Résumé du cours, compilation et exécution

Théorie et pratique de la programmation
Michel Schinz – 2013-05-27

Collections

Types de collections

Nous avons examiné trois types de collections :

- les **listes**, collections ordonnées d'éléments éventuellement dupliqués,
- les **ensembles**, collections non ordonnées d'éléments sans duplication,
- les **tables associatives**, collections non ordonnées de paires (clef, valeur) avec recherche rapide de la valeur associée à une clef.

Organisation des éléments

Les collections contiennent, par définition, un nombre variable d'éléments. Deux techniques de base existent pour organiser ces éléments :

- les **tableaux** (organisation spatiale) : les éléments sont placés côte-à-côte en mémoire,
- le **chaînage** : les éléments sont placés arbitrairement en mémoire mais sont liés entre-eux par des références.

Les tableaux ont l'avantage d'offrir un accès en O(1) à un élément étant donné son index. Leur inconvénient majeur est leur rigidité qui rend l'ajout et la suppression d'éléments coûteuse dans la plupart des cas.

Mise en œuvre : listes

Nous avons examiné deux mises en œuvre des listes :

- Les **tableaux-listes**, qui offrent un accès en O(1) à un élément étant donné son index, mais les insertions et suppressions sont en O(*n*), sauf à la fin.
- Les **listes chaînées**, qui offrent des insertions et suppressions en O(1) à n'importe quelle position, mais un accès à un élément étant donné son index en O(*n*), sauf éventuellement le premier et le dernier.

Mise en œuvre : ensembles

Nous avons examiné deux mises en œuvre des ensembles :

- les **arbres binaires de recherche**, qui utilisent le chaînage mais tirent parti d'un ordre sur les éléments pour offrir un accès en $O(\log n)$,
- les **tables de hachage**, qui combinent tableau et chaînage pour offrir – sous certaines conditions – un accès en $O(1)$.

Arbre binaire de recherche

Dans un arbre binaire de recherche, tous les éléments du fils gauche d'un nœud sont strictement plus petits que celui du nœud, ceux du fils droit strictement plus grands.
Cette propriété permet de diriger le parcours et offre un accès en O(log *n*) si l'arbre est équilibré.

