmez à l'interface (3)



**Pourquoi programmer à l'interface :**

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

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

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

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

# Du bon usage des interfaces

## Fabriques abstraites



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

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

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

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

# Du bon usage des interfaces

## Fabriques abstraites à l'âge des lambdas

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

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

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

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

# Du bon usage des interfaces

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

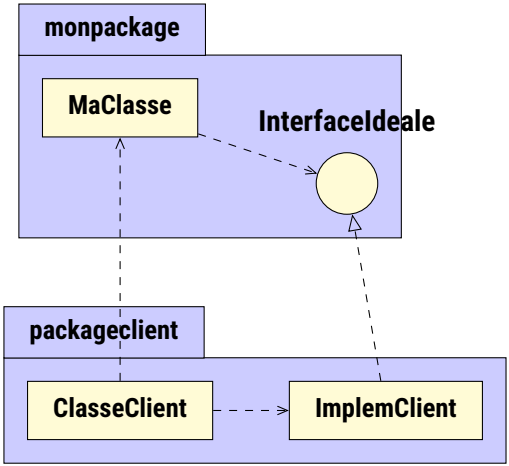
- **Quand ?** quand on programme une bibliothèque dependant d'un certain composant et qu'il n'existe pas d'interface « standard » décrivant exactement les fonctionnalités de celui-ci. [1].
- **Quoi ?** → on définit alors une interface idéale que la dépendance devrait implémenter et on la joint au *package* [2] de la bibliothèque.
  Les utilisateurs de la bibliothèque auront alors charge d'implémenter cette interface [3] (ou de choisir une implémentation existante) pour fournir la dépendance.

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

# Du bon usage des interfaces

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






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

# Du bon usage des interfaces

## Principe d'inversion de dépendance (3)

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

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

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

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

# Du bon usage des interfaces

## Le patron « adaptateur » (GoF) (1)



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

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

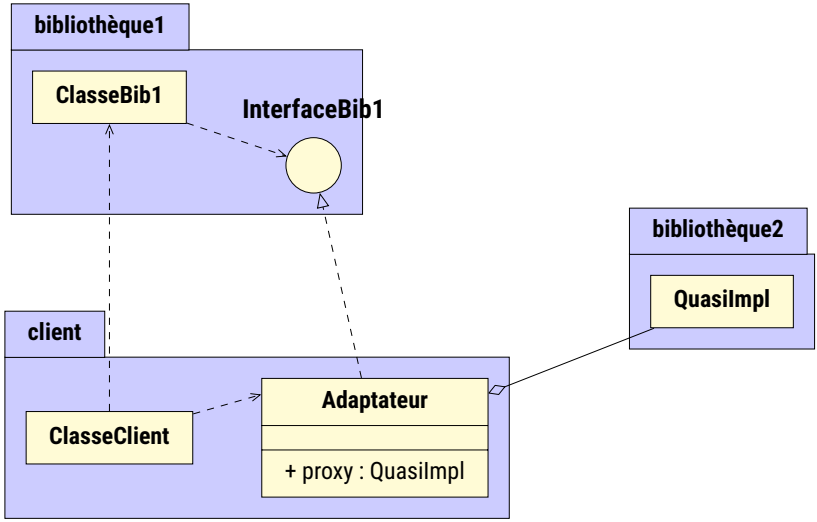
et dans laquelle les méthodes de `I` sont implémentées [1] par des appels de méthodes sur `proxy`.

1. De préférence très simplement et brièvement...

# Du bon usage des interfaces

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





# Héritage

## Obtenir réutilisation et sous-typage avec élégance



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

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

# Héritage – syntaxe…

… par l'exemple

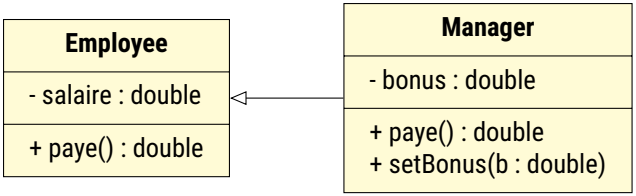


```java
class Employee {
    private double salaire;
    public Employee(double s) { salaire = s; }
    public double paye() { return salaire; }
}

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

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

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

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

# Héritage

## Notation UML






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

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

# Héritage

## Folklore et « piliers » de la POO, avertissement



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

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

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

---

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

# Extension vs. implémentation

**En POO, en théorie :**

- implémentation d'interface $\leftrightarrow$ 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.

# Héritage

## De classe ou d'interface ?



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

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

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

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

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

## Comparatif des 4 relations



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

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

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

# Héritage : vocabulaire

## Héritage direct ou extension

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

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

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

# Héritage : vocabulaire

## Héritage généralisé



**Héritage** (sous-entendu : « généralisé ») :

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

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

# Héritage : la racine

## La classe `Object`



La classe `Object` est :

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

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

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

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

# Héritage : la racine

## La classe `Object` : ses méthodes (1)



`Object` possède les méthodes suivantes :

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

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

# Héritage : la racine

## La classe `Object` : ses méthodes (2)



**Conséquences :**

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

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

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

# Héritage : ajout, redéfinition, masquage

## Le panorama... (1)



Dans une sous-classe :

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

...

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

# Héritage : ajout, redéfinition, masquage

## Le panorama... (2)



...

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

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

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

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

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

# Héritage : ajout, redéfinition, masquage

## Remarques diverses (1)



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

# Héritage : ajout, redéfinition, masquage

## Remarques diverses (2)



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

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

# Héritage : ajout, redéfinition, masquage

## Exemple



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

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

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

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

# Héritage : ajout, redéfinition, masquage

## Cas tordu



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

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

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

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

$\rightarrow$ Contradiction apparente avec ce qui avait été dit.

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

# Liaisons statique et dynamique

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



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

**À l'exécution** :

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

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

# Liaisons statique et dynamique

## Exemple



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

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

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

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

# Liaison statique

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



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

Pour l'explication, nous distinguons cependant :

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

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

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

# Liaison statique

## Principe : cas sans surcharge (membres non-méthode)



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

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

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

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

# Liaison statique

## Principe : cas des méthodes

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

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

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

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

# Liaison dynamique

## Principe



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

→ Principe de la liaison dynamique :

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

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

# Liaison dynamique

## Table virtuelle (implémentation optimisée de la liaison dynamique)



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

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

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

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

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

# Liaison dynamique

## Cas simple avec redéfinition



- Classe dérivée : certaines méthodes peuvent être redéfinies

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

- mais aussi...

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

# Liaison dynamique

## Cas compliqué, question ?



Imaginons le cas suivant, avec redéfinition et surcharge :

```
class Y1 {}

class Y2 extends Y1 {}

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

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

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

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

Qu'est-ce qui s'affiche ?

# Liaison dynamique

## Cas compliqué, réponse !

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

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

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

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

# Liaison dynamique

## Retour sur redéfinition et surcharge



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

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

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

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

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

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

1. même pas un masquage

# Méthodes et classes finales

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

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

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

Aussi possible :

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

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

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

# Méthodes et classes abstraites

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

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

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

# Méthodes et classes abstraites

## Exemple

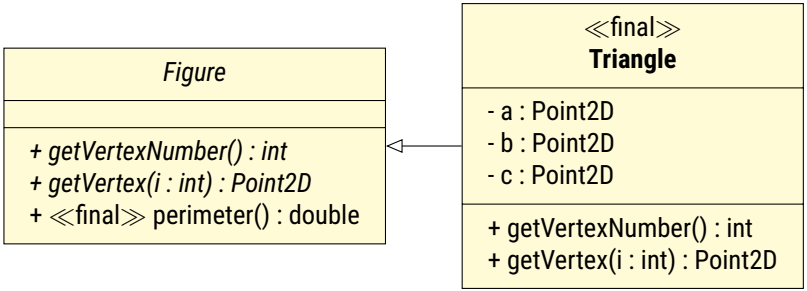


```
abstract class Figure {
    Point2D centre; String nom; // autres attributs éventuellement
    public abstract int getVertexNumber();
    public abstract Point2D getVertex(int i);
    public final double perimeter() {
        double peri = 0;
        Point2D courant = getVertex(0);
        for (int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

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

# Méthodes et classes finales ou abstraites

## Notation UML






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

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

# abstract et final

## le bon usage pour les classes (1)



- **`abstract`** et **`final`** contraignent la façon dont une classe s'utilise.
- Pourquoi contraindre ? → pour empêcher une utilisation incorrecte non prévue (cf. exemples ci-après).

Plus précisément :

- **`final`**, en figeant les méthodes (une ou toutes) d'une classe, permet d'assurer des propriétés qui resteront vraies pour toutes les instances de la classe.
- **`abstract`** (appliqué à une classe [1]) empêche qu'une classe qui ne représenterait qu'une implémentation incomplète ne soit instanciée directement.

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

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

# abstract et final

## le bon usage pour les classes (2)

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

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

Si cela est vrai, alors une classe ni finale ni abstraite est louche ! [2]. Comme ~~`abstract final`~~ est exclus d'office, toute classe devrait alors être soit (juste) `abstract` soit (juste) `final`.

| | pas `abstract` | `abstract` |
|---|---|---|
| pas `final` | louche | OK |
| `final` | OK | interdit |

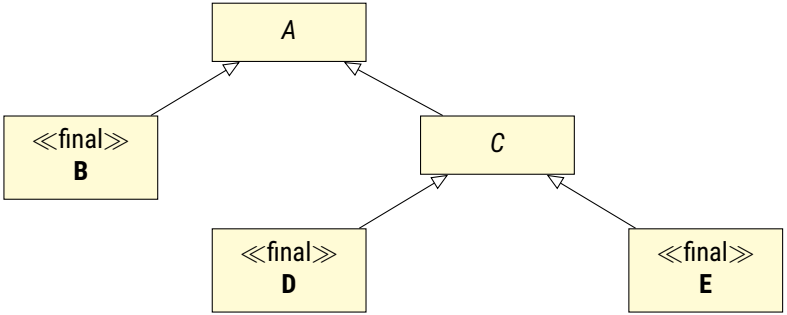
1. Ce n'est pas toujours vrai : certaines classes proposent un comportement par défaut tout à fait valable, tout en laissant la porte ouverte à des modifications (cf. composants Swing).
2. On parle de *code smell*. Cela dit, c'est « louche », mais pas absurde, cf. remarque précédente.

# abstract et final

## le bon usage pour les classes : illustration en UML



En UML, une bonne structure d'héritage donnerait des diagrammes comme celui-ci :



# abstract et final

## le bon usage pour les classes : exemple



### Exemple (à ne pas faire !) :

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

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

### Mieux :

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

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

# abstract et final

## le bon usage pour les méthodes : contre-exemple



À ne pas faire non plus :

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

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

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

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

**Récursion non bornée !** → `StackOverflowError`.

# abstract et final

## le bon usage pour les méthodes

**Quand on programme une classe extensible :**

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

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

# abstract et final

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

On entend souvent dire « L'héritage casse l'encapsulation. » (mais c'est exagéré).

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

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

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

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

1. À commencer par l'évidence : les membres `protected`. Mais même cela ne suffit pas.

# abstract et final

## le bon usage : résumé

**Une stratégie simple et extrême :**

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

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

# Application : les types scellés

## Définition et intérêt



- **Type scellé** : type dont l'ensemble des sous-types est fixé à la compilation de celui-ci.
- **Utilité :**
  - Il devient possible de prouver que les contrats du type sont bien implémentés pour toutes les instances présentes et futures : il suffit de le prouver pour un ensemble de cas fini et déjà connu.
  - Les fameuses listes de `else if (... instanceof ...){ ... }` deviennent plus acceptables car l'exhaustivité est garantie. Cela est utile quand la liaison dynamique ne peut pas être utilisée [1].
- **Exemples :** une classe `final`, n'ayant pas de sous-type du tout, est clairement scellée. De même, toute `enum` (étant par définition un type fini).

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

# Application : les types scellés

## Les types scellés de Java

Plus souple : classes à constructeurs tous privés. Une telle classe est instanciable et extensible seulement depuis l'intérieur de son corps ou de celui de son type englobant.

**Théorème :** [1] en Java [2], les types scellés sont exactement ceux définis par une classe `final` ou à constructeurs privés telle que ses sous-classes directes sont aussi scellées.

**Preuve :**

- $\Leftarrow$ Par récurrence, toute l'arborescence de sous-types est imbriquée dans un même type englobant, donc définie dans le même fichier `.java`. Il n'est donc pas possible d'ajouter un sous type sans modifier le fichier et le recompiler.
- $\Rightarrow$ Réciproquement, si une classe n'est pas `final` et a un constructeur non privé, on peut alors l'étendre depuis un autre fichier.

1. Correct si on considère que les classes privées et locales sont à constructeurs privés.
2. Dans d'autres langages (Scala, Kotlin), un mot-clé **sealed** permet de déclarer un type scellé sans bricoler avec les constructeurs. Ce mot-clé est maintenant en *preview* dans Java 15, cf. JEP 360.

# Application : les types scellés

## Exemple typique : type algébrique



```java
public abstract class BoolExpr { // classe scellée (et abstraite !)
    public abstract boolean eval();
    private BoolExpr() {}

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

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

    public static final class And extends BoolExpr {
        private final BoolExpr subexpr1, subexpr2;
        public And(BoolExpr subexpr1, BoolExpr subexpr2) { this.subexpr1 = subexpr2; this.subexpr2 =
            subexpr2; } // même remarque
        @Override public boolean eval() { return subexpr1.eval() && subexpr2.eval(); }
    }

    public static final class Or extends BoolExpr {
        private final BoolExpr subexpr1, subexpr2;
        public Or(BoolExpr subexpr1, BoolExpr subexpr2) { this.subexpr1 = subexpr2; this.subexpr2 =
            subexpr2; } // même remarque
        @Override public boolean eval() { return subexpr1.eval() || subexpr2.eval(); }
    }
}
```

# Application : les types scellés

## Exemple typique : type algébrique, version non objet



```
public abstract class BoolExpr {
    public static boolean eval(BoolExpr expr) { // en vrai, la version précédente était mieux
        if (expr instanceof Var) return ((Var) expr).value;
        else if (expr instanceof Not) return !eval(((Not) expr).subexpr);
        else if (expr instanceof And) return eval(((And) expr).subexpr1) && eval(((And) expr).subexpr2);
        else if (expr instanceof Or) return eval(((Or) expr).subexpr1) || eval(((Or) expr).subexpr2);
        else { assert false : "Cannot␣happen :␣the␣pattern␣matching␣is␣exhaustive!"; return false; }
    }
    private BoolExpr() {}

    public static final class Var extends BoolExpr {
        private final boolean value;
        public Var(boolean value) { this.value = value; }
    }

    public static final class Not extends BoolExpr {
        private final BoolExpr subexpr;
        public Not(BoolExpr subexpr) { this.subexpr = subexpr; }
    }

    public static final class And extends BoolExpr {
        private final BoolExpr subexpr1, subexpr2;
        public And(BoolExpr subexpr1, BoolExpr subexpr2) { this.subexpr1 = subexpr2; this.subexpr2 =
            subexpr2; }
    }

    public static final class Or extends BoolExpr {
        private final BoolExpr subexpr1, subexpr2;
        public Or(BoolExpr subexpr1, BoolExpr subexpr2) { this.subexpr1 = subexpr2; this.subexpr2 =
            subexpr2; }
    }
```

# Application : les types scellés

## Exemple typique : type algébrique, version Java 15 preview



```java
public sealed interface BoolExpr { // sealed, pas besoin de constructeur privé ! (et interface autorisée)
    static boolean eval(BoolExpr expr) {
        if (expr instanceof Var varExpr) return varExpr.value; // on profite du nouveau pattern matching
        else if (expr instanceof Not notExpr) return !eval(notExpr.subexpr);
        else if (expr instanceof And andExpr) return eval(andExpr.subexpr1) && eval(andExpr.subexpr2);
        else if (expr instanceof Or orExpr) return eval(orExpr.subexpr1) || eval(orExpr.subexpr2);
        else { assert false : "Cannot happen : the pattern matching is exhaustive!"; return false; }
    }

    public static final class Var implements BoolExpr {
        private final boolean value;
        public Var(boolean value) { this.value = value; }
    }

    public static final class Not implements BoolExpr {
        private final BoolExpr subexpr;
        public Not(BoolExpr subexpr) { this.subexpr = subexpr; }
    }

    public static final class And implements BoolExpr {
        private final BoolExpr subexpr1, subexpr2;
        public And(BoolExpr subexpr1, BoolExpr subexpr2) { this.subexpr1 = subexpr2; this.subexpr2 =
              subexpr2; }
    }

    public static final class Or implements BoolExpr {
        private final BoolExpr subexpr1, subexpr2;
        public Or(BoolExpr subexpr1, BoolExpr subexpr2) { this.subexpr1 = subexpr2; this.subexpr2 =
              subexpr2; }
    }
}
```

# Application : les types scellés

## Exemple typique : type algébrique, avec pattern-matching switch (futur)



```java
public sealed interface BoolExpr {
    static boolean eval(BoolExpr expr) {
        return switch(expr) {
            case Var varExpr -> varExpr.value;
            case Not notExpr -> !eval(notExpr.subexpr);
            case And andExpr -> eval(andExpr.subexpr1) && eval(andExpr.subexpr2);
            case Or orExpr -> eval(orExpr.subexpr1) || eval(orExpr.subexpr2);
        } // liste exhaustive, pas besoin de cas default !
    }

    public static final class Var implements BoolExpr {
        private final boolean value;
        public Var(boolean value) { this.value = value; }
    }

    public static final class Not implements BoolExpr {
        private final BoolExpr subexpr;
        public Not(BoolExpr subexpr) { this.subexpr = subexpr; }
    }

    public static final class And implements BoolExpr {
        private final BoolExpr subexpr1, subexpr2;
        public And(BoolExpr subexpr1, BoolExpr subexpr2) { this.subexpr1 = subexpr2; this.subexpr2 =
            subexpr2; }
    }

    public static final class Or implements BoolExpr {
        private final BoolExpr subexpr1, subexpr2;
        public Or(BoolExpr subexpr1, BoolExpr subexpr2) { this.subexpr1 = subexpr2; this.subexpr2 =
            subexpr2; }
    }
```

# Types finis

## Pour quoi faire ?

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

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

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

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

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

# Types finis

## Pour quoi faire ?

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

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

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

---

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

# Écrire un type fini

## Comment faire ?

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

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

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

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

# Classes d'énumération

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

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

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

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

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

# Énumérations

## La base



### Exemple simple :

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

public class Test {
    public static void main(String[] args) {
        for (Day d : Day.values()) {
            switch (d) {
            case SUNDAY:
            case SATURDAY:
                System.out.println(d + ": sleep");
                break;
            default:
                System.out.println(d + ": work");
            }
        }
    }
}
```

# Énumération = classe particulière

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

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

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

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

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

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

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

# Énumérations

## Extensions



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

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;
    public boolean isWorkDay() {
        switch (this) {
        case SUNDAY:
        case SATURDAY:
            return false;
        default:
            return true;
        }
    }
    public static void main(String[] args) {
        for (Day d : Day.values()) {
            System.out.println(d + ": " + (d.isWorkDay() ? "work" : "sleep"));
        }
    }
}
```

# Énumérations

## Extensions



On peut même ajouter des constructeurs (privés seulement). Auquel cas, il faut passer les paramètres du(d'un des) constructeur(s) à chaque constante de l'enum :

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

# Énumérations

## Extensions



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

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

    public boolean isWorkDay() { return true; }

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

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

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

# Énumérations

## Types énumérés et sous-typage



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

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

# Énumérations

## Méthodes



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

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

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

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

# Énumérations

## Quand les utiliser

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

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

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

# Énumérations

## Bien les utiliser

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

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

# Énumérations et collections (1)



Il existe des implémentations de collections optimisées pour les énumérations.

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

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

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

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

# Énumérations et collections (2)

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

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

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

# Autour de l'héritage

## Le bon usage de l'héritage de classe (1)

On a coutûme de dire que :

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

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

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

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

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

# Autour de l'héritage

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

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

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

Mais...

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

# Autour de l'héritage

## Limites de l'héritage de classe et avertissements



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

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

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

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

Pour des objectifs simples, préférer des techniques alternatives :

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

1. Obligatoire et assumé dans des langages comme Rust, où l'héritage ne crée pas de sous-typage.
2. EJ3 20 : « *Prefer interfaces to abstract classes* »
3. Faux en cas de méthodes `default` → même besoin de documentation que pour l'héritage de classe.
4. EJ3 18 : « *Favor composition over inheritance* »

# Autour de l'héritage

## Alternative : ce qu'on entend par « composition »

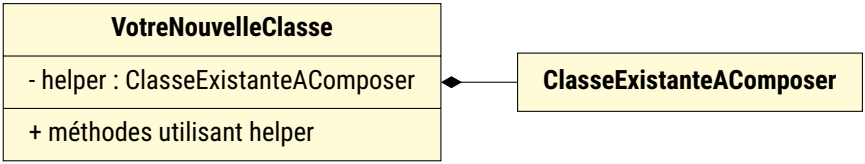


**Composition** : utilisation d'un objet à l'intérieur d'un autre pour réutiliser les fonctionalités codées dans le premier. Exemple :

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

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

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

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

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

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




Notez le losange plein.

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

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

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

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

## Exemple problématique



Mathématiquement les entiers sont sous-type des rationnels. Mais comment le coder ?

**Pas terrible :**

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

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

→ utilisation trop importante de mémoire (pas très grave)

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

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

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

## Modélisation à l'aide d'interfaces



**Mieux :**

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

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

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

## Modélisation à l'aide de classes scellées



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

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

```
public abstract class Nombre {
    private Nombre() {} // Scellage !
    // hiérarchie publique
    public static abstract class Rationnel extends Nombre { private Rationnel() {} }
    public static abstract class Entier extends Rationnel { private Entier() {} }
    // implémentations immuables
    private static final class RationnelImpl extends Rationnel {
        final int numerateur, denominateur;
        RationnelImpl(int p, int q) { numerateur = p; denominateur = q; }
    }
    private static final class EntierImpl extends Entier {
        final int valeur;
        EntierImpl(int valeur) { this.valeur = valeur; }
    }
    // fabriques statiques
    public static Rationnel rationnel(int p, int q) { return new RationnelImpl(p, q); }
    public static Entier entier(int n) { return new EntierImpl(n); }
} // il faudra bien sûr ajouter getters et opérations arithmétiques dans toutes les classes
```

Les types publics ont alors la bonne relation de sous-typage et leur scellage garantit leur immuabilité.

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

## Exemple problématique



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

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

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

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

Solutions (1)



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

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

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

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

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





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

   **Point fort :** sans faille et très facile

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

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

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

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

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

## Le patron décorateur (1)



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

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

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

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

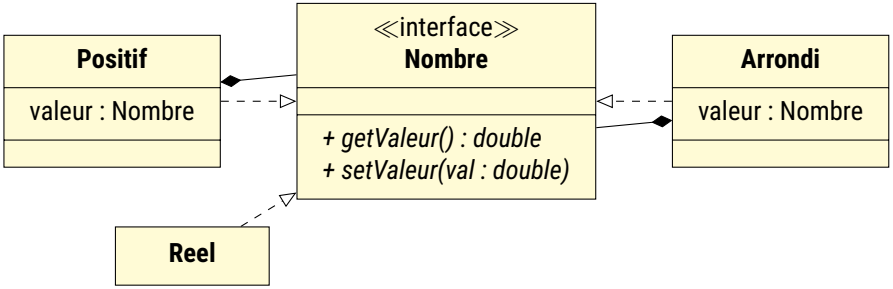
# Objectif : ne pas laisser l'héritage casser l'encapsulation

## Le patron décorateur (2)



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

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




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

# Comment « hériter » de plusieurs classes ?

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

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

# Objectif : simuler un héritage multiple

## Via les interfaces et la composition → patron « délégation »

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

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

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

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

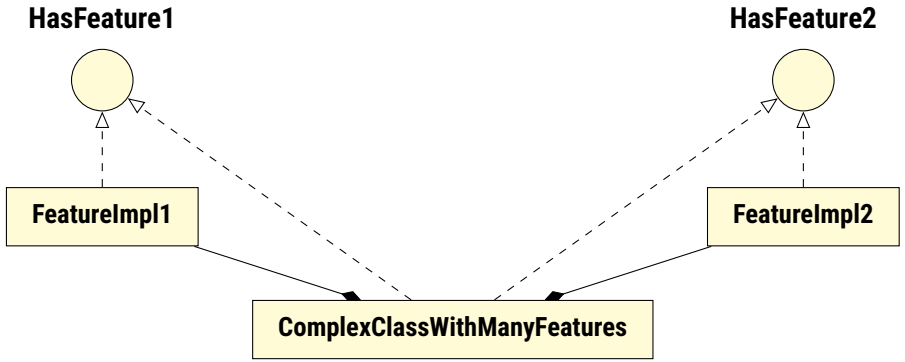
```
class PossedePropAetB implements AvecPropA, AvecPropB {
    PossedePropA aProxy; PossedePropB bProxy;
    void setA(int newA) { aProxy.setA(newA); }
    int getA() { return aProxy.getA(); }
    void setB(int newB) { bProxy.setB(newB); }
    int getB() { return bProxy.getB(); }
}
```

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

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

# Patron « délégation » en UML

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




Ce qu'on ne voit pas sur le diagramme : les implémentations dans `ComplexClassWithManyFeatures` des méthodes de `HasFeatureX` ne comportent qu'un simple appel vers la méthode de `FeatureImplX` de même nom.

## Exemple (de l'API) :

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

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

## Types génériques du JDK :

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

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

# Généricité

## Pourquoi ? (1)



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

**Sur un exemple :**

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

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

**Réutilisabilité :** proche de zéro !

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

# Généricité

## Pourquoi ? (2)



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

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

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

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

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

**Cas d'utilisation problématique :**

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

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

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

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

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

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

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

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

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

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

# Types génériques et types paramétrés

## Usage de base (1)



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

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

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

# Types génériques et types paramétrés

## Usage de base (2)



- **À l'usage**, le type générique sert de constructeur de type : on remplace le paramètre par un type concret et on obtient un **type paramétré**.
  Exemple : `List` est un type générique, `List<String>` un des types paramétrés que `List` permet de construire.
- Le type concret substituant le paramètre doit être un type **référence** : ~~`Triplet<int, boolean, char>`~~ est interdit [1] !

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

# Types génériques et types paramétrés

## Usage de base (3)



- Utilisation de classe générique par instanciation directe :

```java
// à  partir de Java 5 :
Triplet<String, String, Integer> t1 =
    new Triplet<String, String, Integer>("Marcel", "Durand", 23);

// à  partir de Java 7 :
Triplet<String, String, Integer> t2 = new Triplet<>("Marcel", "Durand", 23);

// à  partir de Java 10 (si t3 est une variable locale) :
var t3 = new Triplet<String, String, Integer>("Marcel", "Durand", 23);
```

Le type de `t1`, `t2` et `t3` est le type paramétré `Triplet<String, String, Integer>`.

# Types génériques et types paramétrés

## Usage de base (4)



- Utilisation de classe générique par extension non générique (**spécialisation**) :

```
class TroisChars extends Triplet<Char, Char, Char> {
  public TroisChars(Char x, Char y, Char z) { super(x,y,z); }
}
```

`TroisChars` étend la classe paramétrée `Triplet<Char, Char, Char>`.

- Variante, spécialisation partielle :

```
class DeuxCharsEtAutre<T> extends Triplet<Char, Char, T> {
  public DeuxCharsEtAutre(Char x, Char y, T z) { super(x,y,z); }
}
```

La classe générique `DeuxCharsEtAutre<T>` étend la classe générique partiellement paramétrée `Triplet<Char, Char, T>`.

# Parallèle entre type générique et méthode

La déclaration et l'utilisation des types génériques rappellent celles des méthodes.

**Similitudes :**

- introduction des paramètres (de type ou de valeur) dans l'en-tête de la déclaration ;
- utilisation des noms des paramètres dans le corps de la déclaration seulement ;
- pour utiliser le type générique ou appeler la méthode, on passe des concrétisations des paramètres.

**Principales différences :**

- Les paramètres des génériques représentent des types alors que ceux des méthodes représentent des valeurs.
- Pour les paramètres de type, le « remplacement »[1] a lieu à la compilation.
  Pour les paramètres des méthodes, remplacement par une valeur à l'exécution.

1. Rappel : remplacement oublié, effacé, aussitôt que la vérification du bon typage a été faite.

# Types bruts

- Un nom de type générique seul, sans paramètre (comme « `Triplet` »), est aussi un type légal, appelé un **type brut** (***raw type***).
  Son utilisation est **fortement déconseillée**, mais elle est permise pour assurer la compatibilité ascendante [1].
- Un type brut est supertype direct [2] de tout type paramétré correspondant (ex : `Triplet` est supertype direct de `Triplet<Number, Object, String>`).
- Pour faciliter l'écriture, le *downcast* implicite [3] est malgré tout possible :

```
List l1 = new ArrayList(); // déclaration de l1 avec raw type
List<Integer> l2 = l1; // downcast implicite de l1 vers type paramétré
```

compile avec l'avertissement `unchecked conversion` sur la deuxième ligne.

1. Un source Java < 5 compile avec `javac` $\geq$ 5. Or certains types sont devenus génériques entre temps.
2. C'est une des règles de sous-typage relatives aux génériques, omises dans le début de ce cours.
3. Je crois que c'est l'unique occurrence de *downcast* implicite en Java.

# Déclaration de paramètre de type niveau méthode



- Il est aussi possible d'introduire un paramètre de type dans la signature d'une méthode (possible aussi dans une classe non générique) :

```
static <E> List<E> inverseListe(List<E> l) { ... ; E x = get(0); ...; }
```

- Dans l'exemple ci-dessus, on garantit que la liste retournée par `inverseListe()` a le même type d'éléments que celle donnée en paramètre.
- **Usages possibles :**
  - contraindre plusieurs types apparaissant dans la signature de la méthode à être le même type, sans pour autant dire lequel ;
  - introduire localement un nom de type utilisable dans le corps de la méthode (type non défini, mais dont les contraintes sont connues, ex : type intersection, voir plus loin).

**Remarque :** il est donc possible d'écrire une méthode statique générique et son corps (contexte statique) pourra utiliser le paramètre introduit, contrairement aux paramètres de type introduits au niveau de la classe ou de l'interface.

# Bornes de paramètres de type (1)

- Pour limiter les concrétisations autorisées, un paramètre de type admet des **bornes supérieures** [1] (se comportant comme supertypes du paramètre) :

```
class Calculator<Data extends Number>
```

Ici, `Data` devra être concrétisé par un sous-type de `Number` : une instance de `Calculator` travaillera sur nécessairement avec un certain sous-type de `Number`, celui choisi à son instanciation.

1. On verra dans la suite que les bornes inférieures existes aussi, mais elles ne s'appliquent qu'aux *wildcards* (et non aux paramètres de type).

# Bornes de paramètres de type (2)

- Pour définir des bornes supérieures multiples (p. ex. pour implémenter de multiples interfaces), les supertypes sont séparés par le symbole « & » :

```
class RunWithPriorityList<T extends Comparable<T> & Runnable> implements List<T>
```

« `Comparable<T> & Runnable` » est un **type intersection** [1], il est sous-type direct de `Comparable<T>` et de `Runnable`.

Ainsi, `T` est sous-type de l'intersection (et donc de de `Comparable<T>` et de `Runnable`).

1. Remarque : c'est le seul contexte où on peut écrire un type intersection (type non dénotable). Ainsi, il n'est pas possible de déclarer explicitement une variable de type intersection. Implicitement, à l'aide d'une méthode générique et du mot-clé **`var`**, cela est cependant possible :
`public static <T extends A & B > T intersectionFactory(...){ ... }`
plus loin :
`var x = intersectionFactory(...); //x est de type A & B`
La technique assez « tirée par les cheveux » et d'utilité toute relative... .

# Bornes de paramètres de type (3)



On peut prolonger l'analogie avec les méthodes et leurs paramètres : en effet, les paramètres des méthodes sont eux-mêmes « bornés » par les types déclarés dans la signature.

# Effacement de type (*type erasure*)

## De quoi il s'agit.



**Effacement d'un type** : sur-approximation permettant d'obtenir un **type réifiable** (i.e. : « classique », façon Java 4) à partir de n'importe quel type. Plus précisément (JLS 4.6) :

- L'effacement d'un type générique ou paramétré de forme `G<...>`, est le type brut `G`.
- L'effacement d'une variable de type est l'effacement de sa borne supérieure.
- L'effacement de tout autre type `T` est `T`.

L'idée principale du phénomène appelé **effacement de type** (ou ***type erasure***) c'est que le système de types de la JVM ne connait que les types réifiables.
Autrement dit : la paramétrisation générique n'a pas d'impact à l'exécution.

Plus de détails juste après.

# Effacement de type (*type erasure*)

## Explication



L'effacement de type commence en réalité dès la compilation.

**Descripteur de méthode :** information, dans la **table des constantes** d'une classe compilée, permettant d'identifier une méthode (peut-être surchargée) de façon unique. Tout appel de méthode [1] dans le code-octet fait référence à un tel descripteur.

Or un descripteur consiste en un couple : (nom de méthode, types réifiables des paramètres).

**Conséquences :**

- aucun paramètre de type n'est réellement passé aux constructeurs (et méthodes)
- les objets ne stockent donc pas les valeurs de leurs paramètres de types. Ils ne peuvent donc connaître que leur classe [2]. Les « objets paramétrés » n'existent pas.

1. `invokestatic`, `invokespecial`, `invokevirtual` et `invokeinterface` prennent un index de la table des constantes comme paramètre.
2. Qui n'existe qu'en un seul exemplaire dans la mémoire, quelle que soit la parmétrisation.

# Effacement de type (*type erasure*)

## Explication



**Remarque :** le code-octet contient tout de même encore des types non réifiables. C'est le cas pour les « signatures [1] » des classes et de leurs membres. Cela est indispensable pour vérifier les types génériques lors de la compilation des classes dépendantes.

Il est donc faux que le code-octet ne sait déjà plus rien de la paramétrisation générique d'une classe.

Mais cette information est à destination du compilateur, et non pas de la JVM [2].

1. Signature au sens de la JVM. Pour la JVM, la signature contient toute l'information de typage d'une entité donnée. Par exemple, pour une méthode, c'est son type de retour et les types de ses paramètres. Cette notion est donc différente de la notion de signature dans un code source Java. Elle est aussi différente de la notion de descripteur tout juste évoquée.
2. Ceci dit, il est possible de lire les signatures pendant l'exécution (grâce à la réflexion). Mais cela ne permet en aucun cas de savoir quels paramètres de types effectifs ont été utilisés pour instancier un objet donné ou exécuter une méthode donnée.

# Effacement de type (*type erasure*)

## Conséquences concrètes



À l'exécution, il est impossible de savoir comment un paramètre de type a été concrétisé.

**En particulier :**

- Faute d'être raisonnablement exécutables, ces expressions ne compilent pas :
  - `x instanceof P` (avec `P` paramètre de type) ;
  - `x instanceof TypeG<Y>` [1] (avec `Y` type quelconque).
- On ne peut pas déclarer d'exception générique `Ex<T>` car `catch( Ex<X> ex )` ne serait pas non plus évaluable (même problème qu'`instanceof`).

