Programmation Avancée et Application

Introduction à Java

Jean-Guy Mailly

jean-guy.mailly@u-paris.fr

LIPADE - Université de Paris
http://www.math-info.univ-paris5.fr/∽jmailly/

Introduction à Java

1. Au sujet du module

2. Les bases de Java

3. Types, variables, tableaux

Au sujet du module

Au sujet du module

- Jean-Guy Mailly : jean-guy.mailly@u-paris.fr, Bureau 814 I
- 18h de cours : lundi, 11h15–12h45, Polonovski
- 36h de TD : mercredi, 8h30–11h30, Cordier 523A (J. Delobelle)
  mercredi, 15h45–18h45, Fourier A526 (J.-G. Mailly)
  jeudi, 14h00–17h00, Fourier D529 (J.-G. Mailly)
  jeudi, 17h00–20h00, Cordier 523A (J. Delobelle)

Au sujet du module

- Jean-Guy Mailly : jean-guy.mailly@u-paris.fr, Bureau 814 I
- 18h de cours : lundi, 11h15–12h45, Polonovski
- 36h de TD : mercredi, 8h30–11h30, Cordier 523A (J. Delobelle)
  mercredi, 15h45–18h45, Fourier A526 (J.-G. Mailly)
  jeudi, 14h00–17h00, Fourier D529 (J.-G. Mailly)
  jeudi, 17h00–20h00, Cordier 523A (J. Delobelle)
- Modalités de contrôle de connaissances :
  - Contrôle continu : un contrôle durant le semestre (CC), un projet (P) et un contrôle terminal (CT)
  - Note finale : $\frac{CC}{4} + \frac{P}{4} + \frac{CT}{2}$

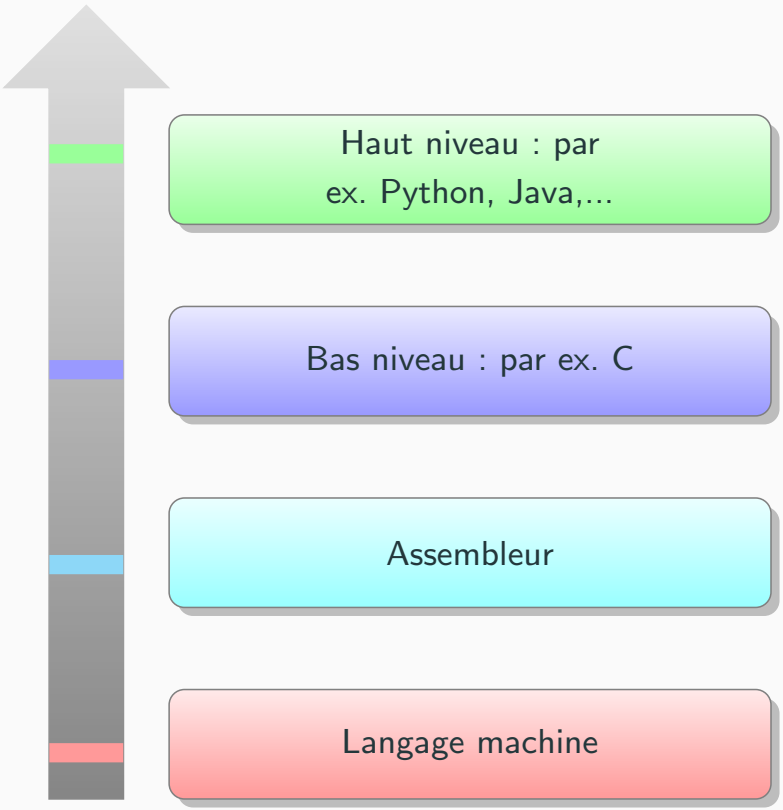
Au sujet du module

- Jean-Guy Mailly : jean-guy.mailly@u-paris.fr, Bureau 814 I
- 18h de cours : lundi, 11h15–12h45, Polonovski
- 36h de TD : mercredi, 8h30–11h30, Cordier 523A (J. Delobelle)
  mercredi, 15h45–18h45, Fourier A526 (J.-G. Mailly)
  jeudi, 14h00–17h00, Fourier D529 (J.-G. Mailly)
  jeudi, 17h00–20h00, Cordier 523A (J. Delobelle)
- Modalités de contrôle de connaissances :
  - Contrôle continu : un contrôle durant le semestre (CC), un projet (P) et un contrôle terminal (CT)
  - Note finale : $\frac{CC}{4} + \frac{P}{4} + \frac{CT}{2}$
- Moodle : **IF05X030 Programmation Avancée et Application**