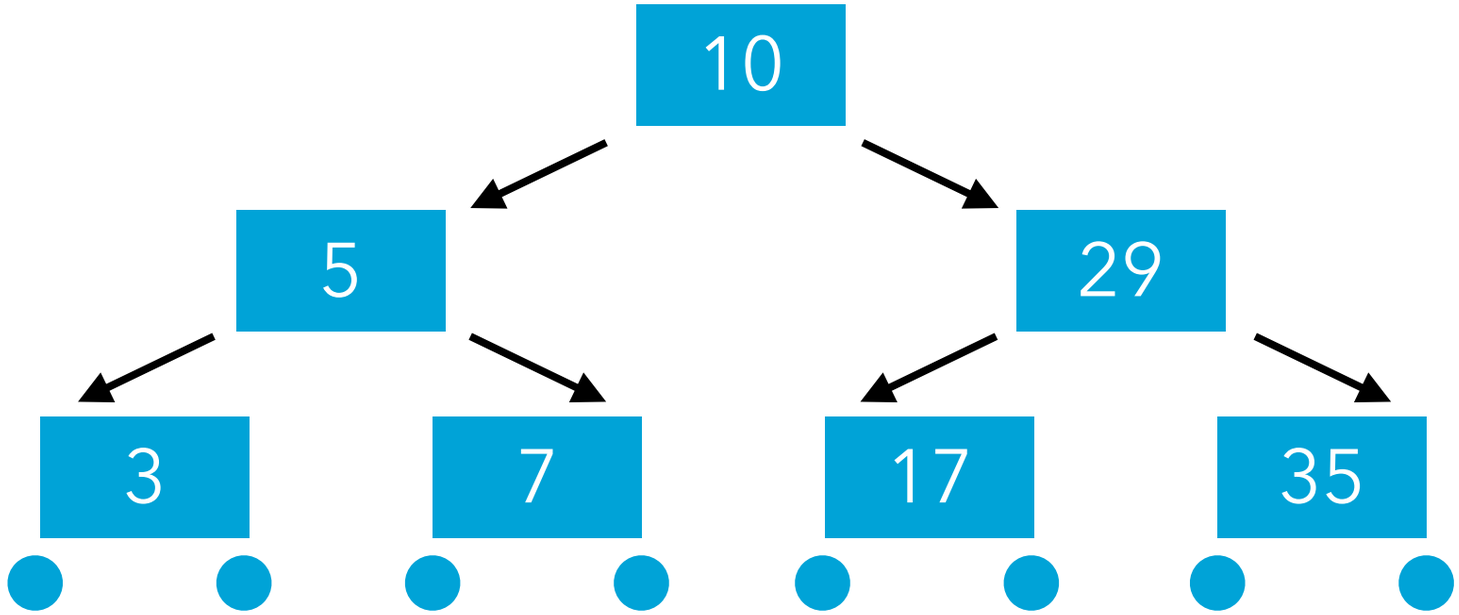
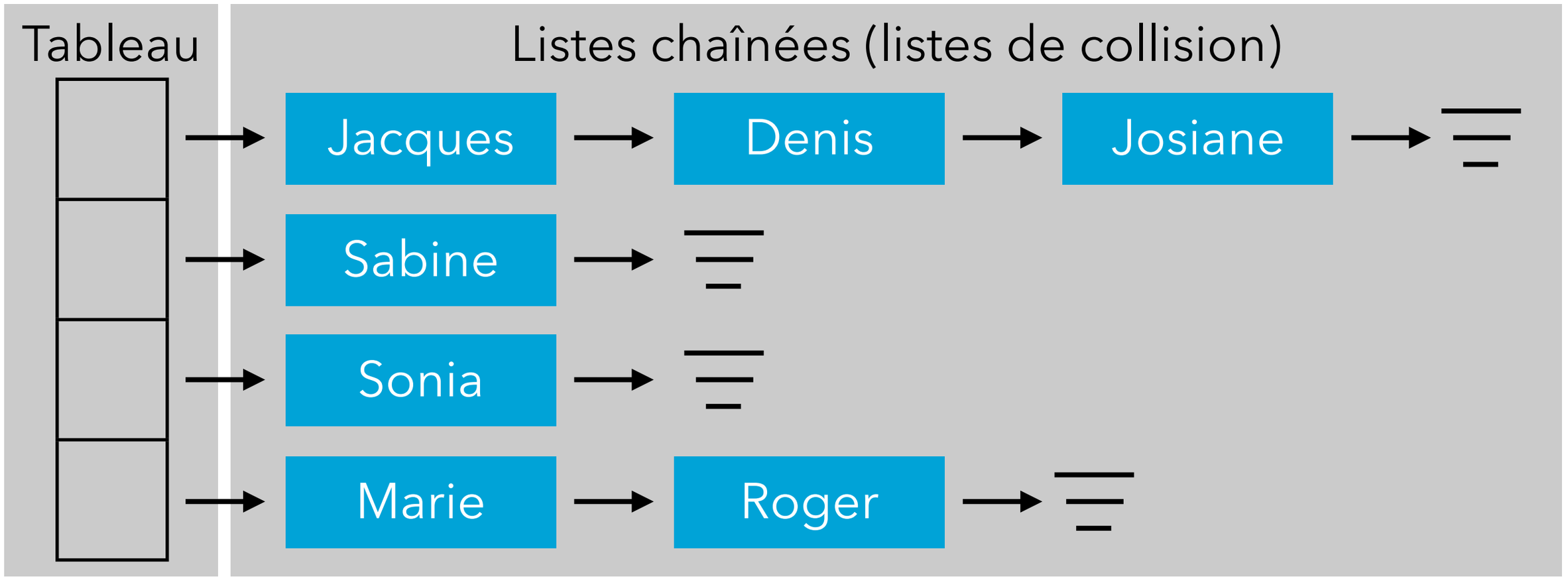


Table de hachage

Dans une table de hachage, la valeur de hachage associée à un élément est utilisée (après réduction !) pour indexer un tableau de listes chaînées.

Si la fonction de hachage est en O(1) et les listes courtes, l'accès est en O(1).



Mise en œuvre : tables assoc.

Les tables associatives peuvent être mise en œuvre au moyen de mêmes techniques que les ensembles.
En effet, une table associative peut être vue comme un ensemble de paires (clef, valeur) dans lequel seules les clefs sont prises en compte pour la comparaison ou le hachage.
(A l'inverse, un ensemble peut être vu comme une table associative dans laquelle les valeurs associées aux clefs sont ignorées).