```
// ne compile pas
public class GenericException<T> extends Exception { ... }
```

1. Mais `x instanceof TypeG` compile, c'est un des rares cas où on tolère le *raw type*.

# Effacement de type (*type erasure*)

## Conséquences concrètes



Autres conséquences directes de l'effacement dans les descripteurs :

- Dans une classe, on ne peut pas définir plusieurs méthodes dont les signatures seraient identiques après effacement (leurs descripteurs seraient identiques).

```
// ne compile pas
public class A {
  List<Integer> f() { return null; }
  List<String> f() { return null; }
}
```

- Une classe ne peut pas implémenter plusieurs fois une interface générique avec des paramètres différents (on se retrouverait dans le cas précédent).

```
// ne compile pas
public class A extends ArrayList<Integer> implements List<String> { ... }
```

# Effacement de type (*type erasure*)

## Mais quand les paramètres de type servent-ils alors ?



La valeur concrète d'un paramètre est donc souvent inconnue à la compilation (code générique pas encore concrétisé) et toujours inconnue à l'exécution.

$\rightarrow$ alors à quoi servent les paramètres de type ?

$\rightarrow$ Un paramètre de type est juste un symbole formel permettant d'exprimer des énoncés logiques **que le compilateur doit prouver** avant d'accepter le programme.

Ceux-ci sont de la forme : $\forall T\,[(\forall B \in \mathit{UpperBounds}(T), T <: B) \Rightarrow \mathit{WellTyped}(\mathit{prog}(T))]$
où *prog* est soit une classe, soit une méthode générique.

# Collections : pourquoi ?

- **Besoin :** représenter des « paquets », des « collections » d'objets similaires.
- **Plusieurs genres de paquets/collections :** avec ou sans doublon, accès séquentiel ou aléatoire ( = par indice ), avec ou sans ordre, etc.
- **Mais nombreux points communs :** peuvent contenir plusieurs éléments, possibilité d'itérer, de tester l'appartenance, l'inclusion etc.
- Pour chaque « genre » plusieurs représentations/implémentations de la structure (optimisant telle ou telle opération...).

# Collections : pourquoi ?

Les collections génériques, introduites dans Java SE 5, remplacent avantageusement :

- Les tableaux [1].
- Les collections non génériques [2] de Java $< 5$, avec leurs éléments de type statique `Object`, qu'il fallait *caste*r avant usage.

  Ex. : classe `Vector` (listes implémentées par tableaux dynamiques synchronisés).

Les collections justifient à elles seules l'introduction de la généricité dans Java.

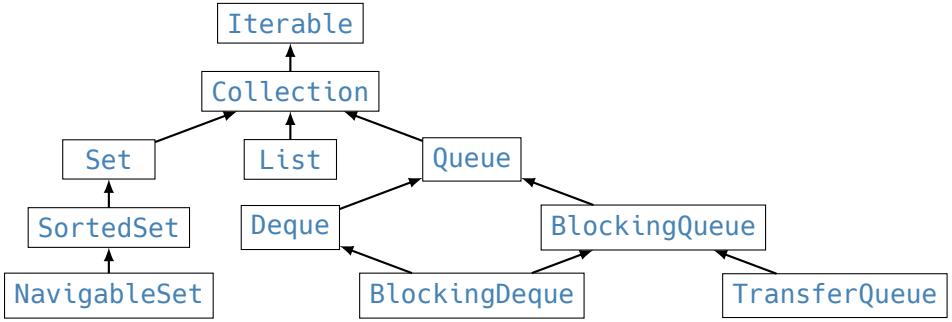
---

1. Qui gardent quelques avantages : syntaxe pratique, efficacité et disposent déjà d'un « genre de généricité » (un `String[]` contiendra des éléments `String` et rien d'autre), dont nous reparlerons.
2. **NB :** les anciennes collections ont été transformées en types génériques (`Vector<E>` au lieu de `Vector`) implémentant l'interface `Collection<E>`.
Les types sans paramètre sont désormais considérés comme des types bruts et sont **à éviter**.
Si vous migrez du code Java $< 5$ vers Java $\geq 5$, remplacez `ArrayList` par `ArrayList<TypeElems>`.

# La hiérarchie des sous-interfaces d'`Iterable`

Interfaces de `java.util` et `java.util.concurrent`[1]





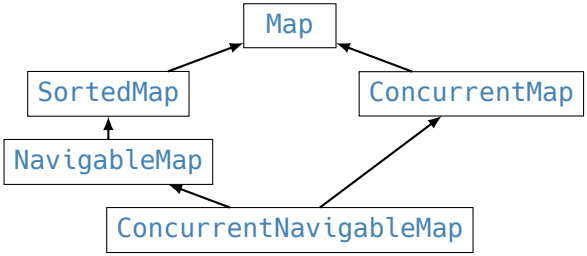

Chacune de ces interfaces possède une ou plusieurs implémentations.

1. Autres sous-interfaces dans `java.nio.file` et `java.beans.beancontext`.

# La hiérarchie des sous-interfaces de Map

## Interfaces de `java.util` et `java.util.concurrent`[1]






Chacune de ces interfaces possède une ou plusieurs implémentations.

1. Autres dans `javax.script`, `javax.xml.ws.handler` et `javax.xml.ws.handler.soap`.

# L'interface `Iterable` (1)



```
public interface Iterable<E> {
    Iterator<E> iterator(); // cf. Iterator
    default Spliterator<E> spliterator() { ... } // pour les Stream
    default void forEach(Consumer<? super T> action) { ... } // utiliser avec lambdas
}
```

Un `Iterable` représente une séquence qu'on peut « itérer » (à l'aide d'un **itérateur**).

- soit avec la construction « *for-each* » (conseillé !) :

```
for ( Object o : monIterable ) System.out.println(o);
```

- soit avec la méthode `forEach` et une lambda-expression (cf. chapitre dédié) :

```
monIterable.forEach(System.out::println);
```

# L'interface `Iterable` (2)

- soit en utilisant explicitement l'itérateur (rare, mais utile pour accès en écriture) :

```
Iterator<String> it = monIterable.iterator();
while (it.hasNext()) {
   String s = it.next();
   if (s.equals("À enlever") it.remove();
   else System.out.println("On garde: " + s);
}
```

Remarque : la construction *for-each* et la méthode `forEach` ne permettent qu'un parcours en lecture seule.

- soit en réduisant un `Stream` (cf. chapitre dédié) basé sur cet `Iterable`[1] :

```
maCollection.stream()
   .filter(x -> !x.equals("À enlever"))
   .forEach(System.out::println());
```

Les paramètres des méthodes de `Stream` sont typiquement des lambda-expressions.

1. En réalité, pour des raisons assez obscures, la méthode `stream` n'existe que dans la sous-interface `Collection`. Mais il est facile de programmer une méthode équivalente pour `Iterable`.

# Itérateurs ?

**Un itérateur :**

- sert à parcourir un itérable et est habituellement utilisé implicitement ;
- s'instancie en appelant la méthode `iterator` sur l'objet à parcourir ;
- est un objet respectant l'interface suivante :

```
public interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();            // opération optionnelle
}
```

`remove`, si implémentée, permet de supprimer un élément en cours de parcours sans provoquer `ConcurrentModificationException` (au contraire des méthodes de l'itérable). Cette possibilité justifie de créer une variable pour manipuler explicitement l'itérateur (sinon, on préfère *for-each*).

# L'interface `Collection`

Une `Collection` est un `Iterable` muni des principales opérations ensemblistes :

```
public interface Collection<E> extends Iterable<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          // opération optionnelle
    boolean remove(Object element);  // opération optionnelle
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);  // opération optionnelle
    boolean removeAll(Collection<?> c);         // opération optionnelle
    boolean retainAll(Collection<?> c);         // opération optionnelle
    void clear();                               // opération optionnelle
    Object[] toArray();
    <T> T[] toArray(T[] a);
    default Stream<E> stream() { ... }
}
```

L'API ne fournit pas d'implémentation directe de `Collection`, mais plutôt des collections spécialisées, décrites dans la suite.

# L'interface Set

- Pas de méthodes autres que celles héritées de `Collection`.
- **Différence :** le contrat de `Set` garantit l'**unicité** de ses éléments (pas de doublon).

**Exemple :**

```
Set<Integer> s = new HashSet<Integer>();
s.add(1); s.add(2); s.add(3); s.add(1);
for (int i : s) System.out.print(i + ", ");
```

Ceci affichera : `1, 2, 3,`
La classe `HashSet` est une des implémentations de `Set` fournies par Java. C'est celle que vous utiliserez le plus souvent.

**Unicité ?** un élément `x` est unique si pour tout autre élément `y`, `x.equals(y)` retourne `false`.
⇒ importance d'avoir une redéfinition correcte de `equals()`.

# L'interface `SortedSet`

Comme `Set`, mais les éléments sont triés.

... ce qui permet d'avoir quelques méthodes en plus.

```
public interface SortedSet<E> extends Set<E> {
    // Range-view
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);

    // Endpoints
    E first();
    E last();

    // Comparator access
    Comparator<? super E> comparator();
}
```

Implémentation typique : classe `TreeSet`.

# L'interface `List` (1)

`List` : c'est une `Collection` ordonnée avec possibilité de doublons. C'est ce qu'on utilise le plus souvent. Permet d'abstraire les notions de tableau et de liste chainée.

**Fonctionnalités principales :**

- acces positionnel (on peut accéder au *i*-ième élément)
- recherche (si on connait un élément, on peut demander sa position)

```
public interface List<E> extends Collection<E> {
    // Positional access
    E get(int index);
    E set(int index, E element);    //optional
    boolean add(E element);         //optional
    void add(int index, E element); //optional
    E remove(int index);            //optional
    boolean addAll(int index,
        Collection<? extends E> c); //optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);
...
```

# L'interface `List` (2)

**Mais aussi :**

- itérateurs plus riches (peuvent itérer en arrière)
- « vues » de sous-listes [1]

```
...

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```

1. Vue d'un objet `o` : objet `v` donnant accès à une partie des données de `o` sans en être une copie (partielle), les modifications des 2 objets restent liées.

# Itérateurs de liste ?

Un itérateur de liste sert à parcourir une liste. Il fait la même chose qu'un itérateur ; mais aussi quelques autres opérations, comme :

- parcourir à l'envers
- ajouter/modifier des éléments en passant
- un itérateur de liste est un objet respectant l'interface suivante :

```
public interface ListIterator<E> extends Iterator<E>{
  void add(E e);
  boolean hasPrevious();
  int nextIndex();
  E previous();
  int previousIndex();
  void set(E e);
}
```

# L'interface `List` (3)



Implémentations principales : `ArrayList` (basée sur un tableau, avec redimensionnement dynamique), `LinkedList` (basée sur liste chainée).

Exemple :

```
ArrayList<Integer> l = new ArrayList<Integer>();
l.add(1); l.add(2); l.add(3); l.add(1);
for (int i : l) System.out.print(i + ", ");
System.out.println("\n3e element: " + l.get(2));
l.set(2,9);
System.out.println("Nouveau 3e element: " + l.get(2));
```

Ceci affichera :

```
1, 2, 3, 1
3e element: 3
Nouveau 3e element: 9
```

# L'interface Queue



Une `Queue` représente typiquement une collection d'éléments en attente de traitement (typiquement FIFO : *first in, first out*).

Opérations de base : insertion, suppression et inspection.

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
    E peek();
    E poll();
    E remove();
}
```

Exemple : la classe `PriorityQueue` présente ses éléments selon l'ordre naturel de ses éléments (ou un autre ordre si spécifié).

# L'interface Deque (1)



`Deque` = *« double ended queue »*.

C'est comme une `Queue`, mais enrichie afin d'accéder à la collection aussi bien par le début que par la fin.

Le même `Deque` peut ainsi aussi bien servir de structure FIFO que LIFO (*last in, first out*).

```
public interface Queue<E> extends Collection<E> {
    boolean addFirst(E e);
    boolean addLast(E e);
    Iterator<E> descendingIterator();
    E getFirst();
    E getLast();
    boolean offerFirst(E e);
    boolean offerLast(E e);
    E peekFirst();
    E peekLast();
...
```

# L'interface Deque (2)



```
...
    E pollFirst();
    E pollLast();
    E pop();
    void push(E e);
    E removeFirst();
    E removeLast();
    E removeFirstOccurrence(Object o);
    E removeLastOccurrence(Object o);
    ...
    // plus methodes héritées
}
```

Implémentations typiques : `ArrayDeque`, `LinkedList`

# L'interface Map (1)



Une `Map` est un ensemble d'associations (clé $\mapsto$ valeur), où chaque clé ne peut être associée qu'à une seule valeur.

Nombreuses méthodes communes avec l'interface `Collection`, mais particularités.

```
public interface Map<K,V> {
   // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();
...
```

# L'interface Map (2)



```
...
   // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();
   // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();
   // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

Implémentation la plus courante : la classe `HashMap`

# L'interface `SortedMap`

`SortedMap` est à `Map` ce que `SortedSet` est à `Set` : ainsi les associations sont ordonnées par rapport à l'ordre de leurs les clés.

```
public interface SortedMap<K, V> extends Map<K, V>{
    Comparator<? super K> comparator();
    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();
}
```

Implémentation typique : `TreeMap`.

# Fabriques statiques

Fabriques statiques du JDK → alternative intéressante aux consructeurs de collections :

- Nombreuses dans la classe `Collections` [1] : collections vides, conversion d'un type de collection vers un autre, création de vues avec telle ou telle propriété, ...
- Pour obtenir une liste depuis un tableau : `Arrays.asList(tableau)`.
- Fabriques statiques de collections immuables, nommées « `of` », dans les interfaces `List`, `Set` et `Map` (Java $\geq$ 9) :

```
List<String> semaine = List.of("lundi", "mardi", "mercredi", "jeudi",
    "vendredi", "samedi", "dimanche");
Map<String, String> instruments = Map.of("guitare", "cordes", "piano", "cordes",
    "clarinette", "vent");
Set<Integer> premiers = Set.of(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31);
```

Appeler une fabrique plutôt qu'un constructeur évite de choisir une implémentation : on fait confiance à la fabrique pour choisir la meilleure pour les paramètres donnés.

1. Noter le 's'

# Collections et tableaux

## Quand les utiliser



**Avantages des tableaux :** syntaxe légère et efficacité.

**Avantages des collections génériques :**

- polyvalence (plein de collections adaptées à des cas différents)
- polymorphisme via les interfaces de collections
- sûreté du typage (« vraie » généricité)

Conclusion, **utilisez les collections**, sauf :

- si vous prototypez un programme très rapidement et vous appréciez la simplicité
- si vous souhaitez optimiser la performance au maximum [1] [2]

---

1. Cela dit, les méthodes du *collection framework*, sont écrites et optimisées par des experts et déjà testées par des milliers de programmeurs. Pensez-vous faire mieux ? (peut-être, si besoin très spécifique)
2. Mais pourquoi programmez-vous en Java alors ?

# Les optionnels



**Le problème :** représenter de façon non ambiguë le fait qu'une méthode puisse retourner (ou qu'une variable puisse contenir) aussi bien une valeur qu'une absence de valeur.

**Exemples :**

- résultat du dépilement d'une pile
  (pile peut-être vide)
- recherche d'un élément satisfaisant un certain critère dans une liste
  (liste ne contenant pas forcément un tel élément)
- identité de la personne ayant réservé un certain siège dans un avion
  (siège peut-être pas encore réservé)

**Solutions (pas très bonnes) :**

- retourner une valeur qui peut être `null` (« **nullable** »). **Inconvénients :** [1]
  - si `getVal()` retourne une valeur nullable, l'appel `getVal().doSomething()` peut causer une `NullPointerException`. En toute généralité, cette exception peut se déclencher bien plus loin dans le programme (déboguage difficile).
  - `null` peut aussi représenter une variable pas encore initialisée (ambiguïté)
- lancer une exception pour l'absence de valeur.
  **Inconvénient :** obligation d'utiliser des `try catch` (lourdeur syntaxique, s'intègre mal au flot du programme) ; mécanisme coûteux à l'exécution.
- Utiliser une liste à 0 ou 1 élément.
  **Inconvénient :** le type liste autorise aussi les listes à 2 éléments ou plus.

1. L'invention de `null` a été qualifiée *a posteriori* par son auteur, Tony Hoare, d'« erreur à un milliard de dollars », ce n'est pas peu dire !

# Les optionnels

**Une solution pas trop mauvaise :** [1] la classe `java.util.Optional` [2]

- une instance de `Optional<T>` est une valeur représentant soit une instance présente de `T` soit l'absence d'une instance (par définition, de façon non ambiguë).
- ainsi, instance de `Optional<T>` contient juste un champ de type `T`
- La présence d'un élément se teste en appelant `isPresent`.
- On accède à la valeur de l'élément via la méthode `get` qui ne retourne jamais **`null`** mais lance `NoSuchElementException` si l'élément est absent.

Bien qu'`Optional<T>` n'implémente pas `Collection<T>`, il est pertinent d'imaginer `Optional<T>` comme un type représentant des collections de 0 ou 1 élément.

1. Les valeurs nullables gardent quelques avantages sur `Optional` : pas besoin d'allouer un conteneur supplémentaire, moins de lourdeurs syntaxiques (comme la nécessité d'appeler `isPresent` et `get`). De plus, même une expression de type `Optional` est elle-même nullable...
Des alternatives existent (hors Java : notamment systèmes de types contenant des types non nullables, en Java : annotations `@NotNull` et `@Nullable` + outil d'analyse statique).
2. Inspirée des langages fonctionnels : la classe `Option` en Scala, la monade `Maybe` en Haskell

# Les optionnels

**Pourquoi c'est plus sûr qu'un type nullable :**

- On ne peut pas appeler directement les méthodes de `T` sur une expression de type `Optional<T>` : il faut d'abord extraire son contenu (méthode `get`).
- Ainsi pas de risque de `NullPointerException` (ni sur l'instance d'`Optional<T>` ni sur le résultat de `get()`).
- `get` peut bien lancer `NoSuchElementException`, mais ça se produit là où `get` est appelée. On voit donc tout de suite si et où on a oublié d'appeler `isPresent`.

**Exemple :**

```
Optional<Client> maybeRes = seat.getReservation();
if (maybeRes.isPresent()) {
    Client res = maybeRes.get(); // on est sûr qu'il n'y a pas d'exception
    res.sendReminder(); // aucun risque de NPE car res est résultat de get()
}
```

**Remarque :** cela peut aussi s'écrire

```
seat.getReservation().ifPresent(res -> res.sendReminder());
```

# Les optionnels

## Ah... et comment les instancier au fait ?



La classe `Optional` est munie de 2 fabriques statiques principales :

- `<T> Optional<T> of(T elem)` : si `elem` est non **`null`**, retourne un optionnel contenant `elem` (sinon `NullPointerException`)
- `<T> Optional<T> empty()` : retourne un optionnel vide du type désiré (en fonction du contexte)

**Exemple :**

```
public static Optional<Integer> findIndex(int[] elems, int elem) {
    for (int i = 0; i < elems.length; i++) {
        if (elems[i] == elem) return Optional.of(i);
    }
    return Optional.empty();
}
```

# Les optionnels

## Optionnels spécialisés

Une dernière remarque sur le sujet :

- En plus de la classe générique `Optional<T>`, `java.util` contient aussi les classes non génériques `OptionalInt`, `OptionalLong` et `OptionalDouble` ;
- celles-ci sont sémantiquement équivalentes à, respectivement `Optional<Integer>`, `Optional<Long>` et `Optional<Double>`.
- L'intérêt est d'économiser les indirections (le fait de suivre 2 pointeurs pour obtenir une valeur primitive) et les allocations multiples (celle de l'`Optional` et celle du `Integer` par exemple) pour le cas des types primitifs.

# Fonctions de première classe et fonctions d'ordre supérieur

## Le besoin (1)



Appeler 5 fois une méthode `f` déjà connue :

```
f(); f(); f(); f(); f();
```

Appeler `f` un nombre de fois inconnu à l'avance :

```
// Facile ! On ajoute un paramètre int :
public static void repeatF(int n) { for (int i = 0; i < n; i++) f(); }
```

Appeler 5 fois une méthode inconnue à l'avance ?

```
// Hm... il faudrait passer une méthode en paramètre ? Tentative :
public static void repeat5(??? f) { // quel type pour f ?
    for (int i = 0; i < 5; i++) f(); // si f une variable, "f()" -> erreur de syntaxe
}
```

- `repeat5` = fonction avec paramètre fonction = **fonction d'ordre supérieur** (FOS).
- Pour que cela existe, il faut des fonctions considérées comme des valeurs (passables en paramètre) par le langage : des **fonctions de première classe** (FPC).

# Fonctions de première classe et fonctions d'ordre supérieur

## Le besoin (2)



- Une FPC peut être affectée à une variable, être le paramètre d'une FOS, ou bien sa valeur de retour : c'est une valeur comme une autre.
- Avec des **valeurs fonction**, il devient possible de manipuler des instructions sans les exécuter/évaluer immédiatement (**évaluation paresseuse**). Elles peuvent ainsi :
  - être transformées, composées avant d'être évaluées ;
  - être évaluées plus tard , une, plusieurs fois ou pas du tout, en fonction de critères programmables ; (condition, répétition, déclenchement par évènement ultérieur, ...) ;
  - exécutées dans un autre contexte (p. ex. autre *thread* [1]).

  De telles modalités d'exécution sont programmables en tant que FOS qui se comportent, en gros, comme de nouvelles structures de contrôle [2].

1. Voir chapitre programmation concurrente. La programmation concurrente a probablement été un argument primordial pour l'introduction des lambda-expressions en Java.
2. À comparer avec `while(...)...`, `for(...)...`, `switch(...)...`, `if (...)... else ...`, ...

# Fonctions de première classe et fonctions d'ordre supérieur

## Plus d'exemples (1)



**Bloc `if`/`else` :**

```
// types et syntaxe d'appel des FPC toujours fantaisistes dans cet exemple
public static void ifElse(??? condition, ??? ifBlock, ??? elseBlock) {
    if (condition()) ifBlock();
    else elseBlock();
}
```

Impossible d'écrire la signature d'une méthode mimant le bloc `if`/`else` sans paramètres FPC. Une telle méthode est nécessairement une FOS.

# Fonctions de première classe et fonctions d'ordre supérieur

## Plus d'exemples (2)



Évidemment, plus intéressant d'écrire de nouveaux blocs de contrôle, par exemple :

**Bloc `retry` :**

```
// pareil, ne faites pas ça à  la maison !
public static void retry(??? instructions, int tries) {
    while (tries > 0) {
        try { instructions(); return; }
        catch (Throwable t) { tries--; }
    }
    throw new RuntimeException("Failure persisted after all tries.");
}
```

# Fonctions de première classe et fonctions d'ordre supérieur

## Plus d'exemples (3)



### Encore un exemple : [1]

```
// toujours en syntaxe fantaisiste --- SURTOUT NE PAS RECOPIER OU MÊME RETENIR !
public static <U, V> List<V> map(List<U> l, ??? f) {
    List<V> ret = new ArrayList<>();
    for (U x : l) ret.add(f(x));
    return ret;
}
```

### Ou encore : [2]

```
public static readPacket(Socket s, ??? callback) {
    ...
}
```

`callback` = FPC pour traiter le prochain paquet reçu = **fonction de rappel**/***callback***.

1. L'API `java.util.stream` contient plein de méthodes de traitement par lot dans ce genre.
2. Lecture asynchrone, similaire à ce qu'on trouve dans l'API `java.nio`.

# Programmation fonctionnelle

Concepts de FPC et de FOS essentiels pour la **programmation fonctionnelle** (PF) :

- PF = paradigme de programmation, au même titre que la POO.
- **Idée de base :** on conçoit un programme comme une fonction mathématique, elle-même obtenue par composition d'un certain nombre d'autres fonctions.
- Or pour pouvoir composer les fonctions, il faut supporter les FOS et donc les FPC.
- Langages fonctionnels connus : Lisp (et variantes : Scheme, Emacs Lisp, Clojure... ), ML (et variantes : OCaml, F#... ), Haskell, Erlang...
- Rien n'empêche d'être à la fois objet et fonctionnel (Javascript, Scala, OCaml, Common Lisp ...). Les langages sont souvent multi-paradigme (avec préférence).

Java ($\geq$ 8) possède quelques concepts fonctionnels [1].

1. Mais n'est pas un vrai langage de PF pour autant (on verra plusieurs raisons). Remarque : quasiment tous les langages modernes supportent les FPC, bien qu'ils ne soient pas tous des LPF.

# Que faut-il pour qu'un langage supporte les FPC ?

**Principalement 3 choses :**

- si langage à typage statique, un système de types permettant d'écrire les types des FPC
- une syntaxe adaptée pour les expressions décrivant les FPC.
  Notamment, il faut des littéraux fonctionnels (appelés aussi **lambda-expressions** [1] ou encore fonctions anonymes).
- une représentation en mémoire adaptée pour les FPC
  (en Java, forcément des objets particuliers)

1. En particulier en Java. C'est donc le nom « lambda-expression » que nous allons utiliser.
Pourquoi « lambda » ? Référence au lambda-calcul d'Alonzo Church : la fonction $x \mapsto f(x)$ s'y écrit $\lambda x.f(x)$.

# FPC avant Java 8

## Analyse de l'existant



Dans tout LOO, une FPC est représentable par un objet ayant la fonction comme méthode.

En Java (toute version) on implémente et instancie une interface à méthode unique. Typiquement, pour créer un *thread* à l'aide d'une classe anonyme :

```
new Thread( /* début de l'expression-fonction */ new Runnable {
 @Override public void run() { /* choses à  faire dans l'autre thread */ }
} /* fin de l'expression-fonction */ ).start()
```

Ici, on passe une fonction (décrite par la méthod `run`) au constructeur de `Thread`.

**Inconvénients :**

- syntaxe lourde et peu lisible, même avec classes anonymes,
- obligation de se rappeler et d'écrire des informations sans rapport avec la fonction qu'on décrit (nom de l'interface : `Runnable`; et de la méthode implémentée : `run`).

# FPC avant Java 8

## Bilan



**Bilan pour les FPC avant Java 8 :**

- **typage :** au cas par cas, rien de prévu, pas de standard : chaque méthode peut spécifier une interface différente pour la fonction passée en argument.
- **syntaxe :** lourde et peu pratique.
- **représentation en mémoire :** instance d'une classe contenant juste une méthode.

**Bilan pour les FPC avant Java 8 :**

- **typage :** au cas par cas, rien de prévu, pas de standard : chaque méthode peut spécifier une interface différente pour la fonction passée en argument.
  → à partir de Java 8 : le package `java.util.function` propose une série d'**interfaces fonctionnelles** standard.
  Sinon, rien n'est changé au système de type de Java (même sa syntaxe).
- **syntaxe :** lourde et peu pratique.
- **représentation en mémoire :** instance d'une classe contenant juste une méthode.

**Bilan pour les FPC avant Java 8 :**

- **typage :** au cas par cas, rien de prévu, pas de standard : chaque méthode peut spécifier une interface différente pour la fonction passée en argument.
  → à partir de Java 8 : le package `java.util.function` propose une série d'**interfaces fonctionnelles** standard.
  Sinon, rien n'est changé au système de type de Java (même sa syntaxe).
- **syntaxe :** lourde et peu pratique.
  → à partir de Java 8, on peut écrire des **lambda-expressions**.
- **représentation en mémoire :** instance d'une classe contenant juste une méthode.

**Bilan pour les FPC avant Java 8 :**

- **typage :** au cas par cas, rien de prévu, pas de standard : chaque méthode peut spécifier une interface différente pour la fonction passée en argument.
  → à partir de Java 8 : le package `java.util.function` propose une série d'**interfaces fonctionnelles** standard.
  Sinon, rien n'est changé au système de type de Java (même sa syntaxe).
- **syntaxe :** lourde et peu pratique.
  → à partir de Java 8, on peut écrire des **lambda-expressions**.
- **représentation en mémoire :** instance d'une classe contenant juste une méthode.
  → comme c'est une idée raisonnable, ça ne change pas.

# Interfaces fonctionnelles et types SAM

- **Interface fonctionnelle** = interface avec 1 seule méthode abstraite [1]
- **type SAM** (*single abstract method*) : type défini par une interface fonctionnelle.
- En fait, il en existait déjà plein avant Java 8 (ex. : interfaces `Comparable`, `Comparator`, `Runnable`, `Callable`, `ActionListener`... ).
- L'API `java.util.function` en ajoute quelques dizaines, afin de standardiser les types des FPC. Cf. les 2 pages suivantes.
- Les types SAM sont ainsi les types des FPC de Java (celles dont la signature est la même que celle de la méthode du type SAM).
- Conséquence : une FPC peut être représentée par plusieurs types SAM (interfaces fonctionnelles de noms différents mais méthode de même signature).

  Le système de types de Java reste scrupuleusement nominal.

1. ... mais autant de méthodes **`static`**, **`default`** ou **`private`** que l'on souhaite !

# Catalogue des interfaces fonctionnelles de Java 8

## dans le package `java.util.function` (1)



`java.util.function` contient toute une série d'interfaces fonctionnelles standard.

**Interfaces génériques :**

| Interface | Type représenté | Méthode unique |
|---|---|---|
| `BiConsumer<T,U>` | $T \times U \to \{()\}$ | `void accept(T, U)` |
| `BiFunction<T,U,R>` | $T \times U \to R$ | `R apply(T, U)` |
| `BinaryOperator<T>` | $T \times T \to T$ | `T apply(T, T)` |
| `BiPredicate<T,U>` | $T \times U \to \{\bot, \top\}$ | `boolean test(T, U)` |
| `Consumer<T>` | $T \to \{()\}$ | `void accept(T)` |
| `Function<T,R>` | $T \to R$ | `R apply(T)` |
| `Predicate<T>` | $T \to \{\bot, \top\}$ | `boolean test(T)` |
| `Supplier<T>` | $\{()\} \to T$ | `T get()` |
| `UnaryOperator<T>` | $T \to T$ | `T apply(T)` |

De plus, ce *package* contient aussi des interfaces pour les fonctions prenant ou retournant des types primitifs **`int`**, **`long`**, **`double`** ou **`boolean`** (page suivante).