  `https://moodle.u-paris.fr/course/view.php?id=12140`

Les bases de Java

Différents niveaux de langages

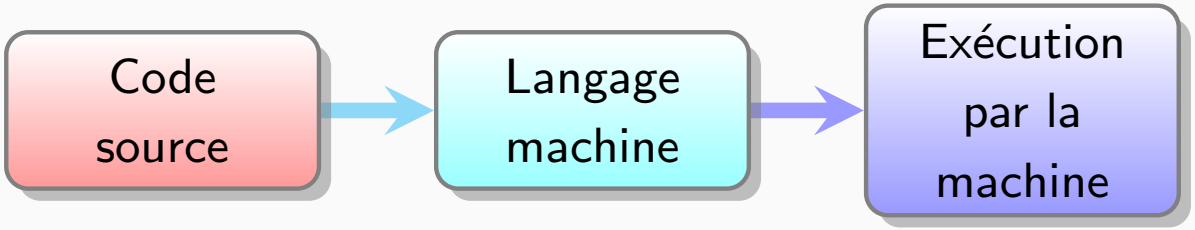


- Langage machine : séquence de 0 et de 1
- Assembleur : représentation du langage machine lisible pour un humain
- Bas niveau : proche de la machine, besoin de gérer soi-même la mémoire, etc
- Haut niveau : le programmeur se concentre sur le problème à résoudre plutôt que sur la machine

Avantages de Java

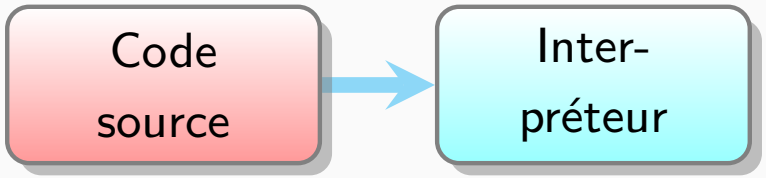
- Langage orienté objet
- Syntaxe « à la C » : similaire à C, C++, C#, Javascript, PHP
- Mécanismes complexes pris en charge :
  - Gestion automatisée de la mémoire
  - Sérialisation
  - Exceptions
- Librairie standard très développée (interfaces graphiques, réseau, bases de données,...)
- Portabilité
- À la base du développement d'applications Android

Rappel : langage compilé



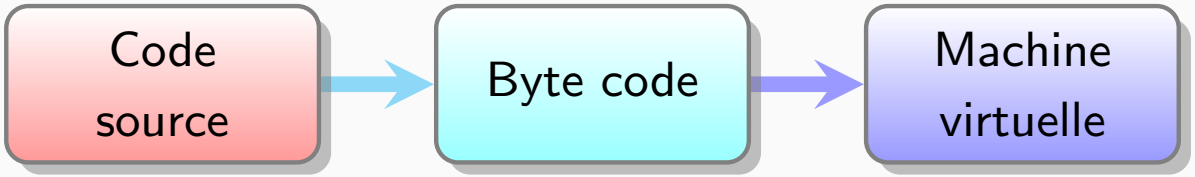
- La compilation est la transformation d'un code source en langage machine
- Le fichier produit par la compilation est directement exécuté par le système exploitation
- Exemple de langages compilés : C, C++, Cobol, Fortran, Pascal
- Avantage : généralement l'exécutable est optimisé pour le système d'exploitation
- Inconvénient : il faut compiler sur chaque système

Rappel : langage interprété



- L'interpréteur exécute directement le programme sans produire de code machine
- Exemple de langages interprétés : Bash, Python, Prolog
- Avantage : facilement portable, il suffit d'avoir un interpréteur disponible pour le système d'exploitation voulu
- Inconvénient : généralement moins efficace (pas d'optimisation)

Java : un langage intermédiaire



- Le code Java est compilé en byte code, un langage non compréhensible par le programmeur, mais différent du langage machine
- Le byte code n'est pas directement exécutable par le système d'exploitation : besoin d'une machine virtuelle
- Portabilité : il existe des machines virtuelles pour de nombreux systèmes d'exploitation ; le même byte code peut donc être utilisé directement sur différentes machines

Versions de Java

- Java est régulièrement mis à jour.
- Version utilisée dans ce cours : Java SE 12 (sortie en mars 2019) [1]
- Distribution de Java SE sous deux formes :
  - Java Runtime Environment (JRE) : machine virtuelle + bibliothèque standard, nécessaire pour exécuter un programme Java
  - Java Development Kit (JDK) : JRE + compilateur, nécessaire pour développer un programme Java
- Bibliothèque standard = API standard (application programming interface)
- Machine virtuelle = JVM (Java Virtual Machine)

1. Pour info, Java 16 est sortie en mars 2021. Nous utiliserons la version disponible à l'Université.

Qu'est-ce qu'un programme Java ?