Généricité

Généricité

La généricité permet de faire abstraction du type des valeurs manipulées.
En rendant une classe ou une interface générique, on peut faire abstraction du type des valeurs manipulées par ses méthodes ou stockées dans ses champs.
En rendant une méthode générique, on peut faire abstraction du type des valeurs qu'elle seule manipule.

Abstraction de valeur

Les fonctions permettent de faire abstraction des *valeurs* manipulées, en les représentant par des variables.

Exemples :

```
int abs(int x) { return (x < 0 ? -x : x); }
int max(int x, int y) {
  return (x > y ? x : y);
}
```

En Java, ces fonctions ne sont pas utilisables telles quelles, c-à-d qu'on ne peut pas écrire simplement `abs` ou `max`. On ne peut que les appliquer à des valeurs pour obtenir une valeur, p.ex. en écrivant `abs(1)` ou `max(v,5)`.

Les langages fonctionnels (p.ex. Scala) suppriment cette restriction. En Java, on la contourne avec le patron *Strategy*.

Abstraction de type

Les classes (ou interfaces) génériques permettent de faire abstraction du *type* des valeurs manipulées, en les représentant par des variables (de type). Exemples :

```
interface List<E> { … }
interface Map<K, V> { … }
```

En Java, ces types ne sont pas utilisables tels quels, c-à-d qu'on ne peut pas écrire simplement `List` ou `Map` (si on ignore les types bruts [*raw types*]). On ne peut que les appliquer à des types pour obtenir un type, p.ex. en écrivant `List<String>` ou `Map<E, Object>`.

Certains langages (p.ex. Scala, Haskell) suppriment cette restriction.

Parallèle valeur / type

valeurs simples

variables (de valeur)

fonctions / méthodes

Univers des ...	
... valeurs	... types
`"hello"`	`Object`
`3.14`	`String`
`x`	`E`
`y`	`K`
`abs`	`List`
`max`	`Map`

types simples

variables de types

fonctions / constructeurs de types

Patrons de conception

Iterator

Le patron *Iterator* permet de parcourir les éléments d'une collection (au sens large) en faisant abstraction de la manière dont les éléments de cette collection sont stockés.

Par exemple, on peut l'utiliser pour parcourir les éléments d'un tableau-liste, d'une liste chaînée, d'un ensemble mis en œuvre par arbre de recherche ou par hachage en écrivant exactement le même code :

```
Iterator<Integer> it = …;
while (it.hasNext()) {
   int nextInteger = it.next();
   …
}
```

Observer

Le patron *Observer* permet à un objet d'être averti du changement d'état d'un autre objet sans que l'observé n'ait connaissance de ses observateurs.
Il est très utilisé dans les bibliothèques d'interface utilisateur pour permettre aux vues (qui affichent les données du modèle) de se mettre à jour dès que le modèle change, sans pour autant que ce dernier n'ait connaissance de ces premières.

MVC

Le patron *MVC* est un patron architectural utile pour décomposer une application dotée d'une interface utilisateur. Il explique comment découpler le **modèle**, qui n'a aucune notion d'interface utilisateur, de la **vue** et du **contrôleur**, chargés respectivement de l'affichage des données et de la gestion de l'interaction avec l'utilisateur.

Le patron *MVC* repose sur plusieurs patrons de conception, principalement *Observer*.

Strategy

Le patron *Strategy* permet de faire abstraction d'un morceau de code. Comme ceux-ci ne sont pas des valeurs en Java, on les encapsule dans des objets qui ne possèdent généralement qu'une seule méthode.
Par exemple, pour écrire une méthode permettant de calculer la valeur maximale d'une liste, on est obligé de faire abstraction du morceau de code qui compare deux éléments entre-eux. Ce morceau de code peut être passé sous la forme d'un comparateur (une stratégie), objet doté d'une seule méthode permettant de comparer deux objets :

```
<T> T max(List<T> l, Comparator<T> c) {…}
```

Decorator, Composite

Le patron *Decorator* permet de changer ou d'augmenter le comportement d'un objet sans changer son type. Cela se fait en « emballant » l'objet en question dans un autre objet du même type, le décorateur, qui laisse l'objet emballé faire le gros du travail mais augmente son comportement au besoin.