# Catalogue des interfaces fonctionnelles de Java 8

## dans le package `java.util.function` (2)



**Interfaces spécialisées :**

| Interface | Type représenté | Méthode unique |
|---|---|---|
| `BooleanSupplier` | $\{()\} \to \{\bot, \top\}$ | `boolean getAsBoolean()` |
| `DoubleBinaryOperator` | $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$ | `double applyAsDouble(double, d` |
| `DoubleConsumer` | $\mathbb{R} \to \{()\}$ | `void accept(double)` |
| `DoubleFunction<R>` | $\mathbb{R} \to R$ | `R apply(double)` |
| `DoublePredicate` | $\mathbb{R} \to \{\bot, \top\}$ | `void test(double)` |
| `DoubleSupplier` | $\{()\} \to \mathbb{R}$ | `double getAsDouble()` |
| `DoubleToIntFunction` | $\mathbb{R} \to \mathbb{Z}$ | `int applyAsInt(double)` |
| ... | ... | ... |

Cf. javadoc de `java.util.function` pour liste complète.

Intérêt des interfaces spécialisées : programmes mieux optimisés [1] qu'avec les types « emballés » (`Int`, `Long`, ...).

1. Moins d'allocations et d'indirections.

# Catalogue des interfaces fonctionnelles de Java 8

## dans le package `java.util.function`... mais pas seulement !



Attention, catalogue incomplet :

- L'interface `java.lang.Runnable` reste le standard pour les « fonctions »[1] de $\{()\} \to \{()\}$ (**`void`** vers **`void`**).
- Pas d'interfaces standard pour les fonctions à plus de 2 paramètres $\to$ il faut définir les interfaces soi-même :

```
@FunctionalInterface public interface TriConsumer<T, U, V> {
    void apply(T t, U u, V v);
}
```

L'annotation facultative `@FunctionalInterface` demande au compilateur de signaler une erreur si ce qui suit n'est pas une définition d'interface fonctionnelle.

1. Ces fonctions sont intéressantes pour leurs effets de bord et non pour la transformation qu'elles représentent. En effet, en mathématiques, $card(\{()\} \to \{()\}) = 1$.

# Retour sur les exemples...

## ... avec les bons types et la bonne syntaxe



```java
public static void repeat5(Runnable f) { for (int i = 0; i < 5; i++) f.run(); }
```

```java
public static void ifElse(BooleanSupplier cond, Runnable ifBlock, Runnable elseBlock) {
    if (cond.getAsBoolean()) ifBlock.run();
    else elseBlock.run();
}
```

```java
public static void retry(Runnable instructions, int tries) {
    while (tries > 0) {
        try { instructions.run(); return; }
        catch (Throwable t) { tries--; }
    }
    throw new RuntimeException("Failure persisted after all tries.");
}
```

```java
public static <U, V> List<V> map(List<U> l, Function<U,V> f) {
    List<V> ret = new ArrayList<>();
    for (U x : l) ret.add(f.apply(x));
    return ret;
}
```

# Interfaces fonctionnelles

Écrire les types des FPC, c'est bien. Mais comment exécuter une fonction ?



Comme on a déjà pu voir sur les exemples :

- Il n'y a pas de syntaxe réservée pour exécuter une expression fonctionnelle.
- Il faut donc à chaque fois appeler explicitement la méthode de l'interface fonctionnelle concernée (et donc connaître son nom...).

**Exemple :**

```
Function<Integer, Integer> carre = n -> n * n;
System.out.println(carre.apply(5)); // <--- ici c'est apply
```

mais...

```
Predicate<Integer> estPair = n -> (n % 2) == 0;
System.out.println(estPair.test(5)); // <--- là  c'est test
```

# Syntaxe des lambda-expressions

## lambda-abstraction : par l'exemple



Écrire une fonction anonyme par lambda-abstraction :

```
<paramètres> -> <corps de la fonction>
```

Exemples :

- ```
  x -> x + 2
  ```

  (raccourci pour `(int x)-> { return x + 2; }`)
- ```
  (x, y) -> x + y
  ```

  (raccourci pour `(int x, int y)-> { return x + y; }`)
- ```
  (a, b) -> {
    int q = 0;
    while (a >= b) { q++; a -= b; }
    return q;
  }
  ```
  ```

# Syntaxe des lambda-expressions

## lambda-abstraction : syntaxe formelle



```
<paramètres> -> <corps de la fonction>
```

Syntaxe en détails :

- <paramètres> : liste de paramètres formels (de 0 à plusieurs), de la forme `(int x, int y, String s)`
  Mais juste `(x,y,s)` fonctionne aussi (type des paramètres inféré).
  Et parenthèses facultatives quand il y a un seul paramètre.
  Il est aussi possible (Java $\geq$ 11) de remplacer les noms de types par **`var`**.
- <corps de la fonction>, au choix :
  - une simple expression, p. ex. `(x==y)?s:""`
  - une liste d'instructions entre accolades, contenant une instruction **`return`** si type de retour non **`void`**

# Syntaxe des lambda-expressions

## Référence de méthode



Pour créer une lambda-expression contenant juste l'appel d'une méthode existante :

- on peut utiliser la lambda-abstraction :
  `x -> Math.sqrt(x)`
- mais il existe une notation encore plus compacte :
  `Math::sqrt`
  Ceci s'appelle une **référence de méthode**

Remarque :
`Math::sqrt` est bien équivalent à `x -> Math.sqrt(x)`, et non à `(double x)-> Math.sqrt(x)`. Cela a une incidence pour l'inférence de type (cf. la suite).

# Syntaxe des lambda-expressions

## Référence de méthode : les cas de figure



Supposons la classe suivante définie :

```
class C {
    int val;
    C(int val) { this.val = val; }
    static int f(int n) { return n; }
    int g(int n) { return val + n; }
}
```

La notation « référence de méthode » se décline pour différents cas de figure :

- Méthode statique → `C::f` pour `n -> C.f(n)`
- Méthode d'instance avec récepteur donné → avec `x = new C()`, on écrit `x::g` pour `n -> x.g(n)`
- Méthode d'instance sans récepteur donné → `C::g` pour `(x, n)-> x.g(n)`
- Constructeur → `C::new` pour `n -> new C(n)`

En cas de surcharge, Java déduit la méthode référencée du type attendu.

# Utiliser les méthodes (FOS) de nos exemples...

## ... en leur passant des lambda-expressions



```
// Dire 5 fois "Bonjour !" :
repeat5(() -> { System.out.println("Bonjour !"); });
```

```
// Tirer pile ou face
ifElse( () -> Math.random() > 0.5,
    () -> { System.out.println("pile"); },
    () -> { System.out.println("face"); }
);
```

```
// Essayer d'ouvrir un fichier jusqu'à  3 fois
retry(() -> { ouvre("monFichier.txt"); }, 3);
```

```
// Calculer les racines carrées des nombres d'une liste
List<Double> racines = map(maListe, Math::sqrt);
```

# Ce qui se cache derrière les lambda-expressions



**Nous avons montré :**

- d'une part, les types qui sont utilisés pour les FPC en Java (interfaces fonctionnelles)
- d'autre part, la syntaxe permettant d'écrire des FPC (lambda-expressions)

**Deux questions se posent alors :**

- À la compilation, étant donnée une lambda-expression, quel type le compilateur lui donne-t-il ?
- À l'exécution, comment est évaluée une lambda-expression ?

# Ce qui se cache derrière les lambda-expressions

## Concernant le typage (1)



- Hors contexte, une lambda-expression n'a pas de type (plusieurs types possibles).
- En contexte, sous réserve de compatibilité, son type est le type attendu à son emplacement dans le programme (**inférence de type**).
- Compatibilité si :
  - le type attendu est défini par une interface fonctionnelle (= est un type SAM)
  - et la méthode abstraite de cette interface est redéfinissable par une méthode qui aurait la même signature et le même type de retour que la lambda-expression. [1]

Exemple, on peut écrire `Function<Integer,Double> f = x -> Math.sqrt(x);` car l'interface `Function` est comme suit :

```
public interface Function<T,R> { R apply(T t); }  // apply a une signature compatible
```

1. Ou bien, dans le cas où les types des arguments ne sont pas précisé, s'il existe une façon de les ajouter qui rend la signature compatible.

# Ce qui se cache derrière les lambda-expressions

## Concernant le typage (2) — petits pièges



Le fait de préciser le type des paramètres d'une lambda-expression restreint les possibilités d'utilisation.

Ces exemples compilent :

```
Function<Integer,Double> f = x -> Math.sqrt(x);
Function<Integer,Double> f = (Integer x) -> Math.sqrt(x);
```

Mais ceux-ci ne compilent pas :

```
Function<Integer,Double> f = (Double x) -> Math.sqrt(x);
IntFunction<Double> f = (double x) -> Math.sqrt(x); // pourtant la lambda-expression
    accepte double (plus large que int).
```

Remarquablement, ceci compile (malgré le fait que `sqrt` ait un paramètre **`double`**) :

```
Function<Integer,Double> f = Math::sqrt;
```

# Ce qui se cache derrière les lambda-expressions

## Concernant le typage (3)



**Attention :** n'importe quel type SAM peut être le type d'une lambda-expression. Pas seulement ceux définis dans `java.util.function`.

Partout où une expression de type SAM attendue, on peut utiliser une lambda-expression, même si ce type (ou la méthode qui l'attend) date d'avant Java 8.

Ainsi, en *Swing*, à la place de la classique « invocation magique » :

```
SwingUtilities.invokeLater(
  new Runnable() {
    public void run() { MonIG.build(); }
  }
);
```

on peut écrire : `SwingUtilities.invokeLater(()-> MonIG.build());`

ou encore mieux : `SwingUtilities.invokeLater(MonIG::build);`

# Ce qui se cache derrière les lambda-expressions

## Concernant l'évaluation



À l'évaluation, Java construit un objet singleton (instance d'une classe anonyme) qui implémente l'interface fonctionnelle en utilisant la fonction décrite dans la lambda-expression.

Toujours dans le même exemple, `javac` sait alors qu'il doit compiler l'instruction

```
Function<Integer,Double> f = x -> Math.sqrt(x);
```

de la même façon [1] que :

```
Function<Integer,Double> f = new Function<Integer, Double>() {
    @Override public Double apply(Integer x) {
        return Math.sqrt(x);
    }
};
```

1. En fait, pour les lambdas, la JVM construit la classe anonyme à l'exécution seulement. À cet effet, javac a en fait compilé l'expression en écrivant l'instruction **invokedynamic** (introduite dans Java 8 dans ce but). Autrement, les classes, même anonymes, sont créées à la compilation et existent déjà dans le code octet.

# Avertissement

## Attention aux variables locales !

- Valeur d'une lambda-expression = instance de classe locale (anonyme).
- Ainsi, une lambda-expression de Java ne peut utiliser que les variables locales effectivement **`final`**.[1]
- **Comparaison avec OCaml :** en OCaml, cette « limitation » n'est pas perçue car, en effet, les « variables » ne sont pas réaffectables[2] (tout est « **`final`** »).

  Comme Java, OCaml se contente de recopier les valeurs des variables locales dans la clôture de la fonction ($\simeq$ l'instance de la classe locale).

---

1. **Rappel :** dans une classe locale, on a accès aux variables locales seulement si elles sont **effectivement `final`** (c.-à-d. jamais modifiées après leur initialisation). Cette restriction permet d'éviter les incohérences (modifications locales non partagées).
2. On simule des données locales modifiables en manipulant des « références » mutables :

```
let ref x = 42 in x := !x + 1; x;
```

Dans l'exemple, `x` n'est pas réaffectable, mais la valeur stockée à l'adresse contenue dans `x` l'est. L'équivalent en Java serait un objet-boîte contenant un unique attribut non **`final`**. D'ailleurs, rien n'empêche d'utiliser cette technique en Java ; il faut juste l'écrire « à la main ».

# Avertissement

## Exemple :

### Incorrect :

```
int a = 1;
a++; // a réaffectée
Function<Integer, Integer> f = x -> { return x + a; };
```

### Correct :

```
final int a = 1; // a final (non réaffectable)
Function<Integer, Integer> f = x -> { return x + a; };
```

### Aussi correct :

```
int a = 1; // a effectivement final (non réaffectée)
Function<Integer, Integer> f = x -> { return x + a; };
```

### Et correct aussi :

```
class IntRef { int val; IntRef(int val) { this.val = val; } }
final IntRef a = new IntRef(12);
Function<Integer, Integer> f = x -> { return a.val += x; /* modification de a */ };
```

# Fonctions récursives

- Supposons qu'on veuille définir une fonction récursive, comme en OCaml :

```
let rec fact = function
    | 0 -> 1
    | n -> n * fact (n - 1);;
```

- Sachant qu'il n'existe pas l'équivalent de **let rec** en Java, peut-on définir ?

```
IntUnaryOperator fact = n -> (n==0)?1:(n*fact.applyAsInt(n-1));
```

- Problème : la variable `fact` n'est pas encore déclarée quand on compile la lambda-expression $\Longrightarrow$ la compilation échoue.

Il existe des dizaines de façons de contourner cette limite, mais la seule qui soit élégante consiste à définir d'abord une méthode récursive.

# Fonctions récursives

Pour définir cette FPC, il faudrait donc faire en 2 temps :

1. initialiser `fact` (à **`null`** par exemple)
2. écrire la lambda-expression (utilisant `fact`) et l'affecter à `fact`.

Il faudrait donc que la variable `fact` soit réaffectable... donc `fact` ne pourrait pas être locale.

→ Il faudrait que `fact` soit un attribut, mais alors attention à l'encapsulation.

Une possibilité, respectant l'encapsulation, à l'aide d'une classe locale auxiliaire :

```
class FunRef { IntUnaryOperator val; }
final FunRef factAux = new FunRef();
factAux.val = n -> (n==0)?1:(n*factAux.val.applyAsInt(n-1));
IntUnaryOperator fact = factAux.val;
```

**Inconvénient :** niveau d'indirection supplémentaire.

# Fonctions récursives

**Alternative :** déclarer la factorielle comme méthode privée récursive, puis manipuler une référence vers celle-ci.

```
class Autre2 {
    ...
    private int fact(int n) { return (n==0)?1:(n*fact(n-1)); }
    ...

    IntUnaryOperator fact = Autre::fact;
    ...
}
```

Et si on veut tout encapsuler correctement, on revient à une classe anonyme classique :

```
IntUnaryOperator fact = new IntUnaryOperator() {
   @Override public int applyAsInt(int n) { return (n==0)?1:(n*applyAsInt(n-1)); }
}
```

Cette dernière technique n'a pas d'inconvénient [1]. C'est donc celle qu'il faut privilégier.

1. si, un : on troque la syntaxe des lambda-expressions contre celle, plus verbeuse, des classes anonymes

# Bilan des FPC dans Java 8

## Le mauvais



- Pas de notation (flèche) dédiée aux types fonctionnels, juste interfaces classiques.
- Plusieurs interfaces possibles pour une même fonction.
- Pas de syntaxe réservée, unique, pour exécuter une FPC.
  À la place : appel de la méthode de l'interface, dont le nom peut varier.
- Clôture contenant variables effectivement finales seulement.
  Implication : nécessité de « contourner » pour capturer un état mutable. [1]
  ($\Rightarrow$ Impossible de définir simplement une lambda-expression récursive [2].)
- Malgré les apports de Java 8 à 15, le JDK contient peu d'APIs dans le style fonctionnel (On peut néanmoins citer `Stream` et `CompletableFuture`).

Java n'est toujours pas un langage de PF, mais juste un LOO avec support limité des FPC.

1. Cela dit, on évite d'utiliser un état mutable en PF.
2. De plus Java n'optimise pas la récursivité terminale. Ainsi, l'appel d'une méthode récursive sur des données de taille modérément grande risque facilement de provoquer un `StackOverflowError`.
Ainsi rien n'est fait pour encourager la programmation récursive.

# Bilan des FPC dans Java 8

## Le bon



- Java supporte les FPC et les FOS.
  Or les FPC sont amenées à jouer un rôle de plus en plus important, notamment pour la programmation concurrente [1], qui devient de plus en plus incontournable [2].
- Les API `Stream` et `CompletableFuture` sont d'excellents exemples d'API concurrentes, introduites dans Java 8 et utilisant les FOS.
- D'anciennes API se retrouvent immédiatement utilisables avec les lambda-expressions car utilisant déjà des interfaces fonctionnelles (e.g. JavaFX).
  $\rightarrow$ nouvelle concision, « gratuite ».

Malgré ses défauts, le support des FPC et FOS dans Java est un apport indéniable.

1. Quel est le rapport entre FPC/FOS et programmation concurrente ? Plusieurs réponses :
   - la programmation concurrente incite à utiliser des structures immuables pour garantir la correction du programme. Or le style fonctionnel est naturel pour travailler avec les structures immuables.
   - en programmation concurrente, on demande souvent l'exécution asynchrone d'un morceau de code. Pour ce faire, ce dernier doit être passé en argument d'une fonction (FOS) sous la forme d'une FPC.
2. En particulier à cause de la multiplication du nombre de cœurs dans les microprocesseurs.

# Opérations d'agrégation

## Définition et quelques exemples



**Opération d'agrégation :** traitement d'une séquence de données de même type qui produit un résultat synthétique [1] dépendant de toutes ces données.

**Exemples :**

- calcul de la taille d'une collection
- concaténation des chaînes d'une liste de chaînes
- transformation d'une liste de chaînes en la liste de ses longueurs (ex : "bonjour", "le", monde $\rightarrow$ 7, 2, 5)
- recherche d'un élément satisfaisant un certain critère

Tous ces calculs pourraient s'écrire à l'aide de boucles `for` très similaires...

1. synthèse = résumé

# Opérations d'agrégation

## Quelques exemples (1)



Calcul de la taille d'une collection :

```
public static int size(Collection<?> dataSource) {
    int acc = 0;
    for (Object e: dataSource) acc++;
    return acc;
}
```

Concaténation des chaînes d'une liste de chaînes :

```
public static String concat(List<String> dataSource) {
    String acc = "";
    for (String e: dataSource) acc += e.toString();
    return acc;
}
```

# Opérations d'agrégation

## Quelques exemples (2)



Transformation d'une liste de chaînes en la liste de ses longueurs :

```
public static List<Integer> lengths(List<String> dataSource) {
    List<Integer> acc = new LinkedList<>();
    for (String e: dataSource) acc.add(e.length());
    return acc;
}
```

Recherche d'un élément satisfaisant un certain critère [1] :

```
public static <E> E find(List<E> dataSource, Predicate<E> criterion) {
    E acc = null;
    for (E e: dataSource) acc = (criterion.test(e)?e:null);
    return acc;
}
```

Voyez-vous le motif commun ?

1. Remarque : on peut optimiser cette boucle, mais cette présentation illustre mieux le propos.

# Opérations d'agrégation

## Généralisation



On garde ce qui est commun dans une méthode prenant en argument ce qui est différent :

```
public static <E, R> R fold(Iterable<E> dataSource, R zero, ??? op) {
    R acc = zero;
    for (E e : dataSource) acc = op(acc, e); // comment on écrit ça déjà  ?
    return acc;
}
```

# Opérations d'agrégation

## Généralisation



On garde ce qui est commun dans une méthode prenant en argument ce qui est différent :

```
public static <E, R> R fold(Iterable<E> dataSource, R zero, ??? op) {
    R acc = zero;
    for (E e : dataSource) acc = op(acc, e); // comment on écrit ça déjà  ?
    return acc;
}
```

... et on se rappelle le cours sur les fonctions de première classe (FPC) et les fonctions d'ordre supérieur (FOS) :

```
public static <E, R> R fold(Iterable<E> dataSource, R zero, BiFunction<R, E, R> op) {
    R acc = zero;
    for (E e : dataSource) acc = op.apply(acc, e);
    return acc;
}
```

# Opérations d'agrégation

## Généralisation



On peut alors écrire :

```
public static int size(Collection<?> dataSource) {
    return fold(dataSource, 0, (acc, e) -> acc + 1);
}

public static String concat(List<String> dataSource) {
    return fold(dataSource, "", (acc, e) -> acc + e.toString());
}

public static List<Integer> lengths(List<String> dataSource) {
    return fold(dataSource, new LinkedList<>(),
        (acc, e) -> { acc.add(e.length()); return acc; });
}
// "Bof" : on modifie l'argument de op dans op (incorrect si fold est concurrent).
// On doit pouvoir faire mieux (à  méditer en TP... ) !

public static <E> E find(List<E> dataSource, Predicate<E> criterion) {
    return fold(dataSource, null, (acc, e) -> (criterion.test(e) ? e : null));
}
```

# Opérations d'agrégation

## Généralisation



Pour écrire des traitements similaires à ces exemples, on aimerait une API fournissant des FOS analogues à `fold` pour les principaux schémas d'itération [1].

C'est justement ce que fait l'API *stream*. [2]

1. Similaires aux fonctions de manipulation de liste en OCaml.
2. Mais pas seulement...

# L'API *Streams*

## Ce dont il s'agit.



***Streams* :** API introduite dans Java 8 pour effectuer des opérations d'agrégation.

- API dans le style fonctionnel, avec fonctions d'ordre supérieur ;
- distincte de l'API des collections (nouvelle interface `Stream`, au lieu de méthodes ajoutées à `Collection`[1]) ;
- optimisée pour les grands volumes de données : **évaluation paresseuse** (calculs effectués seulement au dernier moment, seulement lorsqu'ils sont nécessaires) ;
- qui sait utiliser les CPUs multi-cœur pour accélérer ses calculs (implémentation parallèle *multi-thread*ée).

**Avertissement :** ce chapitre traite du *package* `java.util.stream` introduit dans Java 8. Ces *streams* n'ont **aucun rapport** avec les classes `InputStream` et `OutputStream` de `java.io`.

1. Heureusement on obtient facilement une instance de de `Stream` depuis une instance de `Collection`, grâce à la méthode `stream` de `Collection`.

# Opérations d'agrégation

## Avec les *streams*

Avec l'API *stream*, une telle opération se décompose sous forme d'un ***pipeline*** d'étapes successives, selon le schéma suivant :

1. sélection d'une **source** d'éléments [1]. Depuis cette source on obtient un ***stream***.
2. un certain nombre (0 ou plus) d'**opérations intermédiaires**. Ces opérations transforment un *stream* en un autre *stream*.
3. une **opération terminale** qui transforme le *stream* en un résultat final (qui n'est plus un *stream*).

Les calculs sont effectués à l'appel de l'opération terminale seulement. Et seuls les calculs nécessaires le sont.

1. Souvent une collection, mais peut aussi être un tableau, une fonction productrice d'éléments, un canal d'entrées/sorties

**Quelques *streams* :**

- Pour toute collection `coll`, le *stream* associé est `coll.stream()`.
- `Stream.of(4,39,2,12,32)` représente la séquence 4, 39, 2, 12, 32.
- `Stream.of(4,39,2,12,32).map(x -> 2 * x)` représente la séquence 8, 78, 4, 24, 64.

Inversement, on peut ensuite obtenir une collection depuis un *stream* :

```
Stream.of(4,39,2,12,32).map(x -> 2 * x).collect(Collectors.toList)
```

$\rightarrow$ on obtient un `List<Integer>` (`collect`, opération terminale, force le calcul de la séquence).

# Les objets *stream*

## Caractéristiques principales



Qu'est-ce qu'un *stream* ? → **2 points de vue :**

1. la représentation implicite d'une séquence d'éléments finie ou infinie
2. la description d'une suite d'opérations permettant d'obtenir cette séquence.

**Remarques importantes :**

- Un objet *stream* n'est pas une collection : il ne contient qu'une référence vers une source d'éléments (parfois une collection, souvent un autre *stream*) et la description d'une opération à effectuer.
- Un objet *stream* n'est pas le résultat d'un calcul, mais la description d'**un calcul à effectuer** [1].

1. Pour les fans de programmation fonctionnelle : le type `Stream<T>` muni des opérations `of` et `flatMap` est une monade.

# Les objets *stream*

## `Stream` versus `Iterator`



Les *streams* et les *iterators* ont beaucoup en commun :

- intermédiaires techniques pour parcourir les collections
- contiennent juste l'information pour faire cela ; pas les éléments eux-mêmes ;
- usage unique (après le premier parcours de la source, l'objet ne peut plus servir)

Et une grosse différence :

- *streams* : opérations agissant sur l'ensemble des éléments (itération implicite). Ce sont les méthodes fournies dans le JDK qui gèrent l'itération (et proposent notamment une implémentation en parallèle sur plusieurs *threads* [1] ).
- *iterators* : 1 opération (`next`) = lire l'élément suivant ($\rightarrow$ itération explicite avec **`for`** ou **`while`**)

1. Le *stream* construit sur une collection utilise en fait le *spliterator* de celle-ci : sorte d'itérateur évolué capable, en plus d'itérer séquentiellement, de couper une collection en morceaux (*"split"*) pour partager le travail entre plusieurs *threads*.

# Quelques exemples de traitements réalisables en utilisant les *streams*

15 nombres entiers aléatoires positifs inférieurs à 100 en ordre croissant :

```
Stream.generate(Math::random) //     on obtient un Stream<Double>
   .limit(15) //                                  Stream<Double>
   .map(x -> (int)(100 * x)) //                   Stream<Integer>
   .sorted() //                                   Stream<Integer>
   .collect(Collectors.toList()); //              List<Integer>
```

Nombre d'usagers d'une bibliothèque ayant emprunté un livre d'Alexandre Dumas :

```
bibli.getLivres() //                                                        List<Livre>
    .stream() //                                                            Stream<Livre>
    .filter(livre -> livre.getAuteur().equals("Dumas, Alexandre")) //       Stream<Livre>
    .flatMap(livre -> livre.getEmprunteurs().stream()) //                   Stream<Usager>
    .distinct() //                                                          Stream<Usager>
    .count(); //                                                            long
```

# *Streams* et parallélisme

## Problème à résoudre

- la plupart des collections ne sont pas *thread safe* (comportement incorrect quand utilisées dans plusieurs *threads* en même temps, notamment à cause des accès en compétition)
- on peut y ajouter de la synchronisation (voir collections synchronisées), mais toujours risque de *dead-lock*.

Pourtant, accélérer le traitement les grandes collections, il est utile de profiter du parallélisme.

# *Streams* et parallélisme

## Problème à résoudre... une solution ?

**Les *streams*, semblent une réponse naturelle à ce problème.** En effet :

- leurs opérations ne modifient pas le contenu de leur source ;
- l'objet de type `Stream` est lui-même à usage unique.

→ protection naturelle maximale contre les accès en compétition.

→ **Ce serait bien que les opérations d'aggrégation d'un *stream* puissent être réparties sur plusieurs *threads*** (lors de l'appel à l'opération terminale)...

**... et bien justement :**

Java permet de lancer les opérations d'agrégation en parallèle [1], sans presque rien changer à l'invocation du même traitement en séquentiel :

- Il suffit de créer le *stream* avec `maCollection.parallelStream()` à la place de `maCollection.stream()`.
- Alternative : à partir d'un *stream* séquentiel, on peut obtenir un *stream* parallèle avec la méthode `parallel`, et vice-versa avec la méthode `sequential`

Un *stream* est soit (entièrement) parallèle, soit (entièrement) séquentiel. L'opération terminale prend seulement en compte le dernier appel à `sequential` ou `parallel` [2].

1. En utilisant (de façon cachée) `ForkJoinPool`/`ForkJoinTask`. Sauf mention contraire, le *thread pool* par défaut `ForkJoinPool.commonPool()` est utilisé.
2. Rappel : l'effectuation des calculs étant seulement déclenchée par l'opération terminale, il est logique que ses modalités concrètes d'exécution ne soient prises en compte qu'à ce moment.

# *Streams* et parallélisme

## Quelques explications

- Les calculs d'un *pipeline* parallèle sont répartis sur plusieurs *threads*.
- Leur ordre d'exécution peut être sans rapport avec celui des éléments de la source.
- L'opération terminale retourne après que tous les calculs parallèles sont terminés (synchronisation !).
- Certaines opérations garantissent que les éléments sont traités dans l'ordre, s'ils en avaient un (p. ex. : `forEachOrdered`), d'autres non (`forEach`).
- Imposer l'ordre demande plus de synchronisation, impliquant moins de parallélisme.
- Certaines opérations sont optimisées pour le traitement parallèle (p. ex. `.collect(Collectors.toConcurrentMap(...)` plus efficace que `.collect(Collectors.toMap(...))`.

# *Streams* et parallélisme

## Piège à éviter : les effets de bord

En général, évitez les **effets de bord** [1] dans le *pipeline*, préférez les **fonctions pures** [2].

- Pour les entrées/sorties, au mieux, pas de contrôle sur leur ordre.
- Pour les modifications d'objets partagés :
  - sans synchronisation, risque d'accès en compétition. Or, dans ce cas, le modèle de concurrence de Java ne garantit rien ($\rightarrow$ résultats incorrects).
  - avec synchronisation : comme on ne contrôle pas l'ordre d'exécution des tâches du *pipeline*, risque de *dead lock*.
    Sinon, de toute façon, la synchronisation ralentit l'exécution.

Heureusement, habituellement [3], il est inutile de modifier des objets extérieurs dans les opérations d'un *stream*.

1. Effet de bord : tout effet externe d'une fonction, c'est à dire toute sortie physique ou modification de mémoire en dehors de son espace propre (= variables locales + champs des objets non partagés).
2. Fonction pure : fonction (méthode ou FPC) sans effet de bord.
3. À part à des fins de déboguage ou de *monitoring*.

# Appendice : les méthodes de l'API *stream*



Les transparents qui suivent :

- sont un résumé des méthodes proposées dans l'API *stream*.
- ne sont pas détaillés en cours magistral
- doivent servir de référence pour les TPs et pour la relecture approfondie du cours.

# L'interface Stream

## L'interface principale du package

```
public interface Stream<T> { // pour des éléments de type T
  ...
}
```

(Il existe aussi `DoubleStream`, `IntStream` et `LongStream`.)

Cette interface contient un grand nombre de méthodes. 3 catégories :

- des méthodes statiques [1] servant à créer des *streams* depuis des sources diverses.
- des méthodes d'instance transformant des *streams* en *streams* (pour les opérations intermédiaires)
- des méthodes d'instance transformant des *streams* en autre chose (pour les opérations terminales).

1. Rappel : oui, c'est possible depuis Java 8.

# Fabriquer un *stream* depuis une source

- Depuis une collection : méthode `Stream<T> stream()` de `Collection<T>`.
- À l'aide d'une des méthodes statiques de `Stream` :
  - `<T> Stream<T> empty()` : retourne un *stream* vide
  - `<T> Stream<T> generate(Supplier<T> s)` : retourne la séquence des éléments générés par `s.get()` (Rappel : `Supplier<T>` = fonction de $\{()\} \to T$.).
  - `<T> Stream<T> iterate(T seed, UnaryOperator<T> f)` : retourne la séquence des éléments `seed`, `f.apply(seed)`, `f.apply(f.apply(seed))` ...
  - `<T> Stream<T> of(T... values)` : retourne le *stream* constitué de la liste des éléments passés en argument (méthode d'arité variable).
- En utilisant un *builder* [1] (`Stream.Builder`) :
  - Un `Stream.Builder` est un objet mutable servant à construire un *stream*.
  - On instancie un *builder* vide avec l'appel statique `b = Stream.builder()`
  - On ajoute des éléments avec les appels `b.add(T e)` ou `b.accept(T e)`.
  - On finalise en créant le *stream* contenant ces éléments : appel `s = b.build()`

1. On parle du patron de conception *builder* (ou "monteur"), ici appliqué aux *streams*. Ainsi, par exemple, il existe une classe `StringBuilder` jouant le même rôle pour les `String`. Voir le TP sur le patron *builder*.

# Transformer un *stream* : les opérations intermédiaires (1)

Pour **`this`** instance de `Stream<T>` :

- ```
  Stream<T> distinct()
  ```
  retourne un *stream* qui parcourt les éléments de **`this`** sans les doublons.
- ```
  Stream<T> filter(Predicate<? super T> p)
  ```
  retourne le *stream* parcourant les éléments `x` de **`this`** qui satisfont `p.test(x)`
- ```
  <R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)
  ```
  retourne la concaténation des *streams* `mapper.apply(x)` pour tout `x` dans **`this`**.
- ```
  Stream<T> limit(long n)
  ```
  tronque le *stream* après `n` éléments.
- ```
  <U> Stream<U> map(Function<T, U> f)
  ```
  retourne le *stream* des éléments `f.apply(x)` pour tout `x` élément de **`this`**.
  ```

# Transformer un *stream* : les opérations intermédiaires (2)

- ```
  Stream<T> peek(Consumer<? super T> c)
  ```

  retourne un *stream* avec les mêmes éléments que **`this`**. À l'étape terminale, pour chaque élément `x` parcouru, `c.consume(x)` sera exécuté [1].

- ```
  Stream<T> skip(long n)
  ```

  retourne le suffixe de la séquence en sautant les `n` premiers éléments.

- ```
  Stream<T> sorted()
  ```

  retourne la séquence, triée dans l'ordre naturel.

- ```
  Stream<T> sorted(Comparator<? super T> comparator)
  ```

  idem mais en suivant l'ordre fourni.