Concrètement, un programme écrit en Java est un ensemble de classes :

- des classes fournies par l'API standard
- des classes fournies par d'autres API
- des classes écrites par le développeur du programme

Qu'est-ce qu'une classe ? (grossièrement)

- structure de données
- fonctions pour manipuler ces données

Comment distribuer un programme Java ?

- Si une JVM est disponible pour un système d'exploitation, alors elle est forcément associée aux classes de l'API standard
- Pour distribuer un programme Java, il suffit donc de distribuer les classes écrites par le développeur (et éventuellement les classes des API tierces)
- Deux méthodes
  - Distribution des fichiers compilés en byte code (fichier `.class`)
  - Distribution d'une archive `jar` qui contient les classes

Comment distribuer un programme Java ?

- Si une JVM est disponible pour un système d'exploitation, alors elle est forcément associée aux classes de l'API standard
- Pour distribuer un programme Java, il suffit donc de distribuer les classes écrites par le développeur (et éventuellement les classes des API tierces)
- Deux méthodes
  - Distribution des fichiers compilés en byte code (fichier `.class`) ✘
  - Distribution d'une archive `jar` qui contient les classes ✔

Compilation d'une classe Java

On suppose ici que le terminal est situé dans le répertoire où se situe le fichier `ClasseA.java` ; sinon :

```
$ cd /chemin/vers/le/repertoire/du/code
$ ls
ClasseA.java
```

Commande de base pour compiler une classe Java :

```
$ javac ClasseA.java
$ ls
ClasseA.class ClasseA.java
```

Exécution d'une classe Java

La classe `ClasseA` décrit un programme qui se contente d'afficher un message. Après la compilation, on l'exécute via :

```
$ java ClasseA
Hello, World!
```

⚠La commande `java` s'appelle avec le nom de la classe, pas le nom d'un fichier (ici, juste `ClasseA` au lieu de `ClasseA.class`). Dans le cas contraire :

```
$ java ClasseA.class
Erreur : impossible de trouver ou de charger la classe principale ClasseA.class
Cause par : java.lang.ClassNotFoundException: ClasseA.class
```

Compilation de plusieurs classes Java

On a maintenant plusieurs classes :

```
$ ls
ClasseA.java  ClasseB.java
```

Compilation de plusieurs classes :

```
$ javac *.java
$ ls
ClasseA.class  ClasseA.java  ClasseB.class  ClasseB.java
```

Arborescence des fichiers

Un projet Java est normalement subdivisé en plusieurs répertoires :

- Un répertoire pour les fichiers sources `.java`
- Un répertoire pour les fichiers compilés `.class`
- D'autres répertoires peuvent apparaître (on en reparlera plus tard...)

Dans notre exemple, on doit avoir :

```
repertoirePrincipal
├── src/
│   ├── ClasseA.java
│   └── ClasseB.java
└── bin/
    ├── ClasseA.class
    └── ClasseB.class
```

Compilation d'un projet

- Plutôt que de compiler dans le répertoire `src`, puis de déplacer les fichiers `.class` dans `bin`, on demande au compilateur de le faire
- Depuis le répertoire principal :

```
$ ls
bin/ src/
$ javac -d bin src/*.java
$ ls -R
bin/ src/
./bin:
ClasseA.class   ClasseB.class
./src:
ClasseA.java   ClasseB.java
```

L'option `-d` de la commande `javac` permet d'indiquer le répertoire (en anglais *directory*) qui doit recevoir les fichiers compilés.