Le patron *Composite* permet de composer plusieurs objets d'un type donné en un « macro-objet » de même type qui combine – d'une manière ou d'une autre – le comportement des objets qu'il compose.

Adapter

Le patron *Adapter* permet d'adapter un objet d'un premier type A pour qu'il se comporte comme un objet d'un second type B. Cela se fait en « emballant » l'objet à adapter dans un autre objet d'un type différent, l'adaptateur, qui laisse l'objet emballé faire le gros du travail mais change juste son interface.

Bien entendu, il faut que le comportement des classes A et B soit relativement similaire, sans quoi l’adaptation n’a pas de sens.

Builder

Le patron *Builder* permet de découper le processus de création des instances d'une classe en plusieurs étapes. Le bâtisseur stocke l'état de l'objet en construction.

Ce patron est particulièrement utile pour permettre la création en plusieurs étapes d'instances d'une classe immutable.

Patrons fabriques

Les patrons fabriques (*factory*) permettent de faire abstraction du code qui créé les instances d'une classe donnée.

Le patron *Factory Method* confie à une méthode redéfinissable la création des objets d'un type donné.

Le patron *Abstract Factory* regroupe plusieurs méthodes fabriques qui créent des instances de types liés dans un seul objet.

Adapter / Decorator

Les patrons *Adapter* et *Decorator* sont très proche l'un de l'autre, puisque tous les deux « emballent » un objet.
La différence cruciale est qu'un décorateur a le même type que l'objet qu'il décore, ce qui n'est pas le cas de l'adaptateur.
Dès lors, il est possible d'empiler plusieurs décorateurs de même type, mais pas plusieurs adaptateurs. Par exemple, les différents décorateurs définis sur les flots d'entrée-sortie Java sont fréquemment empilés :

```
new GZIPOutputStream(
  new BufferedOutputStream(
    new FileOutputStream("file.gz")));
```

Decorator / Composite

Le patron *Decorator* peut être vu comme un *Composite* qui ne compose qu'un seul élément. Ces deux patrons sont donc fréquemment utilisés ensemble.

Strategy / Abstract Factory

Les patrons *Strategy* et *Abstract Factory* sont relativement similaires puisque les deux font abstraction d'un morceau de code.

Une fabrique peut être vue comme un cas spécifique de stratégie, spécialisée dans la création d'objets.

Conseils de programmation

Mutabilité

Conseil concernant la mutabilité :

- définissez si possible des classes immutables,
- utilisez une notion d'égalité (et d'ordre) cohérente avec la mutabilité en prenant garde à ne jamais définir une méthode `equals` dépendant d'état mutable,
- si les versions mutables et immutables d'une classe sont utiles, séparez-les en deux en appliquant le patron *Builder*.

Héritage

Conseils concernant l'héritage :

- lorsque vous écrivez une classe, décidez s'il s'agit d'une classe héritable ou instantiable. Rendez-la abstraite dans le premier cas, finale dans le second,
- n'héritez que de classes qui sont clairement conçues pour être héritables, et appliquez le patron *Decorator* dans les autres cas.

Compilation et exécution

Compilation

Le code Java que vous écrivez n'est pas directement compréhensible par l'ordinateur, et ne peut donc être exécuté tel quel.

Dans un premier temps, il est analysé par un compilateur (intégré à Eclipse) qui, en l'absence d'erreurs, le traduit en un ensemble de fichiers classe (extension `.class`).

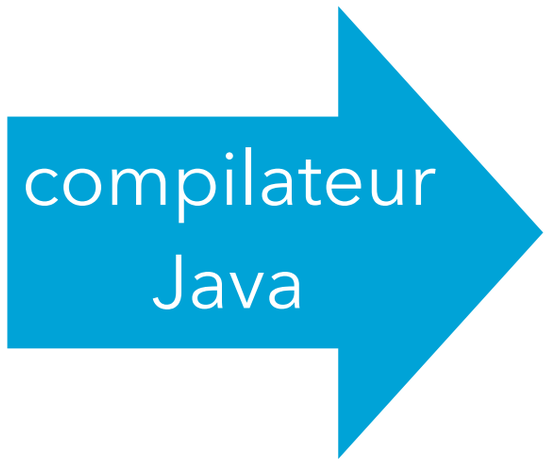
Le code contenu dans ces fichiers classe n'est pas non plus compréhensible par l'ordinateur... Il est donc soit interprété par la machine virtuelle Java, soit compilé par un second compilateur intégré dans celle-ci.