1. Remarque : `c` ne sert que pour ses effets de bord. `peek` peut notamment être utile pour le déboguage.

# Calculer et extraire un résultat : les opérations terminales

## Opérations spécialisées (1)



- `boolean allMatch(Predicate<? super T> p)` retourne vrai si et seulement si `p` est vrai pour tous les éléments du *stream*.
- `boolean anyMatch(Predicate<? super T> p)` retourne vrai si et seulement si `p` est vrai pour au moins un élément du *stream*.
- `long count()` retourne le nombre d'éléments dans le *stream*.
- `Optional<T> findAny()` : retourne un élément (quelconque) du *stream* ou rien si *stream* vide (voir interface `Optional<T>`).
- `Optional<T> findFirst()` : pareil, mais premier élément.

# Calculer et extraire un résultat : les opérations terminales

## Opérations spécialisées (2)



- `void forEach(Consumer<? super T> action)` : applique `action` à chaque élément.
- `void forEachOrdered(Consumer<? super T> action)` : pareil en garantissant de traiter les éléments dans l'ordre de la source si elle en avait un.
- `Optional<T> max(Comparator<? super T> comp)` : retourne le maximum.
- `Optional<T> min(Comparator<? super T> comp)` : retourne le minimum.
- `Object[] toArray()` : retourne un tableau contenant les éléments du *stream*.
- `<A> A[] toArray(IntFunction<A[]> generator)` : retourne un tableau contenant les éléments du *stream*. La fonction generator sert à instancier un tableau de la taille donnée par son paramètre.

# Calculer et extraire un résultat : les opérations terminales

## Opérations “couteau suisse” :



- `Optional<T> reduce(BinaryOperator<T> op)` : effectue la réduction du *stream* par l’opération d’accumulation associative `op`.
- `T reduce(T zero, BinaryOperator<T> op)` : idem, avec le zéro fourni.
- `<U> U reduce(U z, BiFunction<U, ? super T, U> acc, BinaryOperator<U> comb)` : idem avec accumulation vers autre type. [1]
- `<R> R collect(Supplier<R> z, BiConsumer<R, ? super T> acc, BiConsumer<R,R> comb)` : comme reduce avec accumulation dans objet mutable.
- `<R,A> R collect(Collector<? super T,A,R> collector)` : on parle juste après.

1. Opération appelée `fold` dans d’autres langages. Les définitions varient...

# L'interface `Collector`

**interface** `Collector<T,A,R>`



Un `Collector` est un objet servant à "réduire" un *stream* en un résultat concret (en effectuant le calcul). Pour ce faire,

- il initialise un accumulateur du type désiré (p. ex : liste vide),
- puis transforme et aggrège les éléments issus du calcul du *stream* dans l'accumulateur (ex : ajout à la liste)
- enfin il "finalise" l'accumulateur avant retour (p. ex : suppression des doublons).

Trois techniques pour fabriquer un tel objet :

- **(cas courants) utiliser une des fabriques statiques de la classe `Collectors`**
- utiliser la fabrique statique `Collector.of()` ( "constructeur" généraliste)
- programmer à la main une classe qui implémente l'interface `Collector`

# La classe `Collectors`

Cette classe, non instanciable, est une bibliothèque de fabriques statiques pour obtenir simplement les *collectors* les plus courants. Quelques exemples : `Collectors.toList()`, `Collectors.toSet()`, `Collectors.counting()`, `Collectors.groupingBy(...)`, `Collectors.reducing(...)`, `Collectors.toConcurrentMap(...)`...

$\rightarrow$ on retrouve des opérations équivalentes [1] à la plupart des réductions de l'interface `Stream`.

Ainsi, autre façon d'avoir la taille d'un *stream* : `monStream.collect(Collectors.counting())` [2].

1. mais ici : implémentation "mutable", utilisant un attribut accumulateur, alors que dans `Stream`, les réductions utilisent de fonctions "pures"
2. ... mais le plus simple reste `monStream.count()` !

# Construire un collector via l'interface `Collector`

Au cas où la bibliothèque `Collectors` ne contient pas ce qu'on cherche, on peut créer un `Collector` autrement :

- créer et instancier une classe implémentant `Collector`.
  Méthodes à implémenter : `accumulator()`, `characteristics()`, `combiner()`, `finished()` et `supplier()`.
- sinon, créer directement l'objet grâce à la méthode statique `Collector.of()` :

```
c2 = Collector<Integer, List<Integer>, Integer> c2 = Collector.of(
    ArrayList<Integer>::new, List::add,
    (l1, l2) -> {l1.addAll(l2); return l1;}, List::size
    );
```

(façon... un peu alambiquée de calculer la taille d'un *stream*... )

Utiliser la méthode `of()` est plus "légèr" syntaxiquement, mais ne permet pas d'ajouter des champs ou des méthodes à l'objet fabriqué.

# Types génériques et sous-typage

## Invariance des génériques



**Le problème :** examinons l'exemple suivant (qui ne compile pas).

```
public static void main(String[] args) {
  List<VoitureSansPermis> listVoitureSP = new ArrayList<>();

  // l1: ceci est en fait interdit... mais supposons que ça passe...
  List<Voiture> listVoiture = listVoitureSP;

  // l2: instruction bien typée (pour le compilateur), mais....
  listVoiture.add(new Voiture());

  // l3: ... logiquement ça afficherait "Voiture" (contradictoire)
  System.out.println(listVoitureSP.get(0).getClass());
}
```

S'il compilait, en l'exécutant, à la fin, `listVoitureSP = listVoiture` contiendrait des `Voiture` → **contredit la déclaration de `listVoitureSP`!**

Ainsi, Java interdit `l1` : deux spécialisations différentes du même type générique sont incompatibles. Ont dit que les génériques de Java sont **invariants**.

# Types génériques et sous-typage

## Invariance des génériques



*type erasure* → vérification **à la compilation seulement**. Court-circuitons-la, pour voir :

```
// l1': version avec "triche" (pas d'exception car type erasure)
List<Voiture> listVoiture = (List<Voiture>)(Object) listVoitureSP;

/* l2: */ listVoiture.add(new Voiture());

// l3: ça affiche effectivement "Voiture" (oooh !)
System.out.println(listVoitureSP.get(0).getClass());

// l4: et pour la forme, une petite ClassCastException :
VoitureSansPermis vsp = listVoitureSP.get(0);
```

**Note :** cependant le compilateur détecte la conversion « louche » et signale un avertissement (*warning*) « `unchecked conversion` » pour la ligne `l1'`.

**Moralité :** en programmation générique, le paramètre de type fournit une garantie stricte et le compilateur refuse de compiler au moindre doute. Si on passe outre (*cast*), il nous avertit (à raison car `ClassCastException` peut se produire à l'exécution).

# Tableaux – génériques ou pas ? (1)

**Remarque :** l'analogue à l'exemple précédent utilisant `Voiture[]` au lieu de `List<Voiture>` compile sans avertissement :

```
public static void main(String[] args) {
  VoitureSansPermis[] listVoitureSP = new VoitureSansPermis[100];

  // l1: ceci est autorisé !
  Voiture[] listVoiture = listVoitureSP;

  // l2: instruction bien typée (pour le compilateur), mais.... ArrayStoreException !
  listVoiture[0] =  new Voiture();

  // l3: on ne va pas jusque là
  System.out.println(listVoitureSP[0].getClass());
}
```

→ Les tableaux sont **covariants** (l1 autorisé) : $[A <: B] \implies [A[] <: B[]]$.

Mais on crashe plus loin, lors de l'exécution de l2 (bien qu'il n'y ait pas eu de *warning* !).

# Tableaux – génériques ou pas ? (2)



- covariance à la place d'invariance : ⇒ vérifications moins strictes à la compilation, rendant possibles des problèmes à l'exécution [1]
- pour détecter les problèmes au plus tôt : à l'instanciation, un tableau enregistre le nom du type déclaré pour ses éléments (pas d'effacement de type)
- cela permet à la JVM de déclencher `ArrayStoreException` lors de toute tentative d'y stocker un élément du mauvais type, au lieu de `ClassCastException` lors de son utilisation (donc bien plus tard).

→ « Genre de » généricité, mais conception obsolète : avec la généricité moderne, la compilation garantit une exécution sans erreur.

1. Raison : un tableau est à la fois producteur et consommateur. D'un point de vue théorique, une telle structure de données ne peut être qu'invariante, si on veut des garanties dès la compilation.

# Génériques et tableaux

## Différence de philosophie (1)



- Tableaux : (vérification à l'exécution, mais le + tôt possible)
  - **usage normal :** conversion sans warning de `SousType[]` à `SuperType[]` par upcasting (implicite).
    Possibilité d'`ArrayStoreException` à l'exécution.

```
Object[] tab = new String[10];
tab[0] = Integer.valueOf(3); // BOOM ! (ArrayStoreException)
```

    Pas idéal, mais aurait pu être pire : le crash évite que le programme continue avec une mémoire incohérente.
  - **usage anormal,** avec *cast* explicite vers type incompatible : `(String[])(Object)(new Integer[10])` compile mais avec warning et fait `ClassCastException` quand on exécute (tout va bien : on avait été prévenu).

# Génériques et tableaux

## Différence de philosophie (2)



- Génériques : (vérification à la compilation... puis plus rien)
  - **usage normal,** le compilateur rejette toute tentative de conversion implicite de `Gen<A>` à `Gen<B>`, garantissant qu'à l'exécution toute instance de `Gen<T>` sera bien utilisée avec le type $T \rightarrow$ exécution cohérente et sans exception garantie.
  - **usage anormal,** conversion forcée : `(Gen<B>)(Object)(new Gen<A>())` compile avec un warning et... provoque des erreurs à retardement à l'exécution (très mal, mais on a été prévenu) ! Exemple :

```
List<String> ls = new ArrayList<String>();
List<Integer> li = (List<Integer>)(Object) ls; //exécution ok ! (oh !)
li.add(5); // toujours pas de crash... (double oh !)
ls.get(0)); // BOOM à  retardement ! (ClassCastException)
```

# Génériques et tableaux

## Et si on mélangeait ?

**Tableaux et génériques ne font pas bon ménage** : les uns ont besoin de tout savoir à l'exécution, alors que les autres veulent tout oublier !

- Avec `T`, paramètre de type, **`new T[taille]` est impossible**.
  Raison : pour instancier un tableau, Java doit connaître dès la compilation le type concret des éléments du tableau.
  Or à la compilation, `T` n'est pas associé à un type concret.
- Les types tableau de types paramétrés, comme ~~`List<Integer>[]`~~, **sont illégaux**.
  Raison : à l'exécution, Java ne sait pas distinguer `List<Integer>` et `List<String>` et donc ne peut pas accepter de mettre des `List<Integer>` dans un tableau sans aussi accepter `List<String>`.
  $\Longrightarrow$ Tout ce qui est dans le tableau pouvant ensuite être affecté à une variable de type `List<Integer>`, la garantie promise par la généricité serait cassée.

# Génériques et tableaux

Premier interdit : ~~`new T[10]`~~



Supposons `T extends Up` paramètre de type.

~~`new T[10]`~~ est aussi interdit.

**Raison :** après compilation, `T` est oublié et remplacé par `Up`. Au mieux `new T[10]` pourrait être compilé comme `new Up[10]`. Mais si c'était ce qui se passait, on pourrait trop facilement « polluer » la mémoire sans s'en rendre compte :

```
static <T> T[] makeArray(int size) { return new T[10]; /* interdit ! */ }
static {
    String[] tString = makeArray(10); // à  l'exécution on affecterait un Object[]
    Object[] tObject = tString; // toujours autorisé (covariance)
    tObject[0] = 42; // et BOOM ! Maintenant tString contient un Integer !
}
```

En pratique, pour faire compiler cela, il faut « tricher » avec *cast* explicite :
`T[] tab = (T[])new Up[10];`.
Ça n'empêche pas le problème ci-dessus, mais au moins le compilateur affiche un *warning* (« `unchecked conversion` »).

# Génériques et tableaux

## Deuxième interdit : ~~Gen<T>[]~~ (1)



Mauvais scenario, pouvant se produire si on autorisait les tableaux de génériques :

```
class Box<T> {
    final T x;
    Box(T x) { this.x = x; }
}

class Loophole {
    public static void main(String[] args) {
        Box<String>[] bsa = new Box<String>[3];     // supposons que cette ligne compile
        Object[] oa = bsa;                          // autorisé car tableaux covariants

        /* autorisé à  la compilation (Box < Object) le test à  l'exécution est aussi ok
        (parce que le tableau référencé par oa est celui instancié à  la première ligne
        et que le type enregistré dans la JVM est juste Box) */
        oa[0] = new Box<Integer>(3);

        /* ... et là , c'est le drame !
        (ClassCastException, alors que l'instruction est bien typée !) */
        String s = bsa[0].x;
    }
}
```

# Génériques et tableaux

## Deuxième interdit : ~~Gen<T>[]~~ (2)



~~`new Box<Integer>[10]`~~ est interdit.

En effet, le tableau instancié serait de type `Box[]` [1], ce qui rendrait possible le scénario du transparent précédent.

En revanche, Java autorise `new Box[10]`.

**Remarque,** du coup, là aussi, il existe une « triche » pour faire compiler l'exemple du transparent précédent : on remplace `new Box<String>[3]` par `new Box[3]`.
Le compilateur émet heureusement un warning (« unchecked conversion »)... et à l'exécution, on a effectivement `ClassCastException`

Encore une fois, la « triche » permet de compiler, n'empêche pas l'exception à l'exécution, mais seulement on a été prévenu par le warning du compilateur !

1. À cause de l'effacement de tye, `Box<Integer>` n'existe pas à l'exécution. Toutes les spécialisations ont le même type : `Box` !

# Les génériques invariants c'est bien mais...

Invariance des génériques → garanties fortes : très bien, mais... très rigide à l'usage !

**Le besoin :** quand $B <: A$ [1], on aimerait pouvoir écrire `Gen<A> g = new Gen<B>();`.

- Cela favoriserait le polymorphisme (par sous-typage).
- On le fait bien avec les tableaux (`Object[] t = new String[10];`).
- C'est souvent conforme à l'intuition (cf. tableaux).

**Mais on sait que ça risque d'être difficile :**

- On a vu un contre-exemple pathologique (on provoque facilement `ClassCastException` si on force le compilateur à outrepasser l'invariance).
- On a vu les problèmes que posent les tableaux covariants (`ArrayStoreException` possible même dans programme sans *warning*).

1. Ou bien, peut.-être parfois, quand $B <: A$.

# Considérations autour de la variance (1)

Pour les quelques pages qui suivent, **oublions que `javac` impose l'invariance**.

**Question :** parmi les variables `x`, `y`, `z` et `t`, ci-dessous, lesquelles devrait-on, idéalement [1], pouvoir affecter à quelles autres ?

```
class A {}
class B extends A {}
// Interface pour fonctions F -> U (extraite de java.util.function) :
interface Function<T,U> { U apply(T t); }
class Test {
    Function<A,A> x;
    Function<A,B> y;
    Function<B,A> z;
    Function<B,B> t;
}
```

**Le critère :** on cherche les cas où une instance `Function<X,Y>` fournit au moins le service d'une instance de `Function<Z,T>`.

1. par exemple dans un langage où les génériques pourraient ne pas être invariants

# Considérations autour de la variance (2)

**Réponse :** affecter `u` à `v` a un sens si la méthode `apply` de `u` peut remplacer celle de `v` (en toute situation). C.-à-d. :

- si elle accepte tous les paramètres effectifs acceptés par celle-ci
- et si les valeurs retournées appartiennent à un type au moins aussi restreint.

(En résumé : une instance de `Function<X,Y>` peut remplacer une instance de `Function<Z,T>` si `X :> Z` et `Y <: T`.)

$\rightarrow$ en appliquant ce principe, on voudrait donc que le compilateur accepte :

```
z = t; z = x; t = y; x = y; z = y;
```

# Considérations autour de la variance (3)



**Représentation graphique :** type générique → pièce de puzzle.

- Paramètre utilisé en entrée (= type de paramètre de méthode ou type d'attribut public modifiable) → encoche.
- Paramètre utilisé en sortie (= type de retour de méthode, type d'attribut public quelconque)→ excroissance.

L'encoche (resp. excroissance) pour un type donné doit contenir celles de ses sous-types.

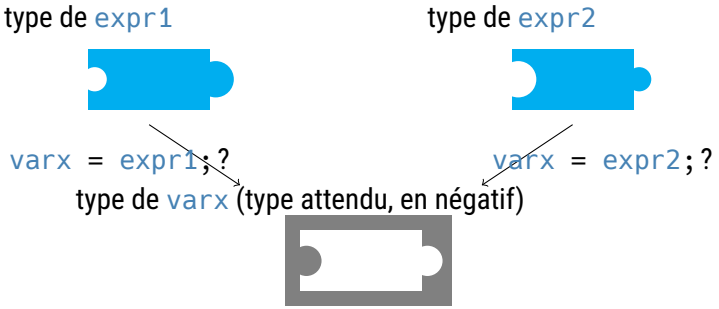
**Exemple :**

`Function<T, U>`

(L'encoche à gauche représente `T` et l'excroissance à droite, `U`.)

# Considérations autour de la variance (4)

Ainsi, inclusion des formes si et seulement s'il y a sous-typage :




→ seul `varx = expr2;` doit fonctionner [1] (pas de chevauchement) :

1. Attention, on ne parle pas de Java, mais seulement d'un système de type « idéal ».

# Considérations autour de la variance (5)



La variance souhaitée n'est donc pas la même pour tous les types génériques :

- intuitif et logique de vouloir
  `Function<Object, Integer>` <: `Function<Double, Number>`.
  **Justification :** le premier paramètre de type est utilisé uniquement pour l'argument de `apply` alors que l'autre est uniquement son type de retour.
  → Emboîtement de `Fuction<T, U>` dans `Function<V,W>` possible dès que `T` :> `V` et `U` <: `W`.

Remarque : tailles de l'encoche et de l'excroissance de `Fuction<T, U>` indépendantes l'une de l'autre car elles représentent 2 paramètres différents. Si le même paramètre de type est utilisé en entrée et en sortie, ça ne marche plus (cf. page d'après).

# Considérations autour de la variance (6)

- mais ~~`List<Integer>`~~ $\not<:$ ~~`List<Number>`~~ serait illogique.

  **Justification :** Le même paramètre apparait à la fois en sortie (méthode `get`) et en entrée (méthodes `set` et `add`) [1]. Donc tailles de l'encoche et de l'excroissance de `List<T>` liées car représentant le même `T`
  $\rightarrow$ impossible d'encastrer la pièce de `List<X>` dans le trou `List<Y>` si $X \neq Y$.

1. Même topo pour `Integer[]` $<:$ `Number[]` avec les opérations `x = t[i]` et `t[i] = x` (... mais ça c'est autorisé : en contrepartie, il est nécessaire de faire des vérifications à l'exécution, avec risque de `ArrayStoreException`).

# Encore un peu de vocabulaire autour de la variance (1)



Dans `Function<T, U>`, `T` et `U` ont des **influences contraires** l'une de l'autre à cause de leur usage dans la méthode de `Function`.

→ 2 catégories d'usage :

- en **position covariante** : utilisé comme type de retour de méthode (ou type d'attribut)
- en **position contravariante** : utilisé comme type d'un argument dans la signature d'une méthode (ou comme type d'un attribut non `final`)

# Encore un peu de vocabulaire autour de la variance (2)



→ 3 catégories de paramètres de type :

- **paramètre covariant** (comme U) : utilisé seulement en position covariante
  → plus le paramètre effectif est petit, plus le type paramétré devrait être petit ;
- **paramètre contravariant** (comme T) : utilisé seulement en position contravariante
  → plus le paramètre effectif est petit, plus le type paramétré devrait être grand ;
- **paramètre invariant** : utilisé à la fois en position covariante et contravariante.

**Attention, ces concepts ne sont que théoriques.**

Il se trouve que ceux-ci n'ont **pas de sens pour le compilateur de Java** : rappelez-vous qu'on avait dit que, pour l'instant, on oubliait l'invariance imposée par Java.

# Gérer la variance dans un programme



2 approches principales pour prendre en compte le phénomène de la variance :

- **annotations de variance sur site de déclaration** (n'existent pas en Java)
  Variance définie (définitivement) dans la déclaration du type générique.
  Exemple en langage Kotlin, on utilise **in** (contravariance) et **out** (covariance) :

```
interface Function<in T, out U> { fun apply(t: T) : U }
class A
class B : A
```

  Alors, dans cet exemple, `Function<A, B>` $<:$ `Function<B, A>`.[1]

- **annotations de variance sur site d'usage**
  Variance choisie lors de l'usage d'un type générique (dans déclarations de variables et signatures de méthodes).
  C'est l'approche utilisée par Java, via le mécanisme des ***wildcards***.

1. Le compilateur de Kotlin vérifie que les paramètres covariants (resp. contravariants) sont effectivement uniquement utilisés en position covariante (resp. contravariante).

# *Wildcards*

## principe de base (1)

**(On revient enfin à Java !)**

- Quand on écrit un type paramétré, les paramètres peuvent en fait être soit des types, soit le symbole « ? » (symbolisant un joker, un ***wildcard***), parfois muni d'une **borne**.
- Les types paramétrés dont le paramètre est compatible avec la borne du *wildcard* se comportent alors comme des sous-types du type contenant le *wildcard*.
  Ainsi `List<Integer>` est sous-type de `List<?>`.

**Remarque :** « ? » tout seul n'est pas un type. Ce caractère ne peut être utilisé que pour écrire un type paramétré (entre « < » et « > »).

# *Wildcards*

## principe de base (2)

- **Exemple de déclaration de méthode :**

  ```
  static double somme(List<? extends Number> liste){ ... }
  ```

  Cette méthode annonce pouvoir faire la somme des éléments d'une liste de n'importe quoi, tant que ce n'importe quoi est un sous-type de nombre.

  Dans ce cas, c'est équivalent à :

  ```
  static <T> double somme(List<T extends Number> liste) { ... }
  ```

- **Exemple de déclarations de variables :**

  ```
  C<? extends A> v1;
  C<? super B> v2;
  ```

  on peut alors affecter à `v1` (resp. `v2`) toute valeur de type `C<X>` pour peu que `X` soit sous type (resp. supertype) de `A` (resp. `B`).

Toute occurrence de « ? » peut se voir associer une borne.

Le principe est similaire aux bornes de paramètres de type, avec quelques différences :

- « ? » bornable à chaque usage (or, paramètres bornables juste à leur introduction).
- Les « ? » admettent des bornes supérieures (`T<? extends A>`), **mais aussi** des bornes inférieures (`T<? super A>`), imposant que toute concrétisation doit être un supertype de la borne.
- Pour un « ? », Java autorise une seule borne à la fois. [1]

1. Si on veut plusieurs types concrets comme bornes supérieures, il est possible de contourner cette limite en introduisant un type intermédiaire : `interface Borne extends Borne1, Borne2 ...`.
Combiner plusieurs bornes inférieures concrètes (disons A et B) ne sert à rien : en effet A et B ont nécessairement un plus petit supertype commun, C, qui a un nom déjà connu quand on écrit le programme. C est le plus petit type contenant $A \cup B$ (qui n'est pas un type de Java). Ainsi `T<? super C>` serait équivalent à `<? super A U B>` (syntaxe fictive).
Sinon, pour mixer des bornes qui sont elles-mêmes des paramètres, d'autres techniques basées sur l'introduction d'une variable de type supplémentaire sont envisageables.

# *Wildcards*

## Vérification du type

L'affectation suivante est-elle bien typée ?

```
List<? extends Serializable> l = new ArrayList<String>();
```

Pour savoir, on vérifie si le terme droite de l'affectation a un type compatible avec son emplacement.

Son type est `ArrayList<String>`, or le type attendu à cet emplacement est `List<? extends Serializable>` (= type de la variable à affecter).

- D'une part, `String` satisfait la borne de ? (`String` implémente `Serializable`)
- et, d'autre part, `ArrayList<String>` $<:$ `List<String>`.

Donc cette affectation est bien typée.

**Généralisons :**

- Soit une expression `expr` utilisées dans un certain contexte (appel de méthode, affectation, ...).
- Soit `TE` le type (déjà « converti par capture »[1], si applicable) de `expr`.
- Supposons que le type attendu dans le contexte soit de la forme `TA<? borne>`.
- Alors il est légal d'utiliser `expr` à cet endroit si et seulement si il existe un type `T` satisfaisant `borne`, tel que `TE <: TA<T>`.

1. Explication un peu plus loin. Ceci concerne le cas où le type de l'expression contient un ?.

# *Wildcards*

## utilisation pour signaler la variance

Pour revenir au problème initial, reprenons notre exemple :

```
interface Function<T,U> { U apply(T t); }
class A {}
class B extends A{}
class Test { Function<A,A> x; Function<A,B> y; Function<B,A> z; Function<B,B> t; }
```

U → position covariante ; T → position contravariante. On « assouplit » donc `Test` :

```
class Test2 {
    Function<? super A,? extends A> x; Function<? super A,? extends B> y;
    Function<? super B,? extends A> z; Function<? super B,? extends B> t;
}
```

Maintenant, les affectations qu'on voulait écrire sont acceptées par le compilateur :

```
z = t; z = x; t = y; x = y; z = y; // vérifiez !
/* et aussi... */ A a; B b; a = x.apply(a); b = y.apply(a); a = z.apply(b); b = t.apply(b);
```

**Recette :** position covariante → **`extends`**, position contravariante → **`super`**.
Se rappeler **PECS** : « producer extends, consumer super ».

Inversez **`super`** et **`extends`** et vérifiez que les appels à `apply` ne fonctionnent plus.

# *Wildcards*

## Conversion par capture (1)

En réalité, pour une expression, avoir le type `Gen<?>` veut dire qu'**il existe** [1] un type `QuelqueChose` (inconnu mais fixé) tel que cette expression est de type `Gen<QuelqueChose>`.

Il faut interpréter le type d'une expression à *wildcards* comme un type inconnu appartenant à l'ensemble des types respectant les contraintes trouvées.

1. Et comme on ne sait rien de ce type, vérifier que l'expression est à sa place c'est vérifier qu'elle est à sa place **pour toute** valeur de `QuelqueChose`.

# *Wildcards*

## Conversion par capture (2)

Concrètement, lors de la vérification de type d'une expression, le compilateur effectue une opération appelée **conversion par capture** [1] :

- À chaque fois qu'un « ? » apparaît au premier niveau [2] du type d'une expression [3], le compilateur le remplace par un nouveau type créé à la volée, recevant un nom de la forme `capture#i-of?` (**capture** de *wildcard*).
- Quand une telle capture est créée, le compilateur se souvient des bornes du « ? » qu'elle remplace (il peut s'en servir dans la suite de l'analyse de types).

---

1. Pour les logiciens, cette transformation s'apparente à la skolémisation : on remplace une variable quantifiée existentiellement par un nouveau symbole.
2. On ne regarde pas en profondeur : `List<? extends Set<?>>` devient `List<capture#1-of?>`, le compilateur se rappelant que `capture#1-of?` est sous-type de `Set<?>`.
3. Cela se produit quand l'expression est une variable typée avec des ?, un appel de méthode dont le type de retour contient des ?, ou une expression castée vers un tel type.

**Exemple :**

- soit une expression :
  `new HashMap<? super String, ? extends List<?>( )`,
- son type « brut » : `HashMap<? super String, ? extends List<?>>`,
- son type après conversion par capture :
  `HashMap<capture#1-of?, capture#2-of?>`.
- Le compilateur se rappelle que `capture#1-of? :> String` et que `capture#2-of? <: List<?>`.

Cette conversion a lieu à chaque fois que le type d'une expression est évalué. Ainsi, une expression composite peut contenir plusieurs captures différentes accumulées depuis l'analyse du type de ses sous-expressions.

# Wildcards

## Exemple pathologique

```
List<? super String> l = new ArrayList<>(); l.add("toto"); //ok
l.add(l.get(0)); // Mal typé ! Mais pourquoi ?
```

**Explication :** à la 2e ligne,

- la 1e occurrence de `l` est de type `List<capture#1-of?>` $\Longrightarrow$ `l.add(...)` attend un paramètre de type `capture#1-of?`;
- la 2e occurrence de `l` est de type `List<capture#2-of?>` (capture indépendante !) $\Longrightarrow$ `l.get(0)` est de type `capture#2-of?`;
- or `capture#1-of?` et `capture#2-of?` sont, du point de vue du compilateur, deux types quelconques sans lien de parenté, d'où l'erreur de type.

On peut contourner en forçant une capture anticipée (via méthode auxiliaire) :

```
// méthode auxilliaire. Ici, tout est ok, car l.get(0) de type T, or l.add() prend du T.
<T> static void aux(List<T> l) { l.add(l.get(0)); }
// plus loin
List<? super String> l = new ArrayList<>(); l.add("toto"); aux(l); // encore ok
```

# *Wildcards*

## Quid des exemples plus complexes ? (1)

Les « ? » peuvent apparaître à différentes profondeurs, y compris dans les bornes :

```
List<A<? super String>> las = new ArrayList<>();
List<? extends A<? super Integer>> lar = las;
```

Pour chaque niveau de <> on vérifie que le type (resp. l'ensemble de types) donné correspond à un élémént (resp. un sous-ensemble) de l'ensemble attendu.

# *Wildcards*

## Quid des exemples plus complexes (2) ?

```
List<A<? super String>> las = new ArrayList<>();
List<? extends A<? super Integer>> lar = las;
```

Dans l'exemple (2e ligne, à droite de =) :

- On veut comparer `List<A<? super String>>` (type reçu = celui de `las`) et `<? extends A<? super Integer>>` (type attendu = celui de `lar`).
- Au premier niveau, on a `List` et `List` $\rightarrow$ OK, vérifions les paramètres.
- Il faut que `A<? super String>` $<:$ `A<? super Integer>`.
- On a `A` des 2 côtés... jusque là tout va bien. Vérifions l'inclusion des paramètres.
- À l'intérieur on attend « `? super Integer` », mais on reçoit « `? super String` ».
- Les deux bornes sont dans le même sens, c'est bon signe.
- Malheureusement, on n'a pas `String` $:>$ `Integer`. Donc « `? super String` » n'est pas inclus dans « `? super Integer` » (p. ex. : le 1er ensemble contient `String` mais pas le 2e). Donc erreur de type !

## Definition (Concurrence)

Deux actions, instructions, travaux, tâches, processus, etc. sont **concurrents** si leurs exécutions sont **indépendantes** l'une de l'autre (l'un n'attend pas de résultat de l'autre).

- **Conséquence :** deux actions concurrentes peuvent s'exécuter simultanément, si la plateforme d'exécution le permet.
- Un **programme concurrent** est un programme dont certaines portions de code sont indépendantes les unes des autres et tel que la plateforme d'exécution sait [1] exploiter ce fait pour optimiser l'exécution. [2].

1. Le plus souvent, cette connaissance nécessite que les portions concurrentes soient signalées dans le code source.
2. Notamment en exécutant simultanément, **en parallèle**, ces portions de code si c'est possible.

# Concurrence

## Où et quand ?



- Naturelle et nécessaire dans des situations variées en programmation [1] :
  - **Serveurs web :** un même serveur doit pouvoir servir de nombreux clients indépendamment les uns des autres sans les faire attendre.
  - **Interfaces homme-machine :** le programme doit pouvoir, tout en prenant en compte, sans délai, les actions de l'utilisateur, continuer à exécuter d'éventuelles tâches de fond, jouer des animations, etc.
  - De manière générale, c'est utile dans tout programme qui doit réagir immédiatement à des évènements de causes et origines variées et indépendantes.
- Possible [2] pour de nombreux algorithmes décomposables en étapes indépendantes.

  Utile de programmer ces algorithmes de façon concurrente car on peut profiter du parallélisme pour les accélérer (cf. page suivante).

1. Et pas seulement en programmation, mais c'est le sujet qui nous intéresse !
2. Et discutablement naturelle aussi.

# Parallélisme

Deux travaux [1] s'exécutent **en parallèle**, s'ils exécutent en même temps.

- Simultanéité au niveau le plus bas : si 2 travaux s'exécutent en parallèle, à un instant t, s'exécutent en meme temps une instruction de l'un et de l'autre.
- $\rightarrow$ exécution sur 2 lieux physiques différents (*e.g.* 2 cœurs, 2 circuits, ...).
- **Degré de parallélisme** [2] = nombre de travaux simultanément exécutables.