Compilation d'un projet avec classes externes (1/2)

- On peut également indiquer au compilateur qu'il doit utiliser des classes prédéfinies
- Par exemple, si `ClassA` a besoin de classes qui sont dans le répertoire `lib/` :

```
$ ls
bin/ lib/ src/
$ javac -d bin -classpath lib src/*.java
```

On peut utiliser la forme raccourcie de l'option :

```
$ javac -d bin -cp lib src/*.java
```

Compilation d'un projet avec classes externes (2/2)

Dans le cas où il y a plusieurs sources externes, on utilise deux points (`:`) pour les séparer :

```
$ ls
bin/ lib/ src/
$ javac -d bin -cp lib1:lib2:lib3 src/*.java
```

On peut également utiliser une archive `.jar` :

```
$ javac -d bin -cp lib:MonArchive.jar src/*.java
```

Exécution d'un projet avec classes externes

Si toutes les classes ne sont pas dans le répertoire courant, il faut également indiquer à la commande `java` quel est le `classpath` :

```
$ ls -R
bin/ lib/ src/
./bin:
ClasseA.class  ClasseB.class
./lib:
ClasseC.class
./src:
ClasseA.java  ClasseB.java
$ java -cp bin:lib ClasseA
Hello, World!
Utilisation de la ClasseC.
```

Remarque : les classes de l'API standard sont automatiquement dans le `classepath`, pas besoin de les indiquer

Hello, World !

```
public class Hello {
  public static void main(String[]){
    System.out.println("Hello, World!");
  }
}
```

- Nous reviendrons en détail sur les mots-clés par la suite
- `System.out.println` permet d'afficher un message sur la sortie standard.
- Ce programme définit une classe qui s'appelle `Hello`. Elle doit être définie dans un fichier `Hello.java`, dont la compilation provoque la création d'un fichier `Hello.class`

La méthode main

Tout programme Java a un unique point d'entrée, qui est la méthode

```
public static void main(String[] args){
   ...
}
```

Sans rentrer dans les détails pour l'instant,

- `String` est une classe de l'API qui permet de représenter des chaînes de caractères
- `String[] args` est donc un tableau de `String` représentant les paramètres du programme sur la ligne de commande
  - Plus de détails sur l'utilisation des tableaux plus tard
- `void` : la méthode ne renvoie aucune valeur
- `public` et `static` : plus tard...

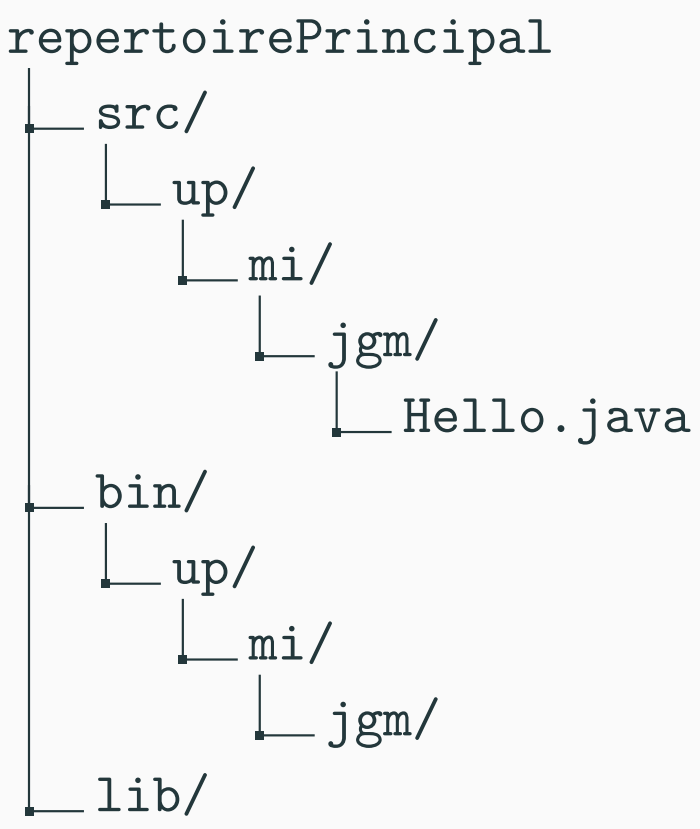
Organisation du projet en packages (1/2)

- Le code source d'un programme Java est découpé en packages
- Chaque package réunit un ensemble de classes qui ont un lien « logique » entre elles
- Par exemple :
  - Un package pour l'interface graphique
  - Un package pour la gestion de la base de données
  - Un package pour l'intelligence artificielle
  - ...
- Cela se manifeste par la création de répertoires équivalents aux packages

```
repertoirePrincipal
├── src/
│   ├── gui
│   ├── database
│   └── ai
└── bin/
```