Compilation

Les fichiers classe produits par le compilateur Java (p.ex. intégré à Eclipse) contiennent une version traduite du code source appelée *bytecode*. Ce *bytecode* est une séquence d'instructions pour une machine à pile relativement simple, la **machine virtuelle Java** (*Java Virtual Machine* ou *JVM*).

Fichier `Math.java`

```
class Math {
  static int abs(int x) {
    return (x < 0 ? -x : x);
  }
}
```



Fichier `Math.class`

```
 0: iload_0
 1: ifge 9
 4: iload_0
 5: ineg
 6: goto 10
 9: iload_0
10: ireturn
```

Machine virtuelle Java

Comme son nom l'indique, la machine *virtuelle* Java n'est pas une « machine » (c-à-d un processeur) réelle. Les instructions contenues dans les fichiers classe ne peuvent donc être directement fournies à un processeur réel. Deux solutions existent pour les exécuter :

- **l'interprétation**, qui consiste à les faire exécuter par un programme qui simule le fonctionnement de la machine virtuelle Java,
- **la compilation**, qui consiste à les traduire en code machine pour un processeur réel, p.ex. de la famille Intel x86.

Machine virtuelle Java

Sur les ordinateurs de bureau, les deux techniques (interprétation et compilation) sont généralement utilisées de manière complémentaire :

- initialement, le contenu des fichiers classe est interprété, ce qui est (relativement) lent mais peut démarrer immédiatement,
- par la suite, le code qui est fréquemment exécuté est identifié, compilé à la volée puis fourni au processeur de l'ordinateur pour exécution.

Le programme qui se charge d'effectuer cela est généralement appelé – par abus de langage – la machine virtuelle Java. Il s'agit en réalité d'une *mise en œuvre* de cette machine.

Interprétation

Exemple d'interprétation de l'appel de méthode `Math.abs(-12)`.

Fichier `Math.class`

```
 0: iload_0
 1: ifge 9
 4: iload_0
 5: ineg
 6: goto 10
 9: iload_0
10: ireturn
```

PC: compteur de programme (*Program Counter*)

Interprétation

Exemple d'interprétation de l'appel de méthode `Math.abs(-12)`.

Fichier `Math.class`

```
 0: iload_0
 1: ifge 9
 4: iload_0
 5: ineg
 6: goto 10
 9: iload_0
10: ireturn
```

PC	Pile
0	[]

PC: compteur de programme (*Program Counter*)

Interprétation

Exemple d'interprétation de l'appel de méthode `Math.abs(-12)`.

Fichier `Math.class`

```
 0: iload_0
 1: ifge 9
 4: iload_0
 5: ineg
 6: goto 10
 9: iload_0
10: ireturn
```

PC	Pile
0	[]
1	[-12]

PC: compteur de programme (*Program Counter*)

Interprétation

Exemple d'interprétation de l'appel de méthode `Math.abs(-12)`.

Fichier `Math.class`

```
 0: iload_0
 1: ifge 9
 4: iload_0
 5: ineg
 6: goto 10
 9: iload_0
10: ireturn
```

PC	Pile
0	[]
1	[-12]
4	[]

PC: compteur de programme (*Program Counter*)

Interprétation

Exemple d'interprétation de l'appel de méthode `Math.abs(-12)`.

Fichier `Math.class`

```
 0: iload_0
 1: ifge 9
 4: iload_0
 5: ineg
 6: goto 10
 9: iload_0
10: ireturn
```

PC	Pile
0	[]
1	[-12]
4	[]
5	[-12]

PC: compteur de programme (*Program Counter*)

Interprétation

Exemple d'interprétation de l'appel de méthode `Math.abs(-12)`.

Fichier `Math.class`

```
 0: iload_0
 1: ifge 9
 4: iload_0
 5: ineg
 6: goto 10
 9: iload_0
10: ireturn
```

PC	Pile
0	[]
1	[-12]
4	[]
5	[-12]
6	[12]

PC: compteur de programme (*Program Counter*)

Interprétation

Exemple d'interprétation de l'appel de méthode `Math.abs(-12)`.