Pour des raisons économiques et technologiques, les microprocesseurs modernes (multi-cœur [3]) ont typiquement un degré de parallélisme $\geq 2$.

C'est une opportunité qu'il faut savoir saisir !

1. Pour ne pas dire « processus », qui a un sens un peu trop précis en informatique.
2. D'une plateforme d'exécution.
3. Sur les CPU de moyenne et haut de gamme, le degré de parallélisme est généralement de 2 par cœur, grâce au SMT (*simultaneous multithreading*), appelé *hyperthreading* chez Intel

Ainsi, l'enjeu de la programmation concurrente est double :

- **Nécessité :** pouvoir programmer des fonctionnalités intrinsèquement concurrentes (serveur web, IG, etc.).
- **Opportunisme :** tirer partie de toute la puissance de calcul du matériel contemporain.
  En effet : des travaux indépendants (concurrents) peuvent naturellement être confiés à des unités d'exécution distinctes (parallèles).

Malheureusement, la programmation concurrente est un art difficile...

Exécuter deux travaux réellement concurrents en parallèle est facile [1], mais la réalité est souvent plus compliquée :

- Si (degré de) concurrence $>$ (degré de) parallélisme, alors **partage du temps** des unités d'exécution ($\rightarrow$ **ordonnanceur** nécessaire).
- Concurrence de 2 sous-programmes jamais parfaite [2] car nécessité de se transmettre/partager des résultats et de se **synchroniser**. [3]

$\rightarrow$ Différentes abstractions pour aider à programmer de façon correcte et, si possible, intuitive, tout en prenant en compte ces réalités de diverses façons.

1. On en affecte un à chaque cœur, pourvu qu'il y ait 2 cœurs disponibles, et on n'en parle plus !
2. Sinon ce ne seraient des sous-programmes mais des programmes indépendants à part entière !
3. En fait, ces deux aspects sont indissociables.

# Questions d'ordonnancement

## Multi-tâche préemptif vs. coopératif

Un **ordonnanceur** est un programme chargé de répartir les tâches concurrentes sur les unités d'exécutions disponibles. Il s'agit souvent d'un sous-système du noyau de l'OS [1].

L'ordonnanceur peut mettre en œuvre :

- un fonctionnement **multi-tâches préemptif** : l'ordonnanceur choisit quand mettre en pause une tâche pour reprendre l'exécution d'une autre. Cela peut arriver (presque) à tout moment.
  *C'est le cas pour la gestion des processus dans les OS modernes pour ordinateur personnel.*
- ou bien un fonctionnement **multi-tâches coopératif** : chaque tâche signale à l'ordonnanceur quand elle peut être mise en attente (par exemple en faisant un appel bloquant).

1. *Operating System*/système d'exploitation

# Questions de synchronisation

## Transmettre des résultats d'une tâche à l'autre



Plusieurs techniques de transmission de résultats :

- **variables partagées** : variables accessibles par plusieurs tâches concurrentes. Données partagées de façon transparente, sans synchronisation a priori, mais le langage permet d'insérer des primitives de synchronisation explicite [1].
- **passage de message** : données « envoyées » [2] d'une tâche à l'autre. Synchronisation implicite de l'émision et de la réception du message : par exemple, une tâche en attente de réception est bloquée tant qu'elle n'a rien reçu. [3]

La réalité physique est plus proche du modèle des variables partagées [4], mais le passage de message est un paradigme plus sûr [5].

1. En Java : `start()`, `join()`, **`volatile`**, **`synchronized`** et `wait()`/`notify()`.
2. Sous-entendu : l'envoyeur ne peut plus accéder à ce qui a été envoyé.
3. C'est une possibilité. On peut aussi bloquer la tâche émettrice (canal borné, « rendez-vous »).
4. Mémoire centrale lisible par plusieurs CPU.
5. Pour lequel la sûreté d'un programme est plus facile à prouver.

# Questions de synchronisation

## Se transmettre des résultats



Mais on peut simuler le passage de message :

```java
public final class MailBox<T> { // classe réutilisable, simulant un passage de message avec "rendez-vous"
    private T content; // mémoire partagée, encapsulée
    public synchronized void sendMessage(T message) throws InterruptedException {
        while (content != null) wait(); // attend la condition content != null
        content = message;
        notifyAll(); // débloque les (autres) threads en attente sur cette MailBox
    }
    public synchronized T receiveMessage() throws InterruptedException {
        while (content == null) wait(); // attend la condition content == null
        T ret = content; content = null;
        notifyAll(); // débloque les (autres) threads en attente sur cette MailBox
        return ret;
    }
}

public final class PingPong {
    public static void main(String[] args) {
        var box = new MailBox<String>();
        new Thread(() -> {
            try { while(true) box.sendMessage("ping!"); }
            catch (Exception e ) { throw new RuntimeException(e); }
        }).start(); // producteur/écrivain
        new Thread(() -> {
            try { while(true) System.out.println((box.receiveMessage() == "ping!")?"pong!":"error!"); }
            catch (Exception e ) { throw new RuntimeException(e); }
        }).start(); // consommateur/lecteur
    }
}
```

# Questions de synchronisation

## Recevoir des messages



- **fonctions bloquantes** : la tâche réceptrice appelle une fonction fournie par la bibliothèque, qui la bloque jusqu'à ce que la valeur attendue soit disponible.

```
ForkJoinTask<Result> task = ForkJoinTask.adapt(() -> {
    ... // tâche 1
    return result;
}).fork();
...
// plus loin
Result x = task.join(); // appel à fonction bloquante join()
... // tâche 2 : fait qqc avec le résultat x de tâche 1
```

- **fonctions de rappel** (*callbacks*) : on passe à la bibliothèque une fonction que celle-ci appellera sur le résultat attendu dès qu'il sera disponible.

```
CompletableFuture.supplyAsync(() -> {
    ... // tâche 1
    return result;
}).thenApply((x) -> { // corps de la fonction de rappel
    ... // tâche 2 : fait qqc avec le résultat x de tâche 1
});
```

# Questions de synchronisation

## Envoyer des messages



Envoyer le résultat `x` d'un calcul, ça peut être simplement :

- retourner `x` à la fin d'une fonction (tâche productrice), c'est le cas dans les 2 exemples précédents (« `return result` »).
- passer `x` en paramètre d'un appel de méthode. Par exemple, on peut soumettre la valeur à une file d'attente synchronisée :

```
... // calcule x
queue.offer(x);
... // fais autre chose (avec interdiction de toucher à  x !)
```

Dans ce dernier cas, la tâche consommatrice reçoit le message en appelant `queue.take()` (fonction bloquante).

Remarque : cela est similaire à l'exemple de la classe `MailBox` donné plus tôt [1].

1. Différence : `MailBox` ne stocke qu'un seul message (= « Rendez-vous »), alors qu'une file d'attente peut en stocker plusieurs, permettant au consommateur et au producteur de ne pas suivre le même rythme.

# Notion de *thread*

## Definition (***Thread*** ou **fil d'exécution**)

Abstraction concurrente consistant en une séquence d'instructions dont l'exécution **simule une exécution séquentielle** (en interne) [1] et **parallèle** à celle des autres *threads*.

- Un nombre quelconque de *threads* s'exécute sur une plateforme de degré de parallélisme quelconque [2]. Un ordonnanceur partage les ressources de la plateforme pour que cela soit possible.
- Ainsi, *n threads* en exécution simultanée simulent un parallélisme de degré *n*
- Un **processus** [3] (= 1 application en exécution) peut utiliser plusieurs *threads* qui ont accès aux mêmes données (mémoire partagée).

1. Ce qui permet de le programmer avec les principes habitules de programmation impérative : séquences d'instructions, boucles, branchements, pile d'appels de fonctions, ...
2. Même inférieur au nombre de *threads*
3. Cette fois-ci au sens où on l'entend en informatique.

# Notion de *thread*

## Exemple simple (1)

Exemple de deux *threads*, l'un qui compte jusqu'à 10 alors que l'autre récite l'alphabet :

```
class ReciteNombres extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++)
            System.out.print(i + " ");
    }
}

class ReciteAlphabet extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 26; i++)
            System.out.print((char)('a'+i) + " ");
    }
}
```

# Notion de *Thread*

## Exemple simple (2)



### Alors

```
public class Exemple {
    public static void main(String[] args) {
        new ReciteNombres().start(); new ReciteAlphabet().start();
    }
}
```

### peut afficher

```
0 1 2 3 4 5 6 7 8 9 a b c d e f g h i j k l m n o p q r s t u v w x y z
```

### mais également

```
0 1 2 3 a b c d 4 5 e 6 f 7 g 8 h 9 i j k l m n o p q r s t u v w x y z
```

### ou encore

```
0 1 2 3 4 a 5 b 6 c 7 d 8 e 9 f g h i j k l m n o p q r s t u v w x y z
```

# Notion de *thread*

## Un concept décliné à plusieurs niveaux (1)



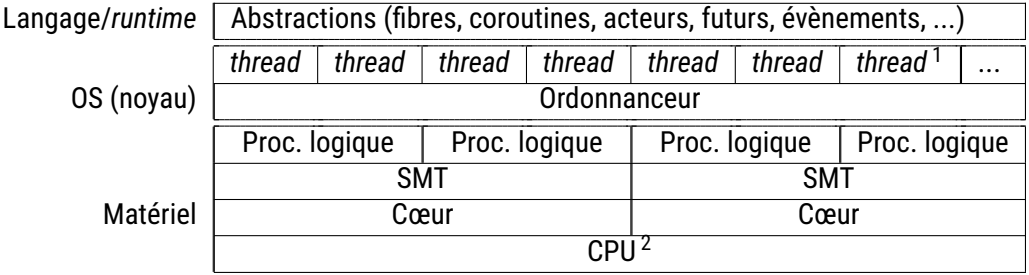
- **Dans le matériel :** p. ex., dans les processeurs *Intel Core i7*, un même cœur exécute 2 *threads* simultanés pour pouvoir utiliser optimalement tous les composants du *pipeline* (SMT/*hyperthreading*).
  Ces 2 *threads* sont présentés à l'OS comme des processeurs séquentiels à part entière (ainsi un *i7* à 4 cœurs, apparaît, pour le système, comme 8 processeurs).
- **Dans les OS multi-tâches :** afin que plusieurs logiciels puissent s'exécuter en même temps, un OS est capable d'instancier un « grand » [1] nombre de *threads* (on parle de « ***threads* système** »).
  L'OS contient un ordonnanceur affectant tour à tour les *threads* aux différents processeurs [2] en gérant les changements de contexte. [3]

1. On parle typiquement de milliers, pas de millions. La limite pratique est la mémoire disponible.
2. réels ou simulés, cf. *hyperthreading*
3. Contexte = pointeur de pile, pointeur ordinal, différents registres...

# Notion de *thread*

## Un concept décliné à plusieurs niveaux (2)

- **Dans le *runtime* des langages de programmation :** des langages de programmation (*Erlang, Go, Haskell, Lua,* ..., mais pas actuellement Java), contiennent une notion de *thread* « léger » (différents noms : *green thread*, fibre, coroutine, goroutine, ...), s'exécutant par dessus un ou des *threads* système.




1. Sous-entendu : « *thread* système » (« *thread* » sans précision = « *thread* système »).
2. Possible aussi : plusieurs CPUs (plusieurs cœurs par CPU, plusieurs processeurs logiques par cœur...)

# À propos des *threads* système

## Avantages et inconvénients



- Ce sont des *threads*.
  **Avantage :** se programment séquentiellement (respectent les habitudes).
  **Inconvénient :** la synchronisation doit être explicitée par le programmeur. [1]
- Multi-tâche préemptif : l'ordonnanceur peut suspendre un *thread* (au profit d'un autre), à tout moment
  **Avantage :** pas besoin de signaler quand le programme doit « laisser la main ».
  **Inconvénient :** changements de contexte fréquents et coûteux.
- Implémentation dans le noyau :
  **Avantage :** compatible avec tous les exécutables de l'OS (pas seulement JVM)
  **Inconvénient :** fonctionnalités rudimentaires. P. ex., chaque *thread* a une pile de taille fixe (1024 ko pour les *threads* de la JVM 64bits) → peu économique !

1. On va voir dans la suite comment. Pour l'instant, retenez qu'il n'y a aucune synchronisation, donc aucun partage de données sûr entre *threads* si on n'ajoute pas « quelque chose ».

# Des APIs de haut niveau pour ne pas manipuler les *threads*

$\rightarrow$ les langages de programmation proposent des mécanismes, utilisant les *threads* système, pour pallier leurs inconvénients tout en essayant [1] de garder leur avantages.

Au moins deux objectifs :

- limiter le nombre de *threads* système utilisés, afin de diminuer l'empreinte mémoire et la fréquence des changements de contexte
- forcer des procédés sûrs pour le partage de données ; ou à défaut, faciliter les bonnes pratiques de synchronisation.

1. avec plus ou moins de succès

# La situation de Java

Java

- utilise directement les *threads* système, via la classe `Thread`.
- a historiquement (Java 1.1) utilisé des *green threads* [1], abandonnés pour des raisons de performance [2].
- pourrait néanmoins, dans le futur, supporter les fibres [3] via le projet Loom.
- dispose actuellement d'un grand nombre d'APIs facilitant où rendant plus sûre l'utilisation des *threads* : les boucles d'évènements Swing et JavaFX, `ThreadPoolExecutor`, `ForkJoinPool`/`ForkJoinTask`, `CompletableFuture`, `Stream`...

---

1. Une sorte de *threads* légers.
2. Ils étaient ordonnancés sur un seul *thread* système, empêchant d'utiliser plusieurs processeurs.
3. Autre type de *threads* légers. Cette fois-ci, le travail peut être distribué sur plusieurs *threads* système. Des implémentations de fibres pour Java existent déjà : bibliothèques Quasar et Kilim. Mais pour fonctionner, celles-ci doivent modifier le code-octet généré par `javac`.

# *Threads* en Java

- 1 *thread* est associé à 1 pile d'appel de méthodes
  *Thread* principal en Java = pile des méthodes appelées depuis l'appel initial à `main()`
  → **vous utilisez déjà des *threads* !**
- Interfaces graphiques (Swing, JavaFX, ...) : un *thread* [1] ($\neq$ `main`) est dédié aux évènements de l'IG :
  - Programmation événementielle → méthodes gestionnaires d'événement
  - Événements → mis en file d'attente quand ils surviennent.
  - Quand le *thread* des évènements est libre, le gestionnaire correspondant au premier événement de la file est appelé et exécuté sur ce *thread*.

  **Intérêt :** [2] pas besoin de prévoir des interruptions régulières dans le *thread* `main` pour vérifier et traiter les événements en attente (l'IG resterait figée entre deux)

1. Pour Swing : *Event Dispatching Thread* (EDT). Pour JavaFX : *JavaFX Application Thread*.
2. Et l'intérêt de n'avoir qu'un seul thread pour cela : la sûreté du fonctionnement de l'IG. Pas d'entrelacements entre 2 évènements, pas d'accès compétition.

- Tous les *threads* ont accès au même tas (mêmes objets) et à la même zone statique (mêmes classes)... mais pas à la même pile !
- Les *threads* communiquent grâce aux **variables partagées**, stockées dans le tas.
- Une même méthode peut être appelée depuis n'importe quel *thread* (pas de séparation syntaxique du code associé aux différents *threads*).
- Pour démarrer un *thread* : `unObjetThread.start();` (où `unObjetThread` instance de la classe `Thread`).
  → aussitôt, appel de `unObjetThread.run()` dans le *thread* associé à cet objet.
- À chaque *thread* correspond une pile d'appels de méthode. En bas de la pile :
  - pour le *thread* `main`, le *frame* de la méthode `main` ;
  - pour les autres, celui de l'appel initial à `run` sur l'objet représentant le *thread*.

# Créer un *thread* en Java

## Quelques techniques classiques pour créer un *thread*



- Définir et instancier une classe héritant de la classe `Thread` :

```
public class HelloThread extends Thread {
    @Override public void run() { System.out.println("Hello from a thread!"); }
}
// plus loin
        new HelloThread().start();
```

- Implémenter `Runnable` et appeler le constructeur `Thread(Runnable target)` :

```
public class HelloRunnable implements Runnable {
    @Override public void run() { System.out.println("Hello from a thread!"); }
}
// plus loin
        new Thread(new HelloRunnable()).start();
```

- Mais pour un *thread* simple, on préfèrera écrire une lambda-expression :

```
        new Thread(() -> { System.out.println("Hello from a thread!"); }).start();
```

# Créer un *thread* en Java

## Remarques



- L'interface `Runnable` a pour seule méthode (abstraite) `void run()`.
  Cette interface n'a, *a priori*, aucun rapport avec les *threads*, mais :
  - ses instances sont souvent passées au constructeur de `Thread` pour programmer leur exécution sur un nouveau *thread* ;
  - `Thread` implémente `Runnable` (et possède d'autres méthodes, voir la suite) ;
  - la méthode `run` de `Thread` appelle la méthode `run` du `Runnable` passé en paramètre (le cas échéant). [1]
- L'approche consistant à définir des tâches en implémentant directement `Runnable` plutôt qu'en étendant `Thread` laisse la possibilité d'hériter d'une autre classe :

```
public class MaClasse extends JFrame implements Runnable {
    public void run() {
        ...
    }
    ...
}
```

1. ⇒ `Thread` est ainsi un décorateur de `Runnable`.

# La classe Thread

## Méthodes importantes



- `String getName()` : récupérer le nom d'un thread.
- `void join()` : attendre la fin de ce *thread* (voir synchronisation).
- `void run()` : la méthode qui lance tout le travail de ce Thread. C'est la méthode qu'il faudra redéfinir à chaque fois que `Thread` sera étendue !.
- `static void sleep(long millis)` : met le *thread* courant (i.e. en cours d'exécution) en pause pendant tant de ms. (NB : c'est une méthode `static`. Le *thread* mis en pause est celui qui appelle la méthode. Il n'y a pas de `this`!).
- `void start()` : démarre le *thread* (conséquence : `run()` est exécutée dans le nouveau thread : celui décrit par l'objet, pas celui de l'appelant!)..
- `void interrupt()` : interrompt le *thread* (déclenche `InterruptedException` si le *thread* était en attente sur `wait()`, `join()`, `sleep()`,..)
- `static boolean interrupted()` : teste si un autre *thread* a demandé l'interruption du *thread* courant.
- `Thread.State getState()` : retourne l'état du *thread*.

# La classe Thread

## États d'un *thread*

Une instance de *thread* est toujours dans un des états suivants :

- `NEW` : juste créé, pas encore démarré.
- `RUNNABLE` : en cours d'exécution.
- `BLOCKED` : en attente de moniteur (voir la suite).
- `WAITING` : en attente d'une condition d'un autre *thread* (voir notify()/wait()).
- `TIME_WAITING` : idem pour attente avec temps limite.
- `TERMINATED` : exécution terminée.

Mais attendons la suite pour en dire plus sur ces états...

# La classe Thread

## Interruptions (1)



- Si `t` est un *thread*, l'appel `t.interrupt()` demande l'interruption de celui-ci.
- Si `t` est en train d'exécuter une méthode interruptible [1], celle-ci quitte tout de suite.
- L'interruption est propagée le long des méthodes de la pile d'appel qui quittent une à une... jusqu'à la méthode principale de la tâche [2] qui quitte aussi.
- Le résultat (non garanti [3]) est la terminaison de la tâche exécutée sur `t` [4].
- La propagation de l'interruption est implémentée par la propagation de l'exception `InterruptedException` et par le contrôle du booléen `Thread.interrupted()` (détails juste après).

1. C'est le cas de toutes les méthodes bloquantes de l'API `Thread` : `wait()`, `sleep()`,`join()`...
2. Habituellement : `run`.
3. Si les méthodes exécutées sur `t` n'ont pas prévu d'être interrompues, rien ne se passe.
4. Si exécution directe dans le *thread*, terminaison du *thread*, sinon, si exécution dans un *thread pool*, le *thread* est juste rendu de nouveau disponible.

# La classe Thread

## Interruptions (2)

Pour écrire une méthode interruptible `f` :

- Quand une interruption est détectée la bonne pratique est de quitter (**`return`** ou **`throw`**) au plus tôt, tout en libérant les ressources utilisées.
- L'interruption peut être détectée de deux façons :
  - soit une méthode auxiliaire `g` appelée depuis `f` quitte sur `InterruptedException`
  - soit on a obtenu **`true`** en appelant `Thread.interrupted()`.
- Le premier cas (exception) doit être traité en mettant tout appel à `g` dans un block **`try`**/**`finally`** (libération explicite des ressources de `f` dans le **`finally`**) ou bien *try-with-resource* (libération implicite).
- Remarque : il faut absolument vérifier `Thread.interrupted()` dans toute boucle de `f` ne faisant pas d'appel à une méthode interruptible comme `g`.
- Dans tous les cas, il faut veiller à propager le statut « interrompu » au contexte d'exécution, pour qu'il puisse, lui aussi, prendre en compte le fait qu'une interruption a eu lieu. 2 cas de figure (voir la suite).

# La classe Thread

## Interruptions (3)



2 cas de figure, selon que la signature de `f` est imposée ou non :

- Si ce n'est pas le cas, on propage le statut « interrompu » en quittant sur `InterruptedException`. 2 cas de figure :
  - si une méthode appelée depuis `f` a elle-même lancé `InterruptedException` : dans ce cas on ne met pas de **`catch`** et la propagation est automatique.
  - sinon, on peut ajouter **`throw new`** `InterruptedException();`

  `InterruptedException` étant une exception sous contrôle, il faut aussi ajouter **`throws`** `InterruptedException` à la signature de `f`.
- Sinon, si la signature de `f` est imposée par l'interface implémentée (ex : `Runnable`) et ne contient pas **`throws`** `InterruptedException`, on ne peut alors pas quitter sur `InterruptedException`.

  Solution : avant **`return`** on appelle `System.currentThread().interrupt()` (ce qui fait que le prochain appel à `interrupted()` retournera bien **`true`**).

## Exemples de méthodes interruptibles :

```
// avec while et acquisition/libération de resource (bloc "try-with-resource")
Data f(Data x) throws InterruptedException {
    try (Scanner s = new Scanner(System.in)) {
        while(test(x)) {
            x = transform(x, s.next());
            if (Thread.interrupted()) throw new InterruptedException(); // <-- ici !
        }
        return x;
    } // s.close() appelée implicitement à  la sortie du bloc (par throw ou par return)
}

// exemple sans boucle, mais avec appel bloquant
void sleep5s() throws InterruptedException {
    System.out.println("Acquisition potentielle de ressource");
    try {
        Thread.sleep(5000); // on attend 5s
    } finally { System.out.println("Libération de la même ressource"); }
    // Pas de "catch". Si sleep() envoie InterruptedException, elle est propagée.
}
```

# Problèmes liés au *multithreading*

Deux principaux problèmes :

1. **Les entrelacements non maîtrisés** : les instructions de 2 threads **s'entrelacent** [1] et accèdent (lecture et écriture) aux mêmes données dans un ordre imprévisible.
   Ce phénomène est « naturel » (l'ordonnanceur est libre de faire avancer un *thread*, puis l'autre au moment où il veut) ; il est parfois gênant, parfois non.
2. **Les incohérences dues aux optimisations matérielles** [2] : la JVM [3] laisse une marge d'interprétation assez large au matériel pour qu'il puisse exécuter le programme efficacement. Principales conséquences :
   - ordre des instructions donné dans le code source pas forcément respecté
   - modifications de variables partagées pas forcément vues par les autres *threads*.

Pour l'instant, concentrons nous sur le problème 1.

1. *interleave*
2. en particulier dans le microprocesseur
3. La JVM s'appuie sur le **JMM** : *Java Memory Model*, un modèle d'exécution relativement laxe.

# Entrelacements de threads

## Exemple (1)



Qu'est-ce qui est affiché quand on exécute le programme suivant ?

```
public class ThreadInterferences extends Thread {
    static int x = 0;

    public ThreadInterferences(String name){ super(name); }

    @Override
    public void run() {
        while(x++ < 10) System.out.println("x incrémenté par " + getName() + ", sa
             nouvelle valeur est " + x + ".");
    }

    public static void main(String[] args){
        new ThreadInterferences("t1").start();
        new ThreadInterferences("t2").start();
    }
}
```

On s'attend à voir tous les entier de 1 à 10 s'afficher dans l'ordre.

# Entrelacements de threads

## Exemple (2)



Exécution possible :

```
x incrémenté par t2, sa nouvelle valeur est 2.
x incrémenté par t1, sa nouvelle valeur est 2.
x incrémenté par t2, sa nouvelle valeur est 3.
x incrémenté par t1, sa nouvelle valeur est 4.
x incrémenté par t2, sa nouvelle valeur est 5.
x incrémenté par t1, sa nouvelle valeur est 6.
x incrémenté par t2, sa nouvelle valeur est 7.
x incrémenté par t2, sa nouvelle valeur est 9.
x incrémenté par t2, sa nouvelle valeur est 10.
x incrémenté par t1, sa nouvelle valeur est 8.
```

Contrairement à ce qu'on pourrait attendre : les nombres ne sont pas dans l'ordre, certains se répètent, d'autres n'apparaissent pas.

# Accès atomique

Avec quelle granularité les entrelacements se font-ils ? Peut-on s'arrêter au milieu d'une affectation, faire autre chose sur la même variable, puis finir ? → notion clé : **atomicité**

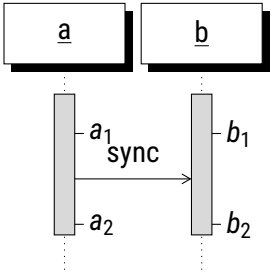
- **Atomique** = non séparable (étym.), non entrelaçable (ici).
  Aucune autre instruction, accédant aux mêmes données, ne peut être exécutée pendant celle des instructions d'une opération atomique.
- Quelques exemples d'opérations atomiques :
  - lecture ou affectation de valeur 32 bits (`boolean`, `char`, `byte`, `short`, `int`, `float`) ;
  - lecture ou affectation de référence (juste la référence, pas le contenu de l'objet) ;
  - lecture ou affectation d'attribut `volatile` [1] ;
  - exécution d'un bloc `synchronized` [2]
- Exemple d'opération non atomique : `x++` (peut se décomposer ainsi : copie `x` en pile, empile `1`, additionne, copie le sommet de pile dans `x`).

1. Notion abordée plus loin.
2. Idem. Dans ce cas, remplacer « accédant aux mêmes données » par « utilisant le même verrou ».

# La synchronisation

**Synchronisation :**

- consiste, pour un *thread*, à attendre le « feu vert » d'un autre *thread* avant de continuer son exécution ;
- interdit certains entrelacements ;
- contribue à établir la relation "arrivé-avant", limitant les optimisations autorisées [1].




Ici, $a_1$ arrive avant $a_2$, $b_1$ avant $b_2$, $a_1$ avant $b_2$, **mais pas $b_1$ avant $a_2$ !**

1. À suivre...

# Avec la classe Thread : méthode `join()`

Synchronisation simple : attendre la terminaison d'un *thread* avec `join()` [1] :

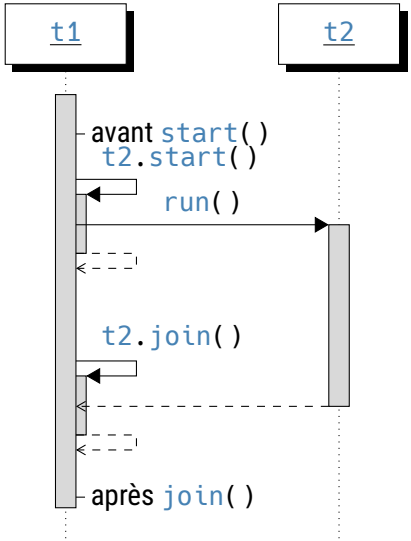
```
public class ThreadJoin extends Thread {
    static int x = 0;

    @Override
    public void run(){ System.out.println(x); }

    public static void main(String[] args){
        Thread t = new ThreadJoin();
        t.start();
        t.join();
        x++;
    }
}
```

affiche `0` alors que le même code sans l'appel à `join()` affichera probablement `1`.

Tout ce qui est exécuté dans le *thread* `t` arrive-avant ce qui suit le `join()` dans le *thread* `main` (ici, l'incrémentation de `x`).

1. Existe aussi en version temporisée : on bloque jusqu'au délai donné en paramètre maximum.

# Synchronisation avec `join()`



# Le verrou intrinsèque

## Qu'est-ce, à quoi cela sert-il ?



- En Java tout objet contient un **verrou intrinsèque** (ou **moniteur**).
- À tout moment, le moniteur est soit libre, soit détenu par un (seul) *thread* donné. Ainsi un moniteur met en œuvre le principe d'exclusion mutuelle.
- Lors de son exécution, un *thread* `t` peut demander à prendre un moniteur.
  - Si le moniteur est libre, alors il est pris par `t`, qui continue son exécution.
  - Si le moniteur est déjà pris, `t` est alors mis en attente jusqu'à ce que le moniteur se libère pour lui (il peut y avoir une liste d'attente).
- Un *thread* peut à tout moment libérer un moniteur qu'il possède.

**Conséquence :** tout ce qui se produit dans un *thread* avant qu'il libère un moniteur arrive-avant ce qui se produit dans le prochain *thread* qui obtiendra le moniteur, après l'obtention de celui-ci.

# Le verrou intrinsèque

## Utilisation en pratique : le mot-clé **synchronized**



### Bloc synchronisé :

```
class AutreCompteur{
    private int valeur;
    private Object verrou = new Object(); // peu importe le type déclaré
    public void incr(){
        synchronized(verrou) { //    <--- ici !
            valeur++;
        }
    }
}
```

**Sémantique :** le *thread* qui exécute ce bloc demande le moniteur de `verrou` en y entrant et le libère en en sortant.

**Conséquence :** pour une instance donnée de `AutreCompteur`, le bloc n'est exécuté que par un seul *thread* en même temps (exclusion mutuelle). Les autres *threads* qui essayent d'y entrer sont suspendus (`BLOCKED`).

# Le verrou intrinsèque

## Utilisation en pratique : le mot-clé **synchronized**



**Méthode synchronisée :** cas particulier avec synchronisation de tout le corps de la méthode sur moniteur de `this`. → syntaxe plus légère, plus souvent utilisée en pratique.

```
class Compteur {
    private int valeur;
    // méthode contenant bloc synchronisé
         sur this
    public void incr(){
        synchronized(this) { valeur++; }
    }
}
```

équivalent à...

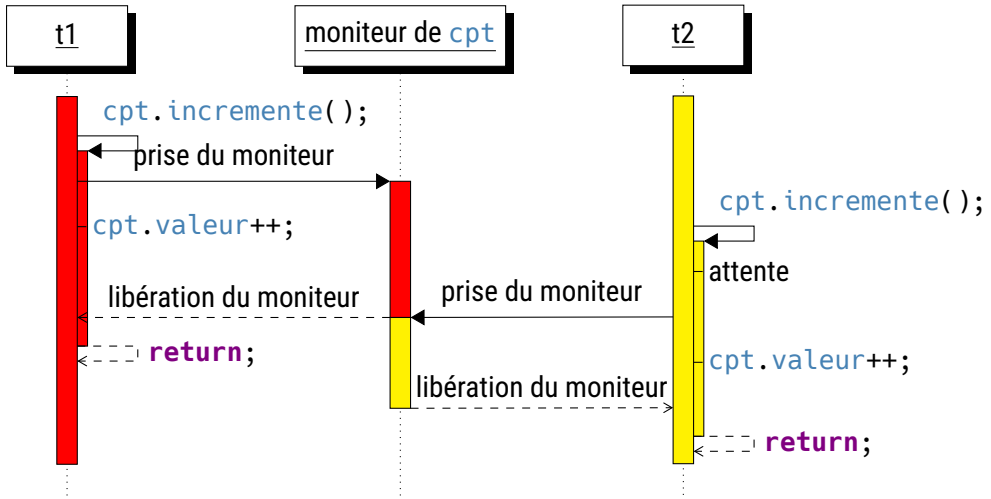
```
class Compteur {
    private int valeur;
    // méthode synchronisée
    public synchronized void incr() {
        valeur++;
    }
}
```

**Note :** l'exclusion mutuelle porte sur le moniteur (1 par objet) et non sur le bloc synchronisé (souvent plusieurs par moniteur).

**Conséquence :** 1 bloc synchronisé n'a qu'une seule exécution simultanée. De plus, aucun autre bloc synchronisé sur le même moniteur ne sera exécuté en même temps.