Organisation du projet en packages (2/2)

- Les fichiers dans lesquels sont définies les classes sont placés dans les répertoires correspondant aux packages
- Une instruction en début de fichier indique le package auquel appartient la classe




```
package up.mi.jgm ;

public class Hello {
    ...
}
```

Compilation du projet avec packages

Depuis le répertoire principal :

```
javac -d bin/up/mi/jgm/
      -cp src:lib
       src/up/mi/jgm/*.java
```

Compilation du projet avec packages

Depuis le répertoire principal :

```
javac -d bin/up/mi/jgm/
      -cp src:lib
       src/up/mi/jgm/*.java
```

Ça peut être très fastidieux de procéder ainsi s'il y a de nombreux packages. La compilation de projets complexes peut cependant être automatisée via un `Makefile`, ou d'autres techniques (Maven, ant).

Compilation du projet avec packages

Depuis le répertoire principal :

```
javac -d bin/up/mi/jgm/
      -cp src:lib
       src/up/mi/jgm/*.java
```

Ça peut être très fastidieux de procéder ainsi s'il y a de nombreux packages. La compilation de projets complexes peut cependant être automatisée via un `Makefile`, ou d'autres techniques (Maven, ant).

Dans notre cas, l'utilisation d'Eclipse permettra de simplifier la compilation et l'exécution.

Exécution du projet avec packages

Depuis le répertoire principal :

```
java -cp bin/up/mi/jgm/:lib/ up.mi.jgm.Hello
```

Convention pour les noms de packages

- Les packages `java` et `javax` sont (par convention) réservés à l'API standard
- Par convention, l'arborescence des packages d'un projet doit correspondre au nom de domaine du site web associé au projet, dans l'ordre inversé
  - Par exemple, un projet développé par la société dont le site web est `exemple.fr` aura pour racine le package `fr.exemple.monprojet`
  - Un projet déposé sur `sourceforge.net` aura pour racine `net.sf.monprojet`
- Pour les exemples du cours, j'utiliserai `up.mi.jgm`, et je vous invite à respecter la même convention pour vos projets

Nom complet d'une classe

- Le nom complet d'une classe contient la spécification détaillée du package auquel elle appartient
  par ex. : `up.mi.jgm.Hello`
- Il est normalement nécessaire de le préciser lors de l'utilisation de la classe :

```
public static void main(String[] args){
  up.mi.jgm.A monObjet = new up.mi.jgm.A();
}
```

Import d'une classe

- Le mot clé `import` permet de préciser une unique fois le nom complet d'une classe

```
import up.mi.jgm.A ;

public static void main(String[] args){
  A monObjet = new A();
}
```

- Une fois que l'`import` est fait, l'utilisation du nom simple `A` fera toujours référence à la même classe `up.mi.jgm.A`
- Il n'est pas nécessaire d'importer les classes du package courant, ni celles du package `java.lang` de l'API standard (classes de base)

Classes homonymes

- Plusieurs classes définies dans différents packages peuvent avoir le même nom
- Une seule (au plus) peut être importée
- Il faut alors utiliser les noms complets

```
import up.mi.jgm.A ;

public static void main(String[] args){
  A monObjet = new A();
  mon.autre.classe.A autre = new mon.autre.classe.A();
}
```

Attention aux classes homonymes !

- Eclipse permet d'importer automatique des classes qui sont utilisées dans votre code
- S'il y a une homonymie, il vous demande de choisir laquelle est la bonne
- ⚠Si vous n'êtes pas attentifs, vous risquez d'importer la mauvaise classe, et d'avoir un code incorrect sans vous en rendre compte !
- Par ex., de nombreuses classes de JavaFX portent le même nom que des classes d'AWT (une API dédiée aux interfaces graphiques que nous n'utiliserons pas dans ce cours)

Compilation et utilisation d'archives jar

- Une archive `jar` est un conteneur dédié au langage Java
- Elle peut être utilisée pour transmettre des classes, sous forme de fichiers `.class` et/ou `.java`
- Un `jar` peut aussi servir à transmettre un programme fonctionnel (on parle de `jar` exécutable, ou *runnable* `jar` *file* en anglais)
- Création de l'archive :
  `jar cf JARFILE INPUTFILES`
- Liste du contenu de l'archive :
  `jar tf JARFILE INPUTFILES`
- Extraction du contenu de l'archive :
  `jar xf JARFILE INPUTFILES`
- Ne pas oublier d'ajouter le `jar` au classpath avec l'option `-cp`