Fichier `Math.class`

```
 0: iload_0
 1: ifge 9
 4: iload_0
 5: ineg
 6: goto 10
 9: iload_0
10: ireturn
```

PC	Pile
0	[]
1	[-12]
4	[]
5	[-12]
6	[12]
10	[]

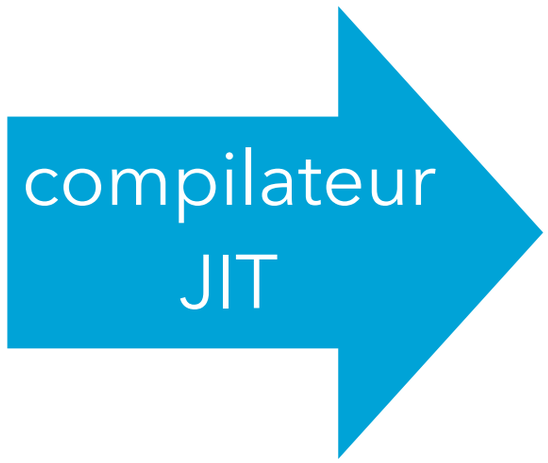
PC: compteur de programme (*Program Counter*)

Compilation JIT

Les méthodes qui sont exécutées très souvent sont compilées à la volée (on dit aussi dynamiquement) par un compilateur intégré à la machine virtuelle et appelé *Just In Time (JIT) compiler*.

Fichier `Math.class`

```
 0: iload_0
 1: ifge 9
 4: iload_0
 5: ineg
 6: goto 10
 9: iload_0
10: ireturn
```



Code Intel x86

```
_abs:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, %eax
        negl    %eax
        cmovll  %edi, %eax
        popq    %rbp
        ret
```

Gestion mémoire

Une caractéristique importante de la machine virtuelle Java est qu'elle gère automatiquement la mémoire.
Cela signifie que la mémoire allouée aux objets créés au moyen de l'opérateur new de Java ne doit – et ne peut – pas être libérée explicitement par le programmeur.
Au lieu de cela, la mémoire allouée aux objets qui sont devenus inatteignables – c-à-d plus référencés – est automatiquement libérée.
La partie de la machine virtuelle Java qui se charge de cette tâche est nommée le **ramasse-miettes** (*garbage collector* ou *GC*).

Ramasse-miettes

La boucle ci-dessous consomme vite la totalité de la mémoire disponible, et l'exécution s'arrête avec une exception `OutOfMemoryError` :

```
int[][] arrays = new int[1000][];
for (int i = 0; i < 1000; ++i) {
  arrays[i] = new int[1000000];
}
```

En ajoutant simplement la ligne suivante :

```
arrays[i] = null;
```

comme seconde ligne de la boucle, le programme s'exécute sans problème puisqu'aucune référence n'est gardée sur les tableaux alloués, qui peuvent donc être libérés par le ramasse-miettes.

Ramasse-miettes

Le travail du ramasse-miettes peut être observé en passant l'option `-verbose:gc` à la machine virtuelle. La première version de la boucle produit ceci :

```
[GC 16349K->16041K(69056K), 0,00653 secs]
[GC 32579K->31666K(87168K), 0,00819 secs]
[GC 67441K->66774K(107008K), 0,01315 secs]
…
[Full GC 984715K->984662K(991168K), …]
[Full GC 984662K->984648K(991168K), …]
Exception in thread "main"
java.lang.OutOfMemoryError: Java heap space
```

Ramasse-miettes

Tandis que la seconde produit ceci :

```
[GC 16347K->368K(68992K), 0,0017120 secs]
[GC 16903K->400K(87040K), 0,0009230 secs]
[GC 36173K->368K(87040K), 0,0009550 secs]
[GC 35687K->352K(123136K), 0,0010370 secs]
[GC 70878K->384K(123136K), 0,0010970 secs]
[GC 70837K->336K(192832K), 0,0009030 secs]
[GC 141146K->308K(192832K), 0,0013020 secs]
[GC 141055K->308K(337216K), 0,0004210 secs]
…
[GC 379227K->308K(431040K), 0,0002650 secs]
[GC 379223K->308K(430528K), 0,0005050 secs]
```

et le programme se termine normalement.