# Le verrou intrinsèque

```
Compteur cpt = new Compteur();
new Thread(cpt::incremente).start(); new Thread(cpt::incremente).start();
```



# Le verrou intrinsèque

## Le mécanisme `notifyAll()`/`notify()`/`wait()`

- 3 méthodes concernées (classe `Object`) : `notify()`, `notifyAll()` et `wait()`.
- Ces méthodes sont appelables seulement dans un bloc synchronisé sur l'objet récepteur de l'appel : `synchronized(x){ x.wait(); }`.
- `wait()` : met le *thread* en sommeil et libère le moniteur (`getState()` passe de `RUNNABLE` à `WAITING`).
  Le *thread* restera dans cet état tant qu'il n'est pas réveillé (par `notifyAll()` ou `notify()`). Il sera alors en attente pour récupérer le moniteur (`WAITING` → `BLOCKED`).
- `notifyAll()` : réveille tous les threads en attente sur l'objet. Ceux-ci deviennent candidats à reprendre le moniteur quand il sera libéré.
- `notify()` : réveille un *thread* en attente sur l'objet.

# Le verrou intrinsèque

## Le mécanisme `notifyAll()`/`notify()`/`wait()` – conseil d'utilisation



- On utilise `wait()` pour attendre une condition `cond`.
- Mais plusieurs *threads* peuvent être en attente. Un autre pourrait être libéré et récupérer le moniteur avant, rendant la condition à nouveau fausse.
- → aucune garantie que `cond` soit vraie au retour de `wait()`.

Ainsi, il faut tester à nouveau jusqu'à satisfaire la condition :

```
synchronized(obj) { // conseil : mettre wait dans un while
    while(!condition(obj)) obj.wait();
    ... // insérer ici instructions qui avaient besoin de condition()
}
```

**Il faut absolument retenir la formule ci-dessus !!!**
(utilisée dans 99,9% des cas d'usage corrects de `wait`... )

**Variantes acceptables :**

- `while(!condition())Thread.sleep(temps);`
  → utile quand on sait qu'aucun *thread* ne notifiera quand la condition sera vraie.
- `while(!condition())Thread.onSpinWait();` (Java $\geq$ 9) : **attente active** (c'est-à-dire : ni blocage ni attente, le *thread* reste `RUNNABLE`).
  → on évite le coût de la mise en attente et du réveil, cette approche est donc conseillée quand on s'attend à ce que la condition soit vraie très vite.

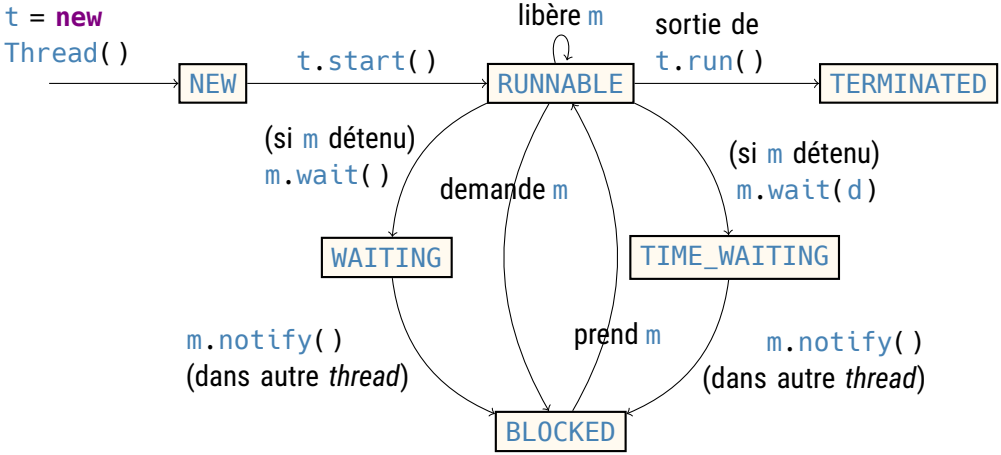
**Déconseillé** [1] **:** ~~`while(!condition(obj)/*rien*/;`~~ : attente active « bête »

→ c'est l'ancienne façon de faire, remplacée avantageusement par la variante avec `onSpinWait`. En effet, `onSpinWait` signale à l'ordonnanceur que le *thread* peut être mis en pause (laisser sa place sur le processeur) prioritairement en cas de besoin.

1. Sauf pour faire cuire une omelette sur son microprocesseur...

# Retour sur les états d'un *thread*

État = une valeur dans `Thread.State` (rectangles) + ensemble de moniteurs détenus




En réalité, racourcis directement vers `RUNNABLE` plutôt que `BLOCKED` quand le moniteur est déjà disponible.

# Verrous explicites (1)

- moniteurs = principe d'exclusion mutuelle + mécanisme d'attente/notification ;
- mais il existe d'autres façons de synchroniser des *threads* par rapport à l'usage d'une resource (exemple : lecteurs/rédacteur) ;
- fonctionnalités possibles : savoir à qui appartient le verrou, qui est en attente, etc. ;
- → bibliothèque de verrous divers dans `java.util.locks`, implémentant l'interface `java.util.concurrent.locks.Lock`.

L'interface `java.util.concurrent.locks.Lock` :

```
public interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    Condition newCondition();
    boolean tryLock();
    boolean tryLock(long time, TimeUnit unit) throws InterruptedException;
    void unlock();
}
```

# Verrous explicites (2)

Comme le verrouillage et le déverrouillage se font par appels explicites aux méthodes `lock` et `unlock`, ces verrous sont appelés **verrous explicites**.

**Inconvénient :** l'occupation du verrou n'est pas délimitée par un bloc lexical tel que `synchronized { ... }`[1].
La logique du programme doit assurer que toute exécution de `lock` soit suivie d'une exécution de `unlock`.

**Avantages :**

- Nombreuses options de configuration.
- Flexibilité dans l'ordre d'acquisition et de libération. [2]

1. Mais on peut programmer un tel bloc à la main à l'aide d'une fonction d'ordre supérieur, et encapsuler un tel verrou dans une classe dont l'interface ne permettrait d'acquérir le verrou que via cette FOS.
2. *Concurrent Programming in Java* (2.5.1.4) montre un exemple de liste chaînée concurrente où, lors d'un parcours, il est nécessaire d'exécuter une chaîne d'acquisitions/libérations croisées de la forme :
`m1.lock() ; ... ; m2.lock() ; m1.unlock() ; ... ; m3.lock() ; m2.unlock() ; ...; m4.lock() ; m3.unlock() ; ... ; m5.lock() ; m4.unlock() ; ... .`

# Dangers de la synchronisation

## quand elle est utilisée à mauvais escient ou à l'excès



**Un dernier avertissement :** la synchronisation doit rester raisonnable !

En général, plus il y a de synchronisation, moins il y a de parallélisme... et plus le programme est ralenti. Pire, il peut bloquer.

**Pathologies typiques :**

- ***dead-lock*** : 2 threads attendent chacun une ressources que seul l'autre serait à même de libérer (en fait 2 ou plus : dès lors que la dépendance est cyclique).
- **famine** (*starvation*) : une ressource est réservée trop souvent/trop longtemps toujours par la même tâche, empêchant les autres de progresser.
- ***live-lock*** : boucle infinie causée par plusieurs threads se faisant réagir mutuellement, sans pour autant faire avancer le programme. [1]

1. S'imaginer deux individus essayant de se croiser dans un couloir, entamant simultanément une manœuvre d'évitement du même côté, mettant les deux personnes a nouveau l'une face à l'autre, provoquant une nouvelle manœuvre d'évitement, et ainsi de suite...

# Dangers de la synchronisation

## Exemple de *dead lock*



```
class SynchronizedObject {
    public synchronized void use() { }

    public synchronized void useWith(SynchronizedObject other) {
        for (int i = 0; i < 1000; i++); // on simule un long travail
        System.out.println(Thread.currentThread() + " claims monitor on " + this);
        other.use(); // ça, ça sent mauvais...
    }
}

public class DeadLock extends Thread {
    private final SynchronizedObject obj1, obj2;

    private DeadLock(SynchronizedObject obj1, SynchronizedObject obj2) {
        this.obj1 = obj1; this.obj2 = obj2;
    }

    @Override public void run() {
        obj1.useWith(obj2);
        System.out.println(Thread.currentThread() + " is done.");
    }

    public static void main(String args[]) {
        SynchronizedObject o1 = new SynchronizedObject(), o2 = new SynchronizedObject();
        // dead lock, sauf si le 1er thread arrive à terminer avant que le 2e ne commence
        new DeadLock(o1, o2).start(); new DeadLock(o2, o1).start();
    }
}
```

# Éviter les dead-locks

## À l'aide d'une utilisation raisonnée des verrous



- Principe pour éviter les *dead-locks* : **toujours acquérir les verrous dans le même ordre** [1] et les libérer dans l'ordre inverse [2] (ordre LIFO, donc).
- En effet : dans l'exemple précédent, une exécution de `run` veut acquérir `o1` puis `o2`, alors que l'autre exécution veut faire dans l'autre sens.
- $\rightarrow$ quand on écrit un programme concurrent à l'aide de verrous explicites, il faut documenter un ordre unique pour prendre les verrous.

L'autre voie est de se reposer sur des abstractions de plus haut niveau, sur lesquelles il est plus aisé de raisonner (cf. la suite).

1. Pas évident en pratique : verrous créés dynamiquement, difficile de savoir quels verrous existeront à l'exécution. On peut aussi ne pas savoir quels verrous une méthode donnée d'une classe tierce utilise.
2. Pour les verrous intrinsèques, ordre inverse imposé par l'imbrication des blocs `synchronized`. Mais rien de tel pour les verrous explicites. La preuve de l'absence de *dead-lock* doit alors se faire au cas par cas.

# Paradigme des *threads*

## Est-ce la réalité ?





- Rappel : *thread* = abstraction
  - simulant la séquentialité dans le *thread* ;
  - permettant une communication instantanée inter-*thread* via une mémoire partagée ;
  - (et simulant le parallélisme parfait [1] entre *threads*).
- Est-ce vraiment la réalité ?

  → Divulgâchage : NON !

En réalité, paradigme idéal trop contraignant, empêchant les optimisations matérielles.

Modèle d'exécution réellement implémenté par la JVM : le **JMM** [2].

Seule garantie : sous condition, ce qu'on observe est indistinguable du paradigme idéal.

1. Hors synchronisation, évidemment.
2. Java Memory Model

# Modèle de mémoire Java et optimisations

## Cohérence de la mémoire



**Réalité physique :** chaque cœur de CPU dispose de son propre cache [1] de mémoire.

**Interprétation :** Chaque *thread* utilise potentiellement un cache de mémoire différent. Ainsi, les données partagées existent en de multiples copies pas forcément à jour.

(on parle de problèmes de visibilité des changements et de cohérence de la mémoire)

**Solution naïve :** répercuter immédiatement les changements dans tous les caches immédiatement.

**Problème :** cette opération est coûteuse et supprime le bénéfice du cache.

→ **le JMM :** ne garantit donc pas une cohérence parfaite (mais un minimum quand-même... )

1. Mémoire locale propre au cœur, plus proche physiquement et plus rapide que la mémoire centrale.

# Modèle de mémoire Java et optimisations

## Consistence de la mémoire



Par exemple, dans le programme suivant :

```
public class ThreadConsistence extends Thread{
    static boolean x = false, y = false;

    public void run(){
        if (x || !y) { x = true; y = true; } else System.out.println("Bug !");
        // Affiche "Bug !" si on trouve y vrai alors que x est faux
    }

    public static void test(int nthreads) throws InterruptedException {
        for (int i = 0; i < nthreads; i++) new ThreadConsistence().start();
    }
}
```

L'appel `test(100)` peut afficher « `Bug !` ».[1]

Par exemple : si un des *threads* finit d'exécuter « `x = true; y = true;` » et un autre reçoit, dans son cache, la modification sur `y` mais pas sur `x`.

1. Le JMM autorise cette possibilité théorique, mais probablement vous ne verrez jamais ce message !

# Modèle de mémoire Java et optimisations

## Réarrangement des instructions



**Réalité physique :** Les CPU sont dotés de mécanismes permettant de réordonner des instructions [1] qu'il sait devoir exécuter (afin de mieux occuper tous ses composants).

**Interprétation :** l'ordre du programme n'est pas toujours respecté (même sur 1 *thread*).

En plus, optimisations différentes d'une architecture matérielle à une autre (p. ex : comportement différent entre x86 et ARM) → ordre peu prévisible.

**Solution naïve :** ajouter des barrières [2] partout dans le code compilé.

**Problème :** vitesse d'exécution sous-optimale (le CPU n'arrive plus à donner autant de travail à tous ses composants).

→ **le JMM :** ne garantit pas le respect exact de l'ordre du programme... mais promet que certaines choses importantes restent bien ordonnées.

1. *out-of-order execution*
2. Insruction spécifique prévue dans les CPU, justement pour empêcher le ré-ordonnancement.

# Modèle de mémoire Java et optimisations

## Réarrangement des instructions



**Exemple :**

- Supposons qu'initialement `x == 0 && y == 0`. On veut exécuter :
  - sur le *thread* 1 : (1) `a = x;` (2) `y = 1;`
  - et, sur le *thread* 2 : (3) `b = y;` (4) `x = 2;`.
- (1) arrive-avant (2) et (3) avant (4)
  $\rightarrow$ à la fin, il semble impossible d'avoir à la fois `a == 2` et `b == 1`.
- Or c'est pourtant possible !
  En effet, sur chaque *thread* isolé, inverser les 2 instructions ne change pas le résultat. Comme il n'y a pas de synchronisation, rien n'interdit donc ces inversions. Il est donc possible d'exécuter les 4 instructions dans l'ordre suivant :
  `y = 1; x = 2; a = x; b = y;`.

# Modèle de mémoire Java et optimisations

## Principe informel du JMM



Entre 2 points de synchronisation, toute optimisation est autorisée tant qu'elle ne change pas le comportement observable d'un *thread* qui s'exécuterait sans interférence d'un autre *thread*.

Donc

- *mono-thread* $\rightarrow$ aucune différence visible due à ces optimisations ;
- mais *multi-thread* $\rightarrow$ différences possibles si synchronisation insuffisante.

$\rightarrow$ au programmeur de faire en sorte d'avoir une synchronisation suffisante afin que ces optimisations ne soient pas un problème.

**Remarque :** il reste à définir précisément ce qu'on entend par interférence et synchronisation suffisante.

# Ordre arrivé-avant et ordre d'exécution

## Définitions : 2 façons d'ordonner les évènements d'une exécution donnée [1]

- **Ordre arrivé-avant** :
  - ordre partiel sur les évènements d'une exécution, indiquant leur relation de causalité (toute modification causée par ce qui arrive-avant est « vue » par ce qui arrive-après).
  - Il est induit par :
    - l'ordre d'exécution des instructions sur un même *thread* tel que demandé par la logique du programme (**ordre du programme**) ;
    - les synchronisations (le réveil d'un *thread* arrive-après l'événement qui l'a réveillé) ;
    - et la causalité entre la lecture d'une variable `volatile` ou `final` et la dernière écriture de celle-ci avant cette lecture.
- **Ordre d'exécution :** ordre chronologique réel d'exécution des instructions.
  Dans une exécution correcte, on voudrait que cet ordre respecte « notre » logique.

1. Par opposition aux instructions d'un programme donné.

# Ordre arrivé-avant et ordre d'exécution

## Relation entre les 2 ordres



Pour une exécution donnée :

- **Ordre d'exécution =** réalité objective, non interprétée, de celle-ci.
  Celui-ci est difficile à prévoir, dépendant des optimisations opérées par le CPU.
  Il ne respecte pas forcément l'ordre du programme, et donc *a fortiori*, pas non plus l'ordre arrivé-avant d'une exécution donnée.
  De très nombreux ordres d'exécution sont possibles pour un même programme.
- **Ordre arrivé-avant =** interprétation idéale de la réalité (qui considère la logique du programme et des synchronisations).
  Il est défini sans ambiguïté et facile à déduire à partir d'un code source et d'une trace d'exécution (p. ex. : depuis un ordre d'exécution).

On prouvera qu'un programme est correct en raisonnant sur les ordres arrivé-avant de certaines exécutions particulières : les **exécutions séquentiellement cohérentes** [1].

1. À suivre !

# Accès en compétition

**Variable partagée :** variable accessible par plusieurs *threads*.

Tout attribut est (à moins de prouver le contraire) une variable partagée. Les autres variables (locales ou paramètres de méthodes) ne sont jamais partagées. [1]

**Accès conflictuels :** dans une exécution, 2 accès à une même variable sont conflictuels si au moins l'un des deux est en écriture.

**Accès en compétition** (*data race*) **:** 2 accès conflictuels à une variable partagée, tels que l'un n'arrive-pas-avant l'autre [2].

1. Mais les attributs de l'objets référencé peuvent être partagés !
2. C'est-à-dire : 2 accès qui ne sont pas reliés par une chaîne de synchronisations et d'ordres imposés par l'ordre des instructions du programme.

# Accès en compétition

## Exemples



### Programme avec accès en compétition :

```
class Boite {
    int x;
}

public class Competition {
    public static void main(String args[]) {
        Boite b = new Boite();
        new Thread(() -> { b.x = 1; }).start();
        new Thread(() -> {
                System.out.println(b.x);
            }).start();
    }

}
```

Ici, rien n'impose que la lecture de `b.x` arrive-avant son affectation ou bien le contraire.

### Programme sans accès en compétition :

```
class BoiteSynchro {
    private int x;
    public synchronized int getX() {
        return x;
    }
    public synchronized void setX(int x) {
        this.x = x;
    }
}

public class PasCompetition {
    public static void main(String args[]) {
        BoiteSyncro b = new BoiteSynchro();
        new Thread(() -> {
                b.setX(1);
            }).start();
        new Thread(() -> {
                System.out.println(b.getX());
            }).start();
    }

}
```

# Programme correctement synchronisé

**Exécution séquentiellement cohérente :** exécution

- qui suit un ordre total,
- respectant l'ordre du programme,
- et telle que pour toute lecture d'un emplacement mémoire, la dernière écriture, dans le passé, sur cet emplacement est prise en compte.

⇒ *Une exécution séquentiellement cohérente est donc une exécution idéale, intuitive, du programme, non affectée par les réordonnancements et incohérences de cache.*

Exemple : si `x = 0`, `x = 1` et `println(x)` s'exécutent dans cet ordre et qu'entre `x = 1` et `println(x)` il n'y a pas d'affectation à `x`, alors c'est bien `1` qui s'affiche.

**Programme correctement synchronisé :** se dit d'un programme dont toute exécution sequentiellement cohérente est sans accès en compétition.

# Modèle d'exécution

## La vérité, enfin !

L'ordre d'exécution exact d'un programme est imprévisible, mais ce qui suit est garanti :

> Si le programme est correctement synchronisé [1],
> alors son exécution est indiscernable d'une exécution séquentiellement cohérente.

**Concrètement**, pour que les optimisations ne provoquent pas d'incohérences, il « suffit » donc qu'il n'y ait **pas de compétition**.

Remarque : le plus dur reste de trouver quels accès sont en compétition...

Heureusement : d'après la propriété ci-dessus, **il suffit de vérifier exécutions « normales »** seulement pour prouver qu'aucune exécution ne se comporte de façon visiblement anormale.

1. Note : si la synchronisation est incorrecte, cela ne veut pas dire qu'on ne sait rien. En fait, la spécification donne tout un ensemble de règles basées sur un critère de causalité... règles trop compliquées et donnant des garanties trop faibles pour être raisonnablement utilisables en pratique.

# Éviter les problèmes, version courte

**Si vous pouvez prouver que tout accès en compétition à vos variables partagées est impossible dans une exécution « normale »[a], alors vous serez pas importuné par les optimisations !**

*a*. « Normale » = séquentiellement cohérente, non optimisée.

# Supprimer les compétitions

Comment éviter les compétitions ?

- éviter de partager les variables quand ce n'est pas nécessaire → préférer les variables locales (jamais partagées) aux attributs ;
- quand ça suffit, privilégier les données partagées en lecture seule → privilégier les structures immuables (voir ci-après) ;
- sinon, renforcer la relation arrivé-avant :
  - utiliser les mécanismes de synchronisation déjà présentés,
  - marquer des attributs comme `final` ou `volatile` (voir ci-après) ;
- utiliser des classes déjà écrites et garanties « *thread-safe* ».
- Souvent, rien de tout ça ne convient : on peut avoir besoin d'attributs modifiables sans synchronisation ! Mais il faudra s'assurer qu'un seul *thread* peut y accéder.

# Les mot-clés **volatile** et **final** (1)

**Attributs volatils :** un attribut déclaré avec **`volatile`** garantit [1] :

- que tout accès en lecture se produisant, chronologiquement, après un accès en écriture, arrive-après celui-ci.
  (concrètement : cet attribut n'est jamais mis dans le cache local d'un *thread*)
- que tout accès simple en lecture ou écriture est atomique (même pour **`long`** et **`double`**).

→ comme si cet attribut était accédé via des accesseurs **`synchronized`**.

**Attributs finaux :**

- déjà vu : un attribut **`final`** ne peut être affecté qu'une seule fois (lors de l'initialisation de la classe ou de l'objet).
- garantie supplémentaire : comme pour **`volatile`**, tout accès en lecture à un attribut **`final`** arrive-après son initialisation (unique, pour **`final`**).

1. Ceci ne concerne pas le contenu de l'éventuel objet référencé.

# Les mot-clés **volatile** et **final** (2)

**Technique infaillible :** tous les attributs `volatile` (ou `final`) $\Longrightarrow$ accès en compétition impossible. Cependant pas idéale car :

- **non réaliste :** un programme utilise des classes faites par d'autres personnes ; [1]
- **non efficace :** `volatile` empêche les optimisations $\Rightarrow$ exécution plus lente. [2]

En plus, `volatile` ne permet pas de rendre les méthodes atomiques $\Rightarrow$ entrelacements toujours non maîtrisés $\Rightarrow$ `volatile` ne suffit pas toujours pour tout.

**Exemple :** avec `volatile int x = 0;`, si on exécute 2 fois en parallèle `x++`, on peut toujours obtenir 1 au lieu de 2.

1. Mais on peut encapsuler leurs instances dans des classes à méthodes synchronisées... au prix d'encore un peu moins de performance.
2. Remarque : pour `final`, la question ne se pose qu'au début de la vie de l'objet. À ce stade, accéder à une ancienne version de l'attribut n'aurait aucun sens. L'optimisation serait nécessairement fausse.

# Les objets immuables

- **Rappel : immuable** = non modifiable. Le terme s'applique aux objets et, par extension, aux classes dont les instances sont des objets immuables.
- Ces objets ont généralement des champs tous `final`. Conséquence : relation « arrivé-avant » entre l'initialisation de l'objet et tout accès ultérieur.
- Pendant la vie de l'objet : pas d'accès en écriture ⇒ pas d'accès en compétition.

⇒ non seulement l'utilisation qui en est faite dans un *thread* n'influe pas sur l'utilisation dans un autre *thread* [1], mais en plus il ne peut pas y avoir d'incohérence de cache par rapport au contenu d'un objet immuable. [2]

Remarque : tout cela reste vrai quand on parle des champs `final` d'un objet quelconque.

1. Donc tout objet immuable est *thread-safe*.
2. Si l'objet immuable est correctement publié, tous les *threads* sont d'accord sur l'ensemble des valeurs publiées.

# Les objets immuables

## Faire des calculs en utilisant les objets immuables



- Typiquement, une étape de calcul consiste à créer un nouvel objet immuable à partir d'objets immuables existants (puisqu'on ne peut pas les modifier).
- Un tel calcul peut être réalisé à l'aide d'une **fonction pure** [1]
- **Inconvénient :** implique d'allouer un nouvel objet pour chaque étape de calcul. (coûteux, mais pas forcément excessif [2])
- Le résultat doit être correctement publié pour être utilisable par un autre *thread* :
  - grâce aux mécanismes (méthodes) de passage de message prévues par l'API utilisée,
  - ou bien « à la main », en l'enregistrant dans une variable partagée (soit `volatile`, soit `private` avec accesseurs `synchronized`) modifiable.
    **Exemple :** `SharedRessources.setX(f(SharedResources.getX()));` (où `getX` et `setX` sont `synchronized` et `f` est une fonction pure).

1. Fonction sans effet de bord, notamment sans modification d'état persistente.
2. Notamment si l'*escape analysis* détermine que l'objet n'est que d'usage local → la JVM l'alloue en pile. Cela dit, ceci ne concerne que les calculs intermédiaires car la variable partagée est stockée dans le tas.

# Variables atomiques

`java.util.concurrent.atomic` propose un certain nombre de classes de **variables atomiques** (classes mutables *thread-safe*).

- Exemples : `AtomicBoolean`, `AtomicInteger`, `AtomicIntegerArray`, …
- Leurs instances représentent des booléens, des entiers, des tableaux d'entiers, ...
- Accès simples : comportement similaire aux variables volatiles.
- Disposent, en plus, d'opérations plus complexes et malgré tout atomiques (typiquement : incrémentation). [1]

1. L'accès atomique est garanti sans synchronisation, grâce à des appels à des instructions dédiées des processeurs, telles que CAS (compare-and-set). Ainsi ces classes ne sont en réalité pas implémentées en Java, car elles sont compilées en tant que code spécifique à l'architecture physique (celle sur laquelle tourne la JVM).

# Autres objets *thread-safe*

Nombre de classes de l'API sont signalées comme *thread-safe*. En particulier, il peut être utile de rechercher la documentation des collections concurrentes (*package* `java.util.concurrent`).

Regardez les différentes implémentations de `BlockingQueue` et de `ConcurrentMap`, par exemple.

# Les nombreux inconvénients de l'API *threads*

Utiliser directement les *threads* et les moniteurs → nombreux inconvénients :

- Chaque *thread* utilise beaucoup de mémoire. Et les instancier prend du temps.
- Trop de *threads* → changements de contexte fréquents (opération coûteuse).
- Nécessité de communiquer par variables partagées → risque d'accès en compétition (et donc d'incohérences)
- En cas de synchronisation mal faite, risque de blocage.

**Heureusement :** de nombreuses API de haut niveau [1] aident à contourner ces écueuils.

→ on travaillera plutôt avec celles-ci que directement avec les *threads* et les moniteurs.

1. programmées par dessus les *threads* et les moniteurs

# APIs pour la concurrence

## Schéma de base, pour limiter le nombre de *threads*

**Idée :** réutiliser un même *thread* pour plusieurs exécuter plusieurs tâches tour à tour.

**Exécuteur :** objet qui gère un certain ensemble de *threads*

- en distribuant des **tâches** sur ceux-ci, selon politique définie ;
- en évitant de créer plus de *threads* que nécessaire [1] ;
- et en évitant de détruire un *thread* aussitôt qu'il est libre (pour éviter d'en re-créer).

**Tâche :**

- séquence d'instructions (en pratique : une fonction) à exécuter sur un *thread*
- ne peut pas être mise en pause pour libérer le *thread* au profit d'une autre. [2] [3]

1. Selon politique de l'exécuteur. Plusieurs possibles. Par exemple : nb. max. *threads* $\leq$ nb. cœurs.
2. Le *thread*, lui, peut être mis en pause par le noyau pour libérer un processeur au profit d'un autre *thread*.
3. En principe, car avec `ForkJoinPool`, notamment, certains appels de méthodes permettent la mise en pause $\rightarrow$ multi-tâche coopératif.

# APIs pour la concurrence

## Différents styles d'APIs pour les tâches



Pour synchroniser et faire communiquer des tâches interdépendantes, 2 styles d'API :

(dans les 2 cas, **passage de messages** plutôt que variables partagées)

- **API bloquante :** appel de méthode bloquante pour attendre la fin d'une autre tâche (comme `join` pour les *threads*) et obtenir son résultat (si applicable).

  **Exemple :**

```
ForkJoinTask<Integer> f = ForkJoinTask.adapt(() -> scanner.nextInt());
ForkJoinTask.adapt(() -> {
    f.fork(); // lancement d'une sous-tâche
    System.out.println(f.join()); // appel bloquant avec récupération du résultat
}).fork(); // lancement de la tâche principale
```

  Dans le JDK : `Thread`, `Future` et `ForkJoinTask` suivent ce principe. [1]

1. Hors JDK, citons le principe des « fibres  dans la bibliothèque Quasar (qui implémente aussi les « acteurs » en API bloquante).

# APIs pour la concurrence

## Différents styles d'APIs pour les tâches

- **API non bloquante :** une tâche qui dépend d'un résultat fourni par une autre est passée en tant que **fonction de rappel** (*callback*) [1]. Cette dernière sera déclenchée par l'arrivée du résultat attendu (plus généralement : un évènement). **Exemple :**

```
CompletableFuture
// tâche 1 : d'abord programme la lecture d'un Scanner
  .supplyAsync(scanner::nextInt)
// tâche 2 : dès qu'un entier est fourni, affiche-le
  .thenAccept(System.out::println);
```

Dans le JDK : Swing, `CompletableFuture`, `Stream`, `Flow`. [2]

1. Sur le principe, une fonction de première classe → en Java traditionnel un objet avec une méthode dédiée ; en Java moderne, une lambda-expression.
2. Hors JDK : Akka (implémentation des « acteurs »), diverses implémentations de la spécification *Reactive Streams* (autres que `Flow`), JavaFX (en effet, JavaFX n'est plus dans le JDK depuis Java 11), ...

# Interfaces `Executor` et `Runnable`

- En Java, les exécuteurs sont les instances de l'interface `Executor` :

```
public interface Executor { void execute(Runnable command); }
```

L'appel `unExecutor.execute(unRunnable)` exécute la méthode `run()` de `unRunnable`. Ainsi, dans ce cas, les tâches sont des instances de `Runnable`.

- **Un exemple :** (`ExecutorService` étend `Executor`)

```
// instanciation d'un exécuteur gérant un thread unique :
ExecutorService executor = Executors.newSingleThreadExecutor();

// lancement d'une tâche (décrite par la lambda expression, type inféré Runnable)
executor.execute(() -> { System.out.println("bla bla bla"); });

// on détruit les threads dès que tout est terminé
executor.shutdown();
```

Implémentations diverses, utilisant un ou plusieurs *threads* (***thread pool***).

# Interfaces `ExecutorService`, `Callable<V>` et `Future<V>`

- `ExecutorService` : étend `Executor` en y ajoutant :
  - `<T> Future<T> submit(Callable<T> task)` : programme une tâche.
  - Des méthodes pour demander et/ou attendre la terminaison des tâches en cours.
- `Callable<V>` est comme `Runnable`, mais sa méthode retourne un résultat :

```
public interface Callable<V> { V call(); }
```

- `Future<V>` = objets pour accéder à un résultat de type `V` promis dans le futur :

```
public interface Future<V> {
  boolean cancel(boolean mayInterruptIfRunning);
  V get() throws InterruptedException, ExecutionException;
  V get(long timeout, TimeUnit unit) throws InterruptedException,
      ExecutionException, TimeoutException;
  boolean isCancelled();
  boolean isDone();
}
```

Les méthodes `get` sont bloquantes jusqu'à disponibilité du résultat.

# Interfaces ExecutorService, Callable<V> et Future<V>

## Usage (schéma général)



```
ExecutorService es = ...;
...
Callable<String> task = ...;
// task sera executé dans thread choisi par es :
Future<String> result = es.submit(task);
...
// le programme attend puis affiche le résultat :
System.out.println("Résultat : " + result.get());
```

# Interfaces ExecutorService, Callable<V> et Future<V>

## Exemple plus complet



```java
public class TestCall implements Callable<Integer> {
    private final int x;
    public TestCall(int x) { this.x = x; }
    @Override public Integer call() { return x; }
}

public class Exemple {
    public static void main(String[] args){
        ExecutorService executor = Executors.newSingleThreadExecutor();
        Future<String> futur1 = executor.submit(new TestCall(1));
        Future<String> futur2 = executor.submit(new TestCall(2));
        try { System.out.println(futur1.get() + futur2.get()); } // affiche "3"
        catch(ExcecutionException | InterruptedException e) { ... }
        finally { executor.shutdown(); }
    }
}
```

Ici, l'exécuteur exécute les 2 tâches l'une après l'autre (un seul *thread* utilisé), mais en même temps que la méthode `main()` continue à s'exécuter.

Cette dernière finit par attendre les résultats des 2 tâches pour les additionner.

# Instancier des exécuteurs

**À l'aide de la classe `Executors` :**

- = bibliothèque de fabriques statiques d'`ExecutorService`.
- `static ExecutorService newSingleThreadExecutor()` : crée un `ExecutorService` utilisant un *worker thread* unique.
- `static ExecutorService newCachedThreadPool()` : crée un exécuteur dont les *threads* du *pool* se créent, à la demande, sans limite, mais sont réutilisés s'ils redeviennent disponibles.
- `static ExecutorService newFixedThreadPool(int nThreads)` : même chose, mais avec limite fixée à `n` *threads* [1].

On peut aussi utiliser directement les constructeurs de `ThreadPoolExecutor` ou de `ScheduledThreadPoolExecutor` [2] (nombreuses options).

