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.

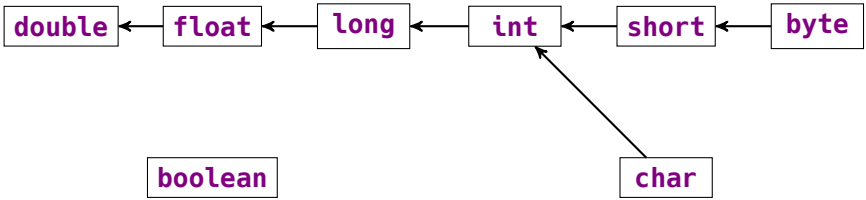
1. Cf. cours sur les interfaces et sur l'héritage pour voir quelles sont les contraintes exactes.

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]

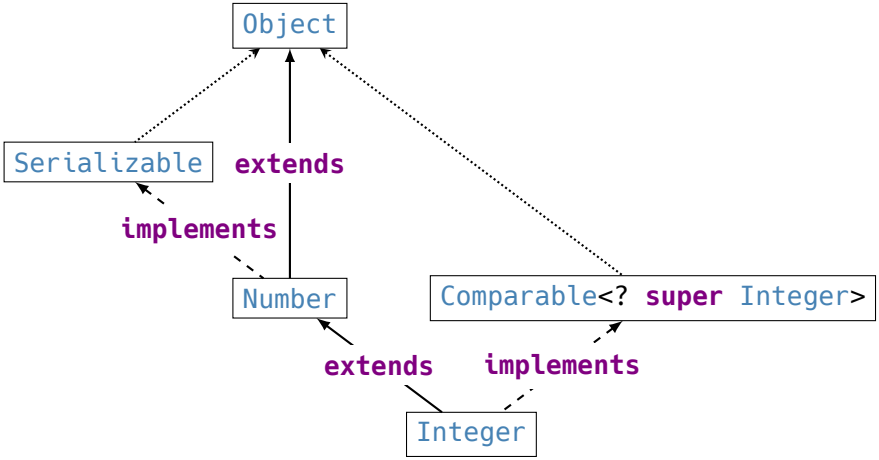
- (**implémentation d'interface**) A_{i-1} est une classe, A_i est une interface et A_{i-1} implémente A_i ;
- (**héritage de classe**) A_{i-1} et A_i sont des classes et A_{i-1} étend A_i ;
- (**héritage d'interface**) A_{i-1} et A_i sont des interfaces et A_{i-1} étend A_i ;
- (**covariance des types 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 ≤ 32 bits vers ≤ 32 bits : dans code-octet et JVM, granularité de 32 bits $\rightarrow$ tous les "petits" entiers codés de la même façon $\rightarrow$ aucune modification nécessaire. [1]
- Pour une conversion de 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 { ... }`.

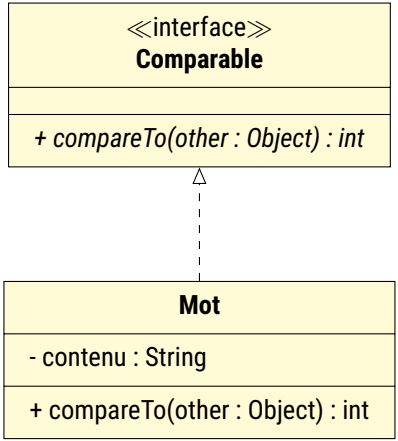
Interfaces : usage (2)

une méthode de tri polymorphe

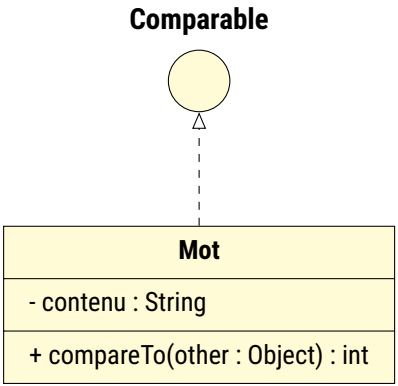

```
public class Tri {
    static void trie(Comparable [] tab) {
    /* ... algorithme de tri
        utilisant tab[i].compareTo(tab[j])
        ...
    */
    }

    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 {}
```

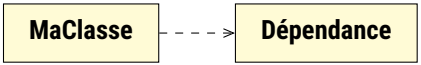
Ce programme compile et `new A().f();` affiche B.

Du bon usage des interfaces

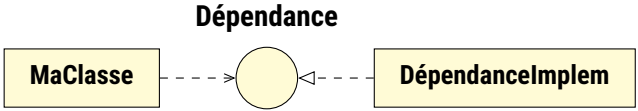
Programmez à l'interface (1)

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

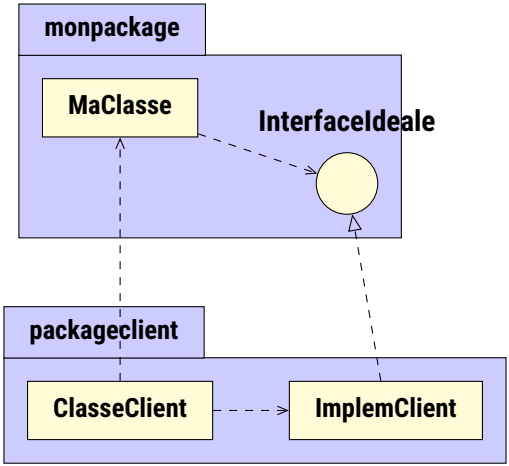
Principe d'inversion de dépendance [4] (DIP) (1)

- **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