1. Choisir `n` en rapport avec le nombre d'unités de calcul/cœurs.
2. Implémente `ScheduledExecutorService`, permettant de programmer des tâches périodiques et/ou différées. Les futurs retournés implémentent `ScheduledFuture<V>`. Regardez la documentation.

# Les limites de `ThreadPoolExecutor`

(et des fabriques `newCachedThreadPool()` et `newFixedThreadPool()`)



- But d'un *thread pool* = réduire le nombre de *threads* → petit nombre de *threads*.
- Or les *threads* bloqués (par appel bloquant, comme `f.get()`, au sein de la tâche) ne sont pas réattribuables à une autre tâche [1].
  → moins de *threads* disponibles dans le *pool* → ralentissement.
- **Cas extrême :** si grand nombre de tâches concurrentes avec interdépendances, il arrive que tout le *pool* soit bloqué par des tâches en attente de tâches bloquées ou non démarrées → rien ne viendra débloquer la situation.
  Cette situation s'appelle un ***thread starvation deadlock*** [2].

**Comment concilier *pool* de taille bornée et garantie d'absence de bloquage ?**

1. Il est impossible de « sortir » une tâche déjà en exécution sur un *thread* pour le libérer.
2. Du point de vue des *threads*, c'est bien un *deadlock* : dépendance cyclique entre *threads*.
Du point de vue des tâches, dépendance pas forcément cyclique, mais bloquage car multi-tâche non-préemptif s'exécutant sur un nombre limité d'unités d'exécution.

# *Work-stealing strategy*

## Decription

**Solution :** stratégie de vol de travail (***work stealing***). Principe :

- une file d'attente de tâches par *thread* au lieu d'une commune à tout le *pool* ;
- tâches générées par une autre tâche $\rightarrow$ ajoutées sur file du même *thread* ;
- quand un *thread* veut du travail, il prend une tâche en priorité dans sa file, sinon il en « vole » une dans celle d'un autre *thread* ;
- **le plus important :** si le résultat attendu n'est pas disponible, `get` (et `join`) exécute d'abord les tâches en file au lieu de bloquer le *thread* tout de suite.

  $\Rightarrow$ C'est là que se met en place la coopération entre tâches.

Le vol de travail assure que si tous les *threads* sont bloqués (par des tâches en attente d'une autre tâche), c'est qu'il n'y a plus de tâche à démarrer.

Cela veut dire que les tâches attendues sont déjà démarrées et non terminées. C'est donc qu'elles sont elles-mêmes en attente d'une tâche... aussi en attente.

$\Longrightarrow$ la seule possibilité c'est que les tâches s'attendant les unes les autres forment un (ou des) cycle de dépendances.

$\Longrightarrow$ si pas de dépendances cycliques, *thread starvation deadlock* impossible.

# *Work-stealing strategy*

## Indications et contre-indications

- Petites tâches à dépendances acycliques (notamment, algorithmes récursifs)
  → stratégie très intéressante (pas de *thread starvation deadlock*).
- Tâches sans dépendances
  → stratégie inutile et plus lourde que la stratégie « naïve »[1].
- Tâches à dépendances cycliques
  → *deadlocks* assurés (mais aucune stratégie ne peut fonctionner dans ce cas !).

1. Sans compter qu'en Java, l'implémentation de celle-ci (`ThreadPoolExecutor`) est plus configurable que l'implémentation de la *work-stealing strategy* (`ForkJoinPool`).

# ForkJoinPool et ForkJoinTask

## L'implémentation de la *work-stealing strategy* en Java



Ainsi Java propose :

- la classe `ForkJoinPool` : implémentation d'`Executor` utilisant cette stratégie
- la classe `ForkJoinTask` : tâches capables de générer des sous-tâches (« `fork()` ») et d'attendre leurs résultats pour les utiliser (« `join()` »).

`ForkJoinPool` est considéré suffisament efficace pour que le *thread pool* par défaut (utilisé implicitement par plusieurs API concurrentes) soit une instance de cette classe.

Obtenir le *thread pool* par défaut : `ForkJoinPool.commonPool()`.

# ForkJoinPool et ForkJoinTask

- Méthodes de `ForkJoinTask` :
  - `ForkJoinTask<T> fork()` : demande l'exécution de `t` et rend la main. Le résultat du calcul peut être récupéré plus tard en interrogeant l'objet retourné.
  - `T invoke()` : pareil, mais attend que `t` soit finie et retourne le résultat.
  - `T join()` : attendre le résultat du calcul signifié par `t` (`T get()` existe aussi car `ForkJoinTask<T>` implémente `Future<T>`)

  `fork` et `invoke` exécutent la tâche dans le *pool* dans lequel elles sont appelées, si applicable, sinon dans le *pool* par défaut.
- Pour exécuter une tâche sur un `ForkJoinPool` précis, on appelle les méthodes suivantes (de la classe `ForkJoinPool`) sur le *pool* `p` cible :
  - `<T> T invoke(ForkJoinTask<T> task)` : demande l'exécution de `task` sur `p` et retourne le résultat dès qu'elle se termine (appel bloquant).
  - `<T> ForkJoinTask<T> submit(ForkJoinTask<T> task)` : idem, mais rend la main tout de suite en retournant un futur, permettant de récupérer le résultat plus tard.
  - `void execute(ForkJoinTask<?> task)` : idem, aussi non bloquant, mais on ne récupère pas le résultat.

# ForkJoinPool et ForkJoinTask

## Et ForkJoinTask ?

- Classe abstraite. Ses objets sont les tâches exécutables par `ForkJoinPool`.
- On préfère étendre une de ses sous classes (abstraites aussi) [1] :
  - `RecursiveAction` : tâche sans résultat (exemple : modifier les feuilles d'un arbre)
  - `RecursiveTask<V>` : tâche calculant un résultat de type `V` (exemple : compter les feuilles d'un abre)

  Dans les 2 cas, on étend la classe en définissant la méthode `compute()` qui décrit les actions effectuées par la tâche :
  - pour `RecursiveAction` : `protected void compute();`
  - pour `RecursiveTask<V>` : `protected V compute();`
- 3 fabriques statiques `ForkJoinTask.adapt(...)` [2] permettent de créer des tâches à partir de `Runnable` et de `Callable<V>`.

```
ForkJoinTask.adapt(() -> { /* instructions */ }).fork() // sympa avec les lambdas !
```

1. Ces deux classes servent juste à faciliter l'implémentation.
2. Le nom `adapt` provient du patron *adapter*.

# ForkJoinPool et ForkJoinTask

## Et car rien ne vaut un exemple...



### La tâche récursive :

```
class Fibonacci extends RecursiveTask<Integer> {
    final int n;

    Fibonacci(int n) { this.n = n; }

    @Override protected Integer compute() {
        if (n <= 1) return n;
        Fibonacci f1 = new Fibonacci(n - 1);
        f1.fork();
        Fibonacci f2 = new Fibonacci(n - 2);
        return f2.compute() + f1.join();
    }
}
```

### Et l'appel initial (dans `main()`) :

```
System.out.println((new ForkJoinPool()).invoke(new Fibonacci(12)));
```

# Une alternative : `CompletableFuture`

- ```
  class CompletableFuture<T> implements Future<T>, CompletionStage<T>
  ```
- `CompletionStage<T>` propose des méthodes pour ajouter des *callbacks* à exécuter lors de la complétion de la tâche.

$\rightarrow$ changement de paradigme : plus d'appels bloquants dans les tâches élémentaires, mais dépendances maintenant décrites en dehors de celles-ci.

Le programme appelant compose ces tâches entre elles pour décrire une « recette » [1] qu'il soumet au *thread pool* (et donc n'effectue pas non plus d'appel bloquant).

1. Les savants parlent de « monade ».
  ```

# Une alternative : CompletableFuture

## Exemple : de ForkJoinTask à CompletableFuture



```
Rf rf = Boulot.f();
Future<Rg> frg = ForkJoinTask.adapt(() -> Boulot.g(rf)).fork();
Rh rh = Boulot.h(rf);
Ri ri = Boulot.i(rh, frg.join());
System.out.println("Résultat : " +ri);
```

Devient :

```
CompletableFuture<Rf> cff = CompletableFuture.supplyAsync(Boulot::f);
CompletableFuture<Rg> cfg = cff.thenApplyAsync(Boulot::g);
CompletableFuture<Rh> cfh = cff.thenApply(Boulot::h);
CompletableFuture<Ri> cfi = cfg.thenCombine(cfh, Boulot::i);
cfi.thenAccept(ri -> { System.out.println("Résultat :" + ri); });
```

# Les *reactive streams* [2] avec `Flow`

`java.util.concurrent.Flow` :

- Implémentation standardisée dans le JDK (Java 9) de la spécification *reactive streams* (2015), issue d'une initiative des sociétés Netflix, Pivotal et Lightbend.
- C'est une API non bloquante, inspirée du patron Observateur/Observé.
- **Idée :** 2 sortes d'objets, `Publisher` et `Subscriber`. Le premier peut produire une séquence de messages, alors que le second réagit aux données qu'on lui envoie.
- Pour cela, le `Subscriber` doit d'abord s'abonner à un `Publisher` (et un seul).
- En cas d'utilisation « normale » [1], le `Subscriber` traite les messages reçus séquentiellement (pas de synchronisation à gérer dans ses méthodes).

Plusieurs `Subscriber` peuvent s'abonner au même `Publisher` (« *fan out* »).

1. Un seul abonnement, et `Publisher` correctement implémenté.
Cela dit, il est possible de coordonner plusieurs abonnements (« *fan in* »), mais il faut le faire « à la main » à l'aide de plusieurs instances de `Subscriber` et un peu de synchronisation.
2. C'est le nom de la spécification, cela n'a pas de rapport avec l'API *stream* de Java.

# Les *reactive streams* avec `Flow`

## Un exemple simple (programme)



### On définit un `Subscriber` :

```
public class PrintSubscriber implements Flow.Subscriber<String> {
  private Flow.Subscription subscription;
  @Override public void onSubscribe(Flow.Subscription subscription) {
      this.subscription = subscription;
      subscription.request(1); // Je suis prêt à recevoir 1 premier message !
      System.out.println(this + " : Je suis inscrit !");
  }
  @Override public void onNext(String item) {
      subscription.request(1); // Je suis prêt à recevoir 1 autre message !
      System.out.println(this + " : " + item + " (thread " + Thread.currentThread().getName() + ")");
  }
  @Override public void onError(Throwable throwable) { System.out.println(this + " : Oups ?"); }
  @Override public void onComplete() { System.out.println(this + " : C'est fini !"); }
}
```

### On connecte les morceaux et on lance :

```
  public static void main(String[] args) throws InterruptedException {
      try (var publisher = new SubmissionPublisher<String>()) { // SubmissionPublisher fourni par JDK
          publisher.subscribe(new PrintSubscriber()); // on abonne une instance
          publisher.subscribe(new PrintSubscriber()); // puis une autre
          List.of("Lorem", "ipsum", "dolor", "sit", "amet").forEach(publisher::submit);
          ForkJoinPool.commonPool().awaitTermination(1000, TimeUnit.MILLISECONDS);
      }
  }
```

# Les *reactive streams* avec `Flow`

## Un exemple simple (affichage)



On observe alors sur la sortie standard :

```
PrintSubscriber@284682b2: Je suis inscrit !
PrintSubscriber@1446b42: Je suis inscrit !
PrintSubscriber@1446b42: Lorem (thread ForkJoinPool.commonPool-worker-19)
PrintSubscriber@284682b2: Lorem (thread ForkJoinPool.commonPool-worker-5)
PrintSubscriber@1446b42: ipsum (thread ForkJoinPool.commonPool-worker-19)
PrintSubscriber@284682b2: ipsum (thread ForkJoinPool.commonPool-worker-5)
PrintSubscriber@1446b42: dolor (thread ForkJoinPool.commonPool-worker-19)
PrintSubscriber@284682b2: dolor (thread ForkJoinPool.commonPool-worker-5)
PrintSubscriber@1446b42: sit (thread ForkJoinPool.commonPool-worker-19)
PrintSubscriber@284682b2: sit (thread ForkJoinPool.commonPool-worker-5)
PrintSubscriber@1446b42: amet (thread ForkJoinPool.commonPool-worker-19)
PrintSubscriber@284682b2: amet (thread ForkJoinPool.commonPool-worker-5)
PrintSubscriber@284682b2: C'est fini !
PrintSubscriber@1446b42: C'est fini !
```

# Les *reactive streams* avec `Flow`

## Les interfaces



```java
public class Flow {
  private Flow() {} // non instanciable

  public static interface Publisher<T> {
    public void subscribe(Subscriber<? super T> subscriber);
  }

  public static interface Subscriber<T> {
    public void onSubscribe(Subscription subscription);
    public void onNext(T item);
    public void onError(Throwable throwable);
    public void onComplete();
  }

  public static interface Subscription {
    public void request(long n);
    public void cancel();
  }

  public static interface Processor<T,R> extends Subscriber<T>, Publisher<R> {  }

  static final int DEFAULT_BUFFER_SIZE = 256;

  public static int defaultBufferSize() { return DEFAULT_BUFFER_SIZE; }
}
```

# Les *reactive streams* avec `Flow`

## Explication



Quelques points restent à expliquer.

Les abonnements :

- Un abonnement est symbolisé par un objet intermédiaire, instance de `Flow.Subscription`.
- C'est le `Flow.Publisher` qui est chargé d'instancier l'abonnement et de le passer à l'abonné (via méthode `onSubscribe`).
- `Flow.Subscription` est généralement implémentée dans ou avec l'implémentation de `Flow.Publisher`.

La gestion de la « **contre-pression** » (ou ***backpressure***) :

- Un abonné ne reçoit pas plus de messages que le nombre qu'il a demandé à recevoir (via appel à `request(n)`).
- Ainsi le `Publisher` doit gérer une file d'attente (messages prêts à être publiés, mais qui n'ont pas encore été envoyés à tous les abonnés).

# Les *reactive streams* avec `Flow`

## En pratique



En pratique, le JDK fournit déjà une implémentation de `Publisher<T>` : `SubmissionPublisher<T>` (contenant une implémentation de `Subscription`).

Cette classe :

- utilse un *thread pool* (soit celui passé en paramètre, soit `ForkJoinPool.commonPool()`)
- a une méthode `public int submit(T item)` qui permet de fournir à ce *publisher* les prochains messages qu'il va diffuser à ses abonnés.
- quand un message doit être envoyé à un abonné, sa méthode `onNext` est appelée dans une tâche soumise au *thread pool* (donc possibilité de répartition sur plusieurs *threads*).

Ainsi, pour créer des graphes de `Flow` sans effort, on peut implémenter des `Publisher` et `Processor` basés sur cette classe (qui implémente déjà tout ce qui est un peu compliqué).

# Les *reactive streams* avec `Flow`

## Exemple de `Processor`



Pour décomposer un traitement en plusieurs étapes, on peut établir une chaîne : `Publisher` → `Processor` → ... → `Processor` → `Subscriber`.

Or `Processor` = `Subscriber` + `Publisher` et, malheureusement, `Publisher` n'a pas d'implémentation dans le JDK et elle est difficile à implémenter.

**Solution :** comme suggéré juste avant, réutiliser `SubmissionPublisher` :

```
public abstract class AbstractProcessor<T, U> implements Flow.Processor<T, U> {
  private final SubmissionPublisher<U> dispatcher; // préférons la composition à l'héritage !
  // constructeurs de même signature que ceux de SubmissionPublisher
  public AbstractProcessor() { this.dispatcher = new SubmissionPublisher<U>(); }
  public AbstractProcessor(Executor executor, int maxBufferCapacity) {
      this.dispatcher = new SubmissionPublisher<U>(executor, maxBufferCapacity);
  }
  public AbstractProcessor(Executor executor, int maxBufferCapacity, BiConsumer<? super
        Flow.Subscriber<? super U>, ? super Throwable> handler) {
      this.dispatcher = new SubmissionPublisher<U>(executor, maxBufferCapacity, handler);
  }
  // délégation des abonnements et de la soumission de nouveaux messages à dispatcher
  @Override public final void subscribe(Flow.Subscriber<? super U> subscriber) {
      dispatcher.subscribe(subscriber);
  }
  protected final int submit(U item) { return dispatcher.submit(item); }
}
```

# Les *reactive streams* avec `Flow`

## Exemple de `Processor`



Ensuite on peut étendre cette classe abstraite pour écrire un `Processor` complet :

```
public class DoublerProcessor extends AbstractProcessor<String, String> {
  Flow.Subscription subscription;
  @Override public void onSubscribe(Flow.Subscription subscription) {
      subscription.request(1);
      this.subscription = subscription;
  }
  @Override public void onNext(String item) {
      subscription.request(1); // requête de nouveau mot à chaque fois, mais on pourrait faire autrement :
      // par exemple, attendre qu'au moins un des abonnés en ait besoin (voire tous les abonnés)
      submit(item + item); // pour chaque mot "mot" reçu, transmet "motmot" à ses abonnés.
  }
  @Override public void onError(Throwable throwable) { }
  @Override public void onComplete() { }
}
```

Écrire directement `DoublerProcessor` sans écrire par `AbstractProcessor` était possible, mais cette dernière peut être réutilisée pour d'autres concrétisation.
En version très courte on aurait pu avoir :

```
public class DoublerProcessor extends SubmissionPublisher<String> implements Flow.Processor<String,
      String> { // héritons de SubmissionPublisher, pour montrer autre chose que la composition !
  // redéfinitions de onSubscribe, onNext, onError et onComplete comme ci-dessus
  ...
}
```

# Choisir une API concurrente

Dans ce cour, 4 API haut niveau concurrentes vous ont été présentées :

- les *streams* concurrents
- *fork/join*
- `CompletableFuture`
- `Flow`

Toutes les 4 permettent d'éviter l'usage de variables partagées (et la synchronisation explicite).

Si certains programmes peuvent être réalisés indifféramment avec plusieurs APIs, certaines sont clairement plus adaptées à certains cas d'usage...

Par ailleurs, certaines API correspondront mieux à votre façon de penser.

Pour vous faire une idée, **expérimentez !**

# Interfaces graphiques en Java

Pour des raison historiques, plusieurs bibliothèques sont utilisables (y compris dans le JDK) :

- AWT : existe depuis les premières versions de Java, se repose sur les composants graphiques “natifs” du système d’exploitation (rapide, mais apparence différente entre Windows, macOS, Linux, etc... )
- Swing : bibliothèque “officielle” de Java. Dépend peu des composants du système (donc apparence différente entre plateformes).
- SWT (et surcouche JFace) : bibliothèque du projet Eclipse. Se repose principalement sur les composants natifs (comme AWT) mais implémente tout ce que le système ne fournit pas.
- JavaFX : alternative plus moderne, similaire à Swing dans les principes. [1]

Dans ce cours : exemple de JavaFX, mais principes similaires pour les autres.

1. Intégrée un temps au JDK comme potentiel successeur de Swing (Java 8), puis finalement confiée au projet OpenJFX (Java 11).

# Interfaces graphiques en Java

## Avertissement



Attention, c'est un sujet très vaste, même en se limitant à JavaFX.

Ce cours ne fera qu'effleurer certains des sujets essentiels et donner quelques pointeurs pour mener à bien le projet.

Il conviendra d'avoir une démarche active pour combler d'éventuels besoins non couverts par le cours (consultez les pages de documentation de Java !!!).

# Principes généraux

- **Interface graphique :** hiérarchie de **composants** graphiques se contenant les uns les autres (ex : un bouton dans un panneau dans une fenêtre... )
- Quelques composants standard (fournis par l'API), mais en général on aime bien les personnaliser. (ex : l'API fournit la fenêtre de base, mais on peut définir une fenêtre "éditeur" qu'on va instancier à volonté)
- Les composants peuvent capter des **évènements** (validation, clic souris, entrée clavier, redimensionnement, ... ), dont le traitement est délégué à des fonctions de rappel (**gestionnaires d'évènements**) $\rightarrow$ **programmation évènementielle**

# Construire une GUI : mode d'emploi (1)

**Pour une fenêtre "statique" :**

1. Conceptualisez d'abord la fenêtre de votre application, dessin à l'appui.
2. Déterminez ses composants et leur hiérarchie (qui contient qui ?) sous forme d'arbre.
3. Programmez/écrivez [1] la description de la fenêtre, de ses composants et de leurs propriétés (taille, couleur, etc.) et des relations entre composants (notamment relations contentant/contenu).

À ce stade, votre programme peut afficher votre jolie fenêtre... qui ne fera rien [2]. Pour en faire une application utile, il faut maintenant associer des actions aux évènements.

1. En fonction du contexte, ça peut être un programme Java... ou bien un fichier dans un langage descriptif tel que HTML ou FXML.
2. On a en fait implémenté la partie "Vue" du patron MVC.

# Construire une GUI : mode d'emploi (2)

## Gérer les évènements



**Pour gérer les évènements** [1] **:**

1. On crée des **gestionnaires d'évènement**, spécifiques à chaque type d'évènement.
2. Pour chaque type d'évènement qu'on veut traiter sur un composant donné, on lui associe un gestionnaire, en le passant à une certaine méthode de ce composant. (Ceci peut se faire lors de l'initialisation du composant en question.)
3. Désormais, à chaque fois que cet évènement se produira, le gestionnaire sera exécuté avec pour paramètre un **descripteur d'évènement**.

Un gestionnaire d'évènement est une "fonction" [2] dont le paramètre est un **descripteur d'évènement** (de type souvent nommé `XXXEvent`), contenant la description des circonstances de l'évènement (composant d'origine, coordonnées, bouton cliqué ...).

1. Ceci correspond à la partie "Contrôleur" du patron MVC.
2. Fonction de rappel, matérialisée comme un objet contenant un méthode qui décrit la fonction ; c'est donc une fonction de première classe.

# JavaFX

- A existé tantôt comme bibiliothèque séparée, tantôt comme composant du JDK (Java 8 à 10). À partir de Java 11, développé au sein du projet séparé OpenJFX [1].
- Avant Java 8, JavaFX pouvait être programmé via un langage de script appelé JavaFX Script, celui-ci a été abandonné depuis.
- JavaFX : bien plus qu'une bibiliothèque de description d'IG.
  Par exemple, en plus des composants typiques, on peut insérer dans l'arbre des formes 3D, et appliquer au tout diverses transformations géométriques, y compris en 3D.

Une particularité de JavaFX, c'est la possiblitié de décrire une IG via un langage de description appelé **FXML** (inspiré de HTML), et de définir le style des composants via des pages de style **CSS**. [2]

1. Mais certaines distributions de Java incluent JavaFX : citons Zulu de Microsoft et Liberica de Bellsoft.
2. Ce cours n'explique pas la syntaxe de FXML et de CSS, mais seulement de la construction de l'IG via des méthodes purement Java. Cependant, vous pouvez les utiliser en TP ou en projet.

# Structure d'une IG en JavaFX

En JavaFX, nous avons la hiérarchie suivante :

- L'arbre est appelé **graphe de scène** (*scene graph*) et est constitué de **nœuds**, instances de sous-classes de `Node`.
- Les nœuds internes sont instances de sous-classes de `Parent`.
- Le nœud racine de l'arbre est associé à un objet de classe `Scene`. La **scène** correspond à la totalité de l'IG destinée à effectuer une tâche donnée.
- La scène, pour être affichée, doit être donnée à un objet de classe `Stage`[1] (le lieu où sera dessinée l'IG ; p. ex., sur un ordinateur de bureau : une fenêtre).

Cette organisation permet de facilement changer le contenu entier d'une fenêtre pour passer d'une tâche à l'autre : il suffit de dire au *stage* d'afficher une autre *scene*.

1. *scene* et *stage* : les deux se traduisent en Français par "scène" mais ont un sens très différent. *Scene* désigne une subdivision temporelle (scène = chapitre d'une pièce de théâtre), alors que *stage* désigne un lieu (scène = les planches sur lesquelles on joue la pièce).
Ainsi, pour éviter les confusions, soit je ne traduirai pas *stage* soit je dirai juste... une fenêtre !

# Exécution de l'IG en JavaFX

1. Pour des raisons techniques [1], la construction ne peut être faite directement depuis `main()` où une méthode appelée par `main()`.
2. À la place, on doit créer une classe `MonAppJFX extends Application`, pour laquelle il faudra implémenter la méthode `void start(Stage stage)`.
   C'est dans cette méthode qu'on initie la construction.
3. Pour démarrer l'interface graphique (par exemple depuis `main()`) on fait : [2]

```
Application.launch(MonAppJFX.class);
```

ou bien, dans le cas où on fait l'appel depuis `MonAppJFX`, juste :

```
Application.launch();
```

1. Des histoires de *threads* dont nous reparlons juste après.
Cette contrainte n'est pas spécifique à JavaFX, mais inhérente à la programmation évènementielle.
2. Cette instruction lance la méthode `start()` dans le *thread* des évènements JavaFX.

# Gérer les évènements en JavaFX

Pour gérer les évènements,

1. les gestionnaires d'évènement de JavaFX implémentent l'interface `EventHandler<T>` :

```
public interface EventHandler<T extends Event> {
  void handle (T e);
}
```

   (où `T` peut être remplacé par le type d'évènement à traiter, ex : `ActionEvent`, `KeyEvent`, `MouseEvent`, `...`).

2. on associe un gestionnaire d'évènement à un composant JavaFX ainsi : `composant.setOnXXXX(gestionnaire)` (ex : `void setOnMousePressed(EventHandler<? super MouseEvent> gest)`)

3. à partir de désormais, quand un évènement du type indiqué se produit, la méthode `handle()` du gestionnaire est exécutée.

# Gérer les évènements en JavaFX

## exemple



Prenons l'exemple d'`ActionEvent`. Je peux par exemple créer la classe :

```
public class GererEnregistrement implements EventHandler<ActionEvent> {
  private final Document doc;
  public GererEnregistrement(Document d) { this.doc = d; }
  public void handle(ActionEvent e) { d.enregistre(); }
}
```

Supposons maintenant que la variable `boutonEnregistrer` désigne un composant de type `Button`, alors pour que cliquer sur le bouton désormais déclenche l'enregistrement, il suffit d'ajouter l'instruction :

```
boutonEnregistrer.setOnAction(new GererEnregistrement(documentCourant));
```

Remarque : si `e` objet évènement, alors `e.getSource()` référence le composant où l'évènement a été créé. Cette référence peut servir faire un traitement différencié en fonction de l'état du composant.

# Gérer les évènements

## Utilisez les lambda-expressions !



`EventHandler` étant une interface fonctionnelle, lambda-expressions possibles :

Pour les gestionnaires courts, préférer les lambda-abstractions :

```
composant.setOnMousePressed(e -> { /* gérer l'évènement e ici */ });
```

Pour les gestionnaires longs, écrire un méthode et en passer une référence :

```
    void methodeDuClic(MouseEvent e) { /* gérer l'évènement e ici */ }
    ...
    composant.setOnMousePressed(controleur::methodeDuClic); // référence de méthode
```

Dernière technique intéressante si programme plus long qu'un simple exemple : la partie où on associe composants et gestionnaires est plus succincte et claire. En plus, on peut regrouper les méthodes de gestion d'événement dans dans une même classe [1].

1. le contrôleur du patron MVC

# Exécution des évènements

## Parler de *threads* sans en parler.

- Les méthodes `handle()` des gestionnaires d'évènement s'exécutent **les unes après les autres** (jamais en même temps).
- De même, ces gestionnaires commencent à s'exécuter seulement **après** la méthode `start()` de l'`Application` (et sa pile d'appels).
- En revanche, la méthode `main()` (et sa pile d'appels) continue à s'exécuter **en parallèle** → ne jamais tenter de modifier/ajouter un composant JavaFX depuis `main()` ou une méthode appelée par `main()` (résultats imprévisibles).
- **Remarque :** un traitement long [1] ou un appel bloquant dans un gestionnaire peut donc ralentir ou bloquer toute l'application.

  Si on a besoin d'exécuter du code long ou bloquant, il faudra le lancer **en parallèle** sur un autre *thread* (concept de ***worker thread*** [2]).

1. i.e. plus long que l'intervalle de rafraîchissement de la fenêtre
2. Vous pouvez, à cet effet, créer un *thread* classique ou, mieux, utiliser `javafx.concurrent.Worker`, l'API prévue par JavaFX, ou bien toute autre API de votre convenance.

# *Threads* en JavaFX

Pour parler "*threads*", ainsi 3 sortes de *threads* dans un programme JavaFX :

- 1 *thread* initial (`main`) : dans une application JavaFX, ne sert qu'à démarrer le JFXAT, via l'appel à `Application.launch()` en plaçant un évènement initial (l'exécution de `start()`) dans la file d'attente des évènements.
- 1 *thread* d'application JavaFX (JFXAT) qui lance les tâches de sa file d'attente (typiquement, les gestionnaires d'évènement).
  - les gestionnaires d'événements sont automatiquement programmés sur le JFXAT
  - ajout possible de tâches depuis autre *thread* avec `Platform.runLater(task)`.
  - Intérêt du JFXAT : permettre à l'IG de tourner indépendamment du reste du programme, en restant réactive.
  - Pourquoi restreindre toute manipulation de l'IG à ce seul JFXAT : on évite les problèmes usuels [1] du *multithreading* en rendant les exécutions de gestionnaires atomiques [2] les unes par rapport aux autres.
- parfois des ***worker threads*** pour les tâches longues ou bloquantes.

1. Entrelacements indésirables, accès en compétition, ...(cf. chapitre sur la concurrence)
2. Elles ne s'interrompent pas les unes les autres.

# Exemple JavaFX minimaliste





```
import javafx.application.Application; import javafx.scene.*; import
    javafx.scene.control.*; import javafx.scene.layout.BorderPane;

public class JFXSample extends Application {
    @Override public void start(Stage stage) throws Exception {
        MenuItem exit = new MenuItem("Exit"); exit.setOnAction(e -> System.exit(0));
        Menu file = new Menu("File"); file.getItems().add(exit);
        MenuBar menu = new MenuBar(); menu.getMenus().add(file);
        Label lbl = new Label("Exemple de Label qui tourne...");
        lbl.setOnMousePressed(e -> lbl.setRotate(lbl.getRotate() + 10));
        BorderPane root = new BorderPane(); root.setTop(menu); root.setCenter(lbl);
        Scene scene = new Scene(root, 400, 400);
        stage.setScene(scene);            // définir la scène à  afficher
        stage.setTitle("Application test");
        stage.show();                              // lancer l'affichage !
    }

    public static void main(String[] args) { Application.launch(args); }
}
```

# Composants de JavaFX

- Vous en avez vus quelques uns dans l'exemple précédent.
- Pour plus d'exemples, le mieux, c'est d'ouvrir les tutoriels que l'on peut trouver sur le web.
- Sinon, vous trouverez une liste exhaustive en regardant la documentation du *package* `javafx.scene.control` : `https://openjfx.io/javadoc/11/javafx.controls/javafx/scene/control/package-summary.html`

Attention quand vous trouvez de la documentation : vérifiez qu'elle concerne bien la version installée chez vous. [1].

1. Les moteurs de recherche tendent encore trop à référencer d'anciennes versions comme JavaFX 2...

# Organiser une application graphique

## Séparation vue et modèle



C'est souvent une bonne idée de séparer les deux aspects suivants :

- **Modèle :** cœur du programme, partie « métier ». C'est ici que sont gérées, organisées, traitées les données. On y trouve les déclarations de structures de données ainsi que les méthodes implémentant les différents algorithmes traitant sur ces structures.
- **Vue :** partie qui sert à présenter l'application à l'utilisateur et sur laquelle l'utilisateur agit.

Idéalement, les classes du modèle **ne dépendent pas** des classes de la vue [1].

Plusieurs stratégies pour coordonner M et V, notamment : Model-View-Controller (MVC), Model-View-Presenter (MVP) et Model-View-ViewModel (MVVM) (dans les 3 cas : ajout d'un 3e composant).

1. ce qui permet de changer la présentation de l'application, notamment pour la porter sur des plateformes différentes

# Organiser une application graphique

## L'architecture MVC

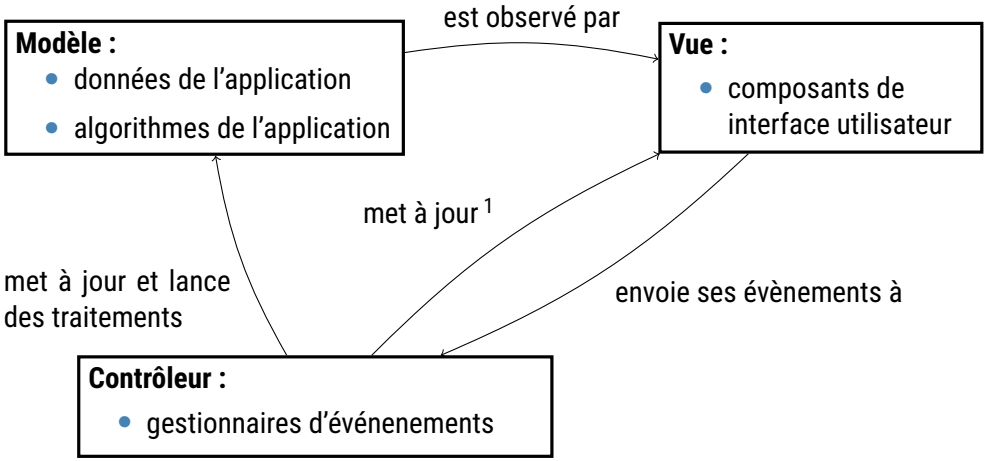


Architecture MVC décrite depuis 1978... encore très populaire pour les applications graphiques, notamment les applications web. Le troisième composant est le **contrôleur**.

- **Modèle :** déjà décrit.
- **Vue :** déjà décrite.
- **Contrôleur :** partie du programme servant à interpréter les évènements (entrées de l'utilisateur dans la vue, mais pas seulement) pour agir sur le modèle (déclencher un traitement, ...) et la vue (ouvrir un dialogue, ...).

JavaFX est particulièrement adapté à mettre en œuvre une stratégie MVC.

# Illustration




1. pour la partie de l'interface utilisateur qui ne sert pas à la présentation des données, par exemple : ouverture des menus, boîtes de dialogue, etc.

# Le patron de conception Observateur/Observable

## En général



- **Cas d'application :** quand les changements d'état d'un objet donné (l'**observable**) peut avoir des répercussions sur de multiples autres objets (les **observateurs**).
- **Principe :** Chaque observable contient une liste d'observateurs abonnés. Quand un changement a lieu, il appelle sur chaque élément de cette liste une même méthode (d'une interface commune) pour prévenir tous ses observateurs.
- **Intérêt :** éviter que la classe de cet objet ne dépende des classes de ses observeurs (la dépendance se créée entre objets, à l'exécution).
- Patron utilisé pour la relation Modèle $\leftrightarrow$ Vue dans MVC : V réagit aux changements de M, sans que celui-ci n'ait de dépendance vers la seconde.

# Le patron de conception Observateur/Observable

## Avec `java.util`



**Implémentation “historique” de Java :** interface `java.util.Observer` et classe `java.util.Observable`. Regardez leurs documentations.

En plus de ce qu’on vient de décrire, `java.util.Observable` contient un mécanisme pour éviter de notifier tout le temps les observateurs, via le couple de méthodes `setChanged()`/`hasChanged()` :

- `setChanged()` : sert à signaler que l’observable vient d’être modifié ;
- `hasChanged()` : renvoie **`true`** si l’observable a été modifié depuis la dernière notification aux observateurs.

# Le patron de conception Observateur/Observable

## Avec `javafx.beans` (1)



JavaFX a sa propre implémentation de ce patron.

Les **propriétés** des composants graphiques (mais pas seulement) sont matérialisées par des instances de `javafx.beans.Property`, sous-interface de `javafx.beans.Observable`.

Pour toute propriété de nom “`value`”, le composant contiendra les 3 méthodes suivantes :

- `public Double getValue()` : lire la valeur de la propriété
- `public void setValue(Double value)` : modifier la valeur de la propriété
- `public DoubleProperty valueProperty()` : obtenir l’objet-propriété lui-même

# Le patron de conception Observateur/Observable

## Avec `javafx.beans` (2))



Comme ces propriétés sont observables, on peut leur ajouter des observateurs :

```
slider.valueProperty().addListener(
    (observable, oldVal, newVal) -> afficheur.textProperty().setValue("valeur : " +
        newVal));
```

**Remarque :** les composants graphiques contiennent de telles propriétés (cf. exemple), mais elles peuvent aussi être utilisées dans toute autre classe, notamment dans la partie Modèle de l'application, permettant l'observation du Modèle par la Vue.

# Quelques erreurs





- Quelques erreurs :
  - division par 0,
  - accès à un indice d'un tableau supérieur ou égal à la longueur de celui-ci,
  - appel d'une méthode sur récepteur `null`,
  - tentative d'affecter une valeur à une variable d'un type incompatible : `int[] t = "toto"`.
- Grâce au typage statique de Java, la dernière erreur est détectée dès la compilation... mais, dans cette liste d'exemples, c'est la seule !
- Que faire des autres erreurs ?

# Autre scénario embêtant...

Que se passe-t-il d'indésirable quand on exécute le programme suivant ?

```
static int factorielle(int n) {
  if (n==0) return 1;
  else return n * factorielle(n-1);
}

static void main(String[] args) { System.out.println(factorielle(-2)); }
```

Quelles solutions voyez-vous, quels sont leurs propres inconvénients ?

- tester $n < 0$ et retourner... quoi ?
  - un code d'erreur ? quelle valeur ? -1 ? [1]
  - et si par mégarde on appelle `factorielle(-12)`, ne risque-t-on pas d'utiliser la valeur -1 comme si c'était réellement une factorielle correcte et provoquer une autre erreur bien plus tard dans l'exécution ? [2]

→ il existe des façons plus « propres » et plus sûres de traiter ce cas !

1. ou **`null`**, si on devait retourner une référence
2. ou, dans le cas des références, appeler une méthode sur la valeur **`null`** → crash sur `NullPointerException` !

# Qu'est-ce qu'une erreur ?

Les « erreurs » se manifestent à plusieurs niveaux :

1. erreur à la compilation (syntaxe, typage, ...) : le compilateur refuse de compiler le programme, qui ne sera jamais exécuté tel quel
2. crash [1] à l'exécution, avec explication (pile d'appel)
3. comportement incorrect : le programme continue à tourner ou bien termine sans signaler d'erreur mais ne fait pas ce qu'il est censé faire.

Ces catégories sont en fait du moins grave au plus grave.

1. Les erreurs à la compilation se corrigent avant de livrer le produit et ne provoqueront jamais aucun dégât.
2. Les crashs sont plus graves (se produisent dans des programmes déjà en production) mais, quand ça arrive, on sait qu'il y a un bug à corriger.
3. Pire cas : le programme peut fonctionner très longtemps en faisant mal son travail sans qu'on ne s'en aperçoive (parfois, conséquences désastreuses, cf. Ariane 5).

1. correspond à une exception non rattrapée

# Gérer les erreurs

## En quoi cela consiste-t-il ?



- Avoir une « hygiène » qui tend à faire ressortir les erreurs de programmation dès la compilation (notamment via le typage fort).
- Utiliser les options du compilateur (`-Xlint`) et des outils complémentaires d'analyse statique pour détecter plus d'erreurs avant exécution.
- Savoir quand le composant qu'on programme génère ses propres erreurs. Dans ce cas, il faut les signaler de façon propre aux client du composant et/ou fournir des façons de les éviter.
- Savoir réagir quand un composant qu'on utilise remonte une erreur :
  - prendre en compte l'erreur et proposer un comportement (correct) alternatif,
  - ou bien propager l'erreur (si possible en ajoutant de l'information ou en la traduisant en tant qu'erreur du composant courant) pour qu'un client la traite.

# Techniques de gestion des erreurs

Techniques proposées dans ce chapitre :

- lancer (et rattraper) des **exceptions** (nous allons voir ce que c'est) ;
- ajouter une méthode auxiliaire pour pré-valider un appel de méthode ;
- utiliser un type de retour « enrichi ».

# Exceptions

## Description du mécanisme



Le mécanisme des « exceptions » :

- consiste à gérer un comportement « exceptionnel » du programme, sortant du flot de contrôle « normal » ;
- sert quand il n'y a pas de valeur sensée à passer dans un `return` (la méthode est dans l'incapacité de terminer normalement [1])
  $\rightarrow$ on ne veut pas redonner la main à l'appelant comme si rien d'anormal ne s'était passé ;
- concerne des événements « exceptionnels » = « rares » (l'exécution de ce mécanisme est en fait coûteuse).

1. Que répondriez-vous si on vous demandait : « Quel nombre réel vaut dix divisé par zéro ? »

# Exceptions

## Factorielle corrigée grâce aux exceptions



```java
class FactorielleNegativeException extends Exception {}

public class Test {
    static int fact(int x) throws FactorielleNegativeException {
        if (x < 0) throw new FactorielleNegativeException();
        else if (x == 0) return 1;
        else return x * fact(x - 1);
    }

    public static void main(String args[]) {
        System.out.println("Entrez un entier");
        int x = (new Scanner(System.in)).nextInt();
        try {
            System.out.println("La factorielle est : " + fact(x));
        } catch (FactorielleNegative e) {  // erreur détectée !
            System.out.println("Votre nombre était négatif !");
        }
    }
}
```

# Exceptions

## Explication rapide



- Signaler une exception (grâce à l'instruction **`throw`**) fait sortir de la méthode sans exécuter de **`return`**.[1]
- L'exception peut ensuite être rattapée ou non.
- Non-rattrapée $\rightarrow$ le programme se quitte en affichant un message d'erreur
- Rattrapée, si
  - exécution sous la portée dynamique du **`try`** d'un groupe **`try ... catch ...`**
  - l'exception a le type donné entre **`( )`** après le **`catch`**

  $\rightarrow$ exécution du bloc d'instruction de ce **`catch`**.
- La méthode contenant **`try ... catch ...`** n'est pas nécessairement celle qui appelle directement la méthode qui fait **`throw`** (traitement non local de l'erreur).

1. **`throw`** = sortie exceptionnelle ; **`return`** = sortie normale

# Exceptions

## Non-localité du mécanisme



**Supposons :**

- que `main` appelle la méthode `f1` qui appelle `f2` qui appelle ... qui appelle `fn` ($\rightarrow$ pile d'appel de méthodes avec `main` en bas de la pile, `fn` en haut)
- et que `fn` signale une exception `exn`

**alors**

- si `fn` rattrape `exn`, on retrouve un fil d'exécution « normal » [1], sinon, on sort de `fn` de façon « exceptionnelle » $\rightarrow$ alors c'est comme si l'exception se produisait dans `fn-1` (**propagation** de l'exception).
- si `fn-1` la rattrape, exécution normale du **`catch`**, sinon propagation à `fn-2` etc.
- si l'exception est propagée jusqu'à `main` et `main` ne la rattrape pas, le programme se quitte en affichant l'exception [2].

1. On exécute le **`catch`**, le **`finally`**, puis les instructions d'après.
2. Dont les informations très utiles qu'elle contient.

# Exceptions

## Les mots-clés et leur syntaxe complète



- `throw new MonException(...);` : instruction pour signaler une exception.
- `try { -1- } catch ( -2- ){ -3- } ... catch ( ... ){ ... } finally { -4- }` : bloc (instruction) de traitement des exceptions.
  - On essaye d'abord d'exécuter le bloc `{ -1- }` du `try` (bloc protégé),
  - Si une exception se produit, pour chaque `catch`, on regarde si l'exception a le type donné dans les `( -2- )` et on exécute le bloc `{ -3- }` du premier `catch` qui correspond.
  - Enfin on exécute le bloc `{ -4- }` du `finally` (dans tous les cas).

  Le `try` doit toujours être suivi d'au moins une clause `catch` ou `finally`[1].
- `... maMethode(...)throws ExceptionType { ... }` : clause indiquant que la méthode déclarée peut signaler[2] une exception.[3]

1. Sauf construction *try-with-resource*, voir plus loin.
2. signaler ou lever (*raise* : mot-clé utilisé en OCaml) ou lancer/jeter (*throw*)
3. Cette clause est parfois obligatoire, parfois pas, détails plus loin.

# Le bloc try/catch

## Détails



- Paramètre de `catch` = déclaration de variable d'un sous-type de `Throwable`.
- **Attention :** en cas de plusieurs clauses `catch`, exceptions doivent être traitées dans l'ordre de sous-typage (les sous-types avant les supertypes) !

```
public class Exception1 extends Exception { }
public class Exception2 extends Exception1 { }
try { .. }
catch (Exception2 e2) { ... }
catch (Exception1 e1) { ... }
```

Si on inverse les deux `catch`, erreur de compilation.

```
error: exception Exception2 has already been caught
```

**Raison :** `Exception2` est un cas particulier de `Exception1`, donc déjà traité dans le premier `catch`. Le deuxième `catch` est alors inutile. [1]

- Variante (« *multi-catch* ») : `catch (Exception1 | Exception2 e){ ... }`

1. Or le compilateur considère que si on écrit du code inutile, c'est involontaire et donc une erreur.

# La construction *try-with-resource* [1]

```
try (Scanner sc = new Scanner(System.in)) {
    System.out.println("Nom ?"); String nom = sc.nextLine();
    System.out.println("Prénom ?"); String prenom = sc.nextLine();
    identite = new Personne(nom, prenom);
}
```

- syntaxe : `try (Resource r = ? /*expr */){ /*instr */? }`
- équivalent : `Resource r = ?; try { ? } finally { r.close(); }`
- Assure que la ressource utilisée sera libérée après usage (avec ou sans exception).
- `Resource` doit implémenter :
  `interface AutoCloseable { void close(); }` (c'est le cas de `Scanner`).
- `r` doit être une variable finale ou effectivement finale.
- on peut aussi écrire juste `try (r){ ... }` si `r` est une variable `AutoCloseable` (effectivement) finale déjà déclarée, ce qui autorise à l'initialiser séparément.

1. Construction introduite dans Java 7 ; initialisation de resource séparée introduite dans Java 9.

# Exceptions

## Mais au fait qu'appelle-t-on une exception ?



**Remarque :** on a utilisé le nom « exception » pour parler de plusieurs choses différentes...

- l'événement qui se produit quand on quitte une méthode via le mécanisme qu'on vient de décrire : « une exception vient d'être signalée » ;
  (dans l'exemple, c'est ce qui se produit quand on exécute `throw new FactorielleNegative();`)
- l'objet qui est passé du point de programme où le problème a lieu au point du programme où on gère le problème ;
  (dans l'exemple, l'objet instancié en faisant `new FactorielleNegative();`, récupéré dans la variable `e` quand on fait `catch (FactorielleNegative e)`)
- à la classe d'un tel objet.
  (dans l'exemple : la classe `FactorielleNegativeException`)

Parlons maintenant des objets et des classes !

# Les objets exception

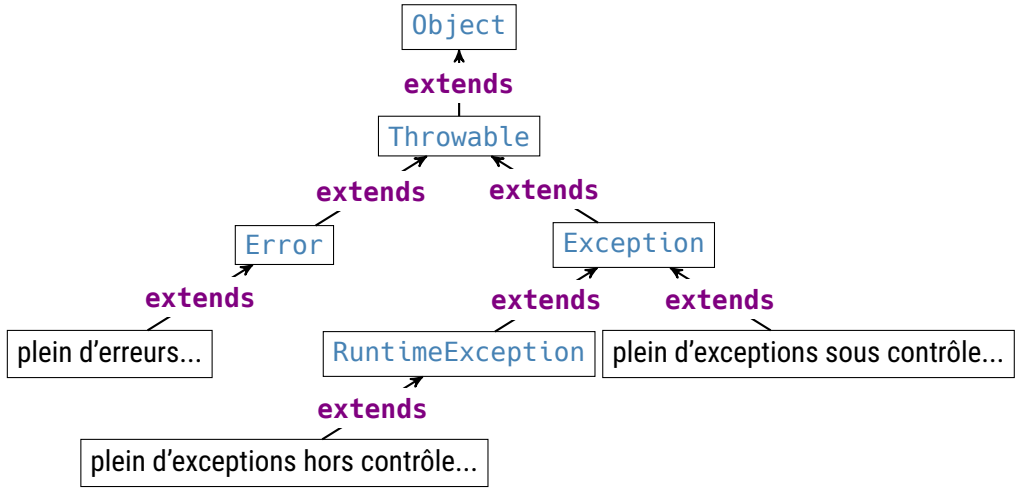
- Objet exception = objet passé en paramètre de **`throw`**, récupérable par **`catch`**.
- Instance (indirecte) de la classe `Throwable`.
- Contient de l'information utile pour récupérer l'erreur (bloc **`catch`**) ou bien déboguer un crash, notamment :
  - sa classe : le fait d'appartenir à l'une sous-classe d'exception ou une autre est déjà une information très utile.
  - message d'erreur, expliquant les circonstances de l'erreur → `String getMessage()`
  - cause, dans le cas où l'exception est elle-même causée par une autre exception → `Throwable getCause()`
  - trace de la pile d'appels, qui donne la liste des appels imbriqués successifs (numéro de ligne et méthode) qui ont abouti à ce signalement d'exception → `StackTraceElement[] getStackTrace()`

**Attention :** génération de la trace → coûteuse en temps et en mémoire.
→ Une raison pour réserver le mécanisme des exceptions aux cas exceptionnels.

# Classes d'exceptions

## Hiérarchie d'héritage





# Classes d'exceptions

## Description des classes proposées (1)



- Classe `Throwable` :
  - Rôle = marqueur syntaxique pour **`throw`**, **`throws`** et **`catch`** → c'est le type de tout ce qui peut être « lancé » et rattrapé.
  - Après **`throw`** l'expression doit être de (super-)type `Throwable`.
  - Après **`throws`** n'apparaissent que des sous-classes de `Throwable`.
  - Le paramètre déclaré dans un **`catch`** a pour type un sous-type de `Throwable`.
- Classe `Error` :
  - Indique les erreurs tellement graves qu'il n'y a pas de façon utile de les rattraper.
  - On a le droit de les passer en argument d'un **`catch`**... mais ce n'est pas conseillé !
  - Ex. : dépassement de la capacité de la pile d'exécution (`StackOverflowError`),
- Classe `Exception` :
  - Erreurs possiblement récupérables.
  - Elles ont vocation à être passées en argument d'une clause **`catch`**.
  - Exemples :
    - opération bloquante interrompue (`InterruptedException`).

# Classes d'exceptions

## Description des classes proposées (2)



- Classe `RuntimeException` :
  - Erreurs se produisant au cours d'une exécution normale [1] du « *runtime* » (i.e. : la JVM).
  - Très souvent : erreurs évitables par de simples vérifications à l'exécution (`if`s)
    → faire en sorte qu'elles ne se produisent pas, **ne pas** rattraper dans un `catch` !
    Exemples :
    - division par 0 (`ArithmeticException`),
    - appel de méthode sur `null` (`NullPointerException`),
    - accès à une case qui n'existe pas dans un tableau (`ArrayOutOfBoundException`),
    - tentative de *cast* illégale (`ClassCastException`)
  - Mais, de plus en plus utilisées à la place des `Exception` normales, afin d'éviter les contraintes des exceptions sous-contrôle. (`throws` non requis).
    Dans ce cas : vocation a être rattrapée dans un `catch` malgré tout.

**Remarque :** l'usage qui est fait de `RuntimeException` est fondamentalement différent des autres sous-classes de `Exception`. Ainsi, « moralement », il est hasardeux de considérer `RuntimeException` comme sous-type de `Exception` (même si c'est vrai).

1. Traduction : si erreur alors que la JVM s'exécute normalement, c'est que le programme a un bug !

# Classes d'exceptions

## Créer ses propres exceptions



- **Attention :** avant de créer une exception, vérifiez qu'une classe d'exception standard n'est pas déjà définie dans l'API Java pour ce cas d'erreur. [1]
- On crée une classe d'exception personnalisée (le plus souvent) comme sous-classe d'`Exception` ou de `RuntimeException`.
- Le `String` passé au constructeur est le message retourné par `getMessage()`.
- Le `Throwable` passsé au constructeur est la valeur retournée par `getCause()`. Cf. chaînage causal des exceptions.
- **Avertissement :** on ne peut pas déclarer de sous-classe générique de `Throwable`. En effet, `catch` (comme `instanceof`) doit tester le type de l'exception à l'exécution. Or à l'exécution, la valeur du paramètre n'est plus disponible [2]. Donc déclarer une telle classe est inutile. Donc le compilateur l'interdit.

1. Très souvent, c'est en fait `IllegalArgumentException` ou `IllegalStateException` qui conviendrait...
2. voir effacement de type

# Chaîner les exceptions

```
    static void f1() { try { f2(); } catch (E2 e) { throw new E1(e); } }
    static void f2() { try { f3(); } catch (E3 e) { throw new E2(e); } }
    static void f3() { throw new E3(); }
    public static void main(String args[]) { f1(); }
```

Quand on exécute, la JVM crashe et affiche l'exception non rattrapée (instance de E1), ainsi que toutes ses causes sucessives.

```
Exception in thread "main" exceptions.E1: exceptions.E2: exceptions.E3
    at exceptions.Exn.f1(Exn.java:25)
    at exceptions.Exn.main(Exn.java:42)
Caused by: exceptions.E2: exceptions.E3
    at exceptions.Exn.f2(Exn.java:33)
    at exceptions.Exn.f1(Exn.java:23)
    ... 1 more
Caused by: exceptions.E3
    at exceptions.Exn.f3(Exn.java:38)
    at exceptions.Exn.f2(Exn.java:31)
    ... 2 more
```

# Exceptions sous contrôle vs. hors contrôle

## La clause **throws**



- Accolée à la déclaration d'une méthode, la clause **`throws`** signale qu'une exception non rattrapée peut être lancée lors de son exécution.

```
public static void f() throws MonException {
    // Code pouvant générer une exception de type MonException.
}
```

- Cette clause est obligatoire pour les exceptions dites « sous contrôle »[1].
- Conséquence : si `MonException` est sous-contrôle et que `f()` (ci-dessus) est appelée dans une autre méthode `g()` qui ne la rattrape pas, alors `g()` doit elle-même avoir **`throws`** `MonException`, et ainsi de suite.
- Du coup un programme ne peut pas crasher sur une exception sous contrôle, sauf si elle est déclarée dans la signature de `main()`.[2]

1. voir page suivante
2. En fait si... on peut tricher (ce n'est pas évident), mais c'est déconseillé !

# Exceptions sous contrôle vs. hors contrôle

## Qui est quoi et pourquoi ?



Deux cas :

- Les exceptions sous contrôle (*checked*) doivent toujours être déclarées (`throws`).

  **Lesquelles ?** toutes les sous-classes de `Exception` sauf celles de `RuntimeException`.

  **Raison :** les concepteurs de Java ont pensé souhaitable [1] d'inciter fortement à récupérer toute erreur récupérable.
- Les exceptions hors contrôle (*unchecked*) n'ont pas besoin d'un tel signalement.

  **Lesquelles ?** uniquement les sous-classes de `Error` et de `RuntimeException`.

  **Raisons :**
  - Les `Error` sont considérées comme fatales : aucun moyen de continuer le programme de façon utile.
  - Les `RuntimeException` peuvent se produire, par exemple, dès qu'on utilise le « . » d'appel de méthode, autant dire tout le temps ! Si elles étaient sous contrôle, presque toutes les méthodes auraient `throws NullPointerException` !

1. Ce point est très controversé. Aucun langage notable, conçu après Java, n'a gardé ce mécanisme.

# Exceptions

## Remarque sur la non-localité

- Non-localité = point fort des exceptions : en effet l'erreur n'est mentionnée que là où elle se produit et là où elle est traitée. [1]
- Mais la non-localité peut être contradictoire avec l'encapsulation.

→ si un composant B utilise un composant A, B doit « masquer » les exceptions de A. En effet : si C utilise B mais pas A, C ne devrait pas avoir affaire à A. [2]

- Bonne conduite : traiter toutes les erreurs de A dans B. À défaut, traduire les exceptions de A en exceptions de B (en utilisant le chaînage) :

```
class A { public void f() { throw new AException(); } }
class B {
    private A a;
    public void g() {
        try { a.f(); } catch (AException e) { throw new BException(e); }
    }
}
```

1. Pas tout à fait vrai avec les exceptions sous contrôle.
2. Les exceptions sous contrôle rendent le problème particulièrement visible vu qu'elles sont affichées dans la signature des méthodes, mais il existe aussi avec les exceptions hors contrôle.

# Pré-valider les appels de méthode

## Description de la technique



Vous avez déjà vu cette technique mise à l'œuvre dans l'API Java. Exemples :

- avec les scanners : avant de faire `sc.nextInt()`, il faut faire `sc.hasNextInt()` pour être sûr que le prochain mot lu est interprétable comme entier
- avec les itérateurs : avant l'appel `it.next()`, on vérifie `it.hasNext()`.

**En résumé :** une méthode dont le bon fonctionnement est soumis à une certaine précondition peut être assortie d'une méthode booléenne retournant la validité de la précondition. En général, les deux méthodes ont des noms en rapport :

- `void doSomething()`/`boolean canDoSomething()`
- `Foo getFoo()`/`hasFoo()`

Le même test doit malgré tout être laissé au début de `doSomething()` :

```
if (!canDoSomething()) throw new IllegalStateException(); // ou IllegalArgumentException
```

(→ si on oublie de tester `canDoSomething` au pire, crash, au lieu de résultat absurde)

# Pré-valider les appels de méthode

## Exemple



Pour la factorielle, on peut créer une méthode de validation du paramètre, mais le plus simple c'est encore de penser à tester $x \geq 0$ avant de l'appeler.

La méthode factorielle s'écrira elle-même avec ce test et une exception hors contrôle :

```
static int fact(int x) {  // pas de throws
        if (x < 0)
                throw new IllegalArgumentException(); // hors contrôle
        else if (x == 0)
                return 1;
        else
                return x * fact(x - 1);
}
```

# Enrichir le type de retour

Il s'agit d'une amélioration de la technique du « code d'erreur » : au lieu de réserver une valeur dans le type, on prend un type somme, « plus large »

- contenant, sans ambiguïté, les vraies valeurs de retour et les codes d'erreur,
- et tel qu'on ne puisse pas utiliser la valeur de retour directement sans passer par un *getter* « fait pour ça ».

Possibilités :

- programmer un « type somme » à la main
- s'il n'y a qu'un seul code d'erreur (ou bien si on ne veut pas distinguer les différentes erreurs), utiliser la classe `Optional<T>` (Java 8).
- si la méthode qui produit l'erreur devait être **`void`**, retourner à la place un **`boolean`** (si on ne souhaite pas distinguer les erreurs) ou mieux, une valeur de type énuméré (chaque constante correspond à un code d'erreur).

# Enrichir le type de retour

## Exemple 1



Toujours avec la factorielle, avec le type `Optional` :

```
static Optional<Integer> fact(int x) {  // pas de throws
        if (x < 0) return Optional.empty();
        else if (x == 0) return Optional.of(1);
        else return Optional.of(x * fact(x - 1).get());
}
```

Comme cette méthode ne retourne pas un **`int`** (ou `Integer`), on n'est pas tenté d'utiliser la valeur de retour directement.

Pour utiliser la valeur de retour :

```
Optional<Integer> result = fact(x);
if (result.isPresent()) System.out.println(x + "! = " + result.get());
else System.out.println(x + " n'a pas de factorielle !'");
```

Autre possibilité : `result.ifPresent(f -> System.out.println(f));`[1]

1. opérateur « –> » introduit dans Java 8, sert à dénoter une valeur fonctionnelle. *Hors programme !*

# Enrichir le type de retour

## Exemple 2



```java
public class Mail {

    /* attributs, constructeurs, etc. */

    public static enum SendOutcome { OK, AUTH_ERROR, NO_NETWORK, PROTOCOL_ERROR }

    /*
     send() : envoie le mail. Sans les erreurs, void suffirait...
     ... mais évidemment des erreurs sont possbles.
     Différents codes sont prévus dans l'enum SendOutCome
    */

    public SendOutcome send() {
        /*
            essaye d'envoyer le mail, mais interrompt le traitement
            avec "return error.TRUC;" dès qu'il y a un souci
        */
        return SendOutcome.OK;
    }
}
```

# Points forts et limites de chaque approche

## Exceptions (1)



- Toutes les exceptions :
  - L'instanciation d'un `Throwable` est coûteux (génération de la pile d'appel, etc.)
  - L'exécution de **`throw`** est coûteuse (proportionnelle à la hauteur de pile d'appel à remonter pour arriver au **`catch`**).
    Cependant : ce coût est inévitable pour pouvoir obtenir un traitement non local [1].
- Exceptions hors contrôle :
  - on peut oublier de les prendre en compte ($\rightarrow$ crash, alors que le programme n'aurait pas dû compiler du tout si l'exception avait été sous contrôle)

$\rightarrow$ idéales, soit pour les erreurs à traitement non local [2], soit pour les erreurs pour lesquelles il est acceptable de laisser crasher le programme.

1. En effet : quel que soit le mécanisme utilisé, un traitement « non local » signifie qu'on doit remonter un nombre arbitraire de niveaux dans la pile d'appels.
2. À condition de ne pas oublier de toutes les traiter ! À cet effet, penser à documenter les méthodes qui lèvent de telles exceptions $\rightarrow$ mot-clé `@throws` de la JavaDoc.

# Points forts et limites de chaque approche

## Exceptions (2)



- Exceptions sous contrôle :
  - « Pollution syntaxique » : obligation d'ajouter au choix des **`throws`** ou des **`try`** dans toute méthode ou une telle exception peut se produire.
    - Si on sait traiter l'exception localement, pas de problème. [1]
    - Sinon on se retrouve à ajouter une clause **`throws`** à l'appelant (qui vient s'ajouter aux autres s'il y en avait déjà... ), puis à l'appelant de l'appelant, puis ... [2]
  - Incitation à faire des **`try catch`** juste pour éviter d'ajouter un **`throws`** : nuisible si on fait taire un problème sans le traiter. P. ex. :

```
void f() throws Ex1 { throw new Ex1(); }
void g() {
  try { f(); } catch (Ex1 e) { /* rien ! <-- ouh, pas bien ! */ }
}
```

→ la pratique moderne semble éviter de plus en plus l'utilisation de ce mécanisme. Cependant l'API Java l'utilise beaucoup et il faut donc savoir comment faire avec.

1. Mais pour un traitement local, à quoi bon utiliser les exceptions ?
2. Modification de signature des appelants qui finalement revient à peu près à changer les types de retour par des types enrichis. → critique courante, dans ce cas, qu'apportent les exceptions sous-contrôle en plus ?

# Points forts et limites de chaque approche

## Méthodes auxiliaires

- Si vérifier les paramètres est aussi coûteux qu'effectuer l'opération, un appel sécurisé devient deux fois plus long.
  → réserver aux cas où la vérification a priori est peu coûteuse
- Le compilateur ne sait pas vérifier qu'on a appelé et testé `canDoSomething()` avant d'appeler `doSomething()`.
  → problèmes potentiellement reportés à l'exécution.
- Parfois (rarement), on ne sait pas quoi faire quand l'appel à `canDoSomething()` retourne **`false`**. Peut-être que seul l'appelant de l'appelant de l'appelant de... l'appelant connait suffisament de contexte pour traiter le cas d'erreur.
  → (relativement) peu adapté au traitement d'erreur non local [1]

1. On peut propager de la façon suivante : l'opération **`void`** `doFoo()` appelle **`void`** `doBar()`, mais `doBar()` a une méthode de vérification **`boolean`** `canBar()`, alors on peut écrire une méthode de vérification **`boolean`** `canFoo()` pour l'opération `doFoo()`, qui appellera `canBar()`.
Mais ce n'est quand-même pas pratique !

# Points forts et limites de chaque approche

## Type de retour enrichi



- « Pollution syntaxique » : il est bien plus concis et clair de travailler sur des `int` que des `Optional<Integer>` (déclarations plus courtes, pas de méthodes spéciales pour accéder aux valeurs).
- La « pollution syntaxique » apparait dans les signatures de toutes les méthodes de la pile jusqu'à celle qui traite le cas d'erreur

  $\rightarrow$ à réserver de préférence pour erreurs à traitement local.
- Perte d'efficacité par rapport à valeur directe : on crée un nouvel objet à chaque fois qu'on retourne de la méthode.

  $\rightarrow$ à réserver pour situation où cas d'erreur aussi fréquent que cas « normal » (en comparaison : `throw new E();` coûteux mais ok si rare).

$\rightarrow$ cette technique pose les mêmes problèmes que les exceptions sous contrôle pour les erreurs traitées non localement, mais peut se révéler plus efficace localement.

# Points forts et limites de chaque approche

## Synthèse des critères



Pas toujours de choix unique et idéal quant à la stratégie de gestion d'une erreur. Les critères suivants, avec leurs exigences contradictoires, peuvent être considérés :

- localité du traitement de l'erreur (local, non local ou pas de traitement) ;
- coût de la vérification de la validité de l'opération avant de l'effectuer ;
- sûreté, c.-à-d. obligation de traiter le cas d'erreur (contre-partie : pollution syntaxique) ;
- fréquence de l'erreur (totalement évitable en corrigeant le programme, rare, fréquente, ...) ;
- qualité de l'encapsulation requise.