Compilation et utilisation d'un jar exécutable

- On peut rendre un jar exécutable en indiquant la classe principale du programme (celle qui contient la méthode `main`)
- Création de l'archive :
  `jar cvfe JARFILE MAINCLASS INPUTFILES`
- Exécution de l'archive :
  `java -jar JARFILE`

Utilisation d'un IDE

- La plupart des démarches que nous avons présentées peuvent être simplifiées grâce à l'utilisation d'un environnement de développement intégré (EDI, ou IDE en anglais) qui comprend
  - L'éditeur de texte « intelligent »
  - Le compilateur
  - Le créateur de `jar`
  - Une interface simplifiée pour la gestion de packages
  - Des outils de débugage
  - Plein d'autres choses...
- Nous verrons en TP comment utiliser Eclipse

Types, variables, tableaux

Types primitifs

En plus de types d'objets (nous y reviendrons plus tard), Java propose des types de données primitifs, pour représenter

- les nombres entiers
- les nombres décimaux
- les caractères
- les booléens

Types primitifs : les entiers

Type	Valeur min	Valeur max	Taille mémoire
`byte`	$-2^7 = -128$	$2^7 - 1 = 127$	1 octet
`short`	$-2^{15} = -32768$	$2^{15} - 1 = 32767$	2 octets
`int`	$-2^{31} \simeq -2 \times 10^9$	$2^{31} - 1 \simeq 2 \times 10^9$	4 octets
`long`	$-2^{63} \simeq 9 \times 10^{18}$	$2^{63} - 1 \simeq 9 \times 10^{18}$	8 octets

Types primitifs : les décimaux

- Les nombres décimaux sont représentés sous la forme $x = s \times m \times 2^e$ où
  - s est le signe, représenté par un bit
  - m est la mantisse
  - e est l'exposant

Type	m	e	Taille mémoire
`float`	23 bits	8 bits	4 octets
`double`	52 bits	11 bits	8 octets

Types primitifs : les booléens

Deux valeurs de vérité constantes `boolean`

- `true`
- `false`

Remarque : l'espace mémoire utilisé dépend de la JVM

Types primitifs : les booléens

Deux valeurs de vérité constantes `boolean`

- `true`
- `false`

Remarque : l'espace mémoire utilisé dépend de la JVM

⚠Pas de conversion automatique des entiers en booléens (contrairement au langage C)

Types primitifs : les caractères

- type `char` : utilise le standard unicode sur 2 octets
- de 0 (`\u0000`) à 127 (`\u007f`) : identique aux codes ASCII
- de 128 (`\u0080`) à 255 (`\uffff`) : codes Latin-1

Caractères spéciaux

Retour à la ligne	`\n`
Tabulation	`\t`
Apostrophe	`\'`
Double apostrophe	`\"`
Backslash	`\\`
Caractère Unicode	`\uxxxx`

Déclaration de variables

Syntaxe générale : `type nomVariable ;`

- `float x ;`
- `int n ;`
- `boolean b ;`
- `char c ;`

Ces variables sont déclarées mais ne sont pas affectées : aucune valeur définie n'est stockée dans la mémoire allouée à ces variables

Affectation de variables

Affectation en deux temps :

```
float x ;
x = 5.2 ;
```

Affectation de variables

Affectation en deux temps :

```
float x ;
x = 5.2 ;
```

Affectation lors de la déclaration :

```
float x = 5.2 :
int n = 5 ;
float y = n ;
```

La dernière instruction fait une conversion automatique de l'entier 5 vers le décimal 5.0

Déclarations et affectations multiples

- On peut déclarer et affecter plusieurs variables de même type en même temps :
  - `int a, b, c ;`
  - `int a = 2, b = 3, c = 4 ;`

Conversions implicites

Une conversion automatique est faite lorsqu'on passe d'un type « plus particulier » à un type « plus général »

1. `byte`
2. `short`
3. `int`
4. `long`
5. `float`
6. `double`

Exemples :

```
short s = 6 ; int i = 8 ;
double d = s ; // conversion implicite de 6 en 6.0
long l = s * i ; // convertit s et i en long pour
                 // faire la multiplication
```

Conversions implicites

Une conversion automatique est faite lorsqu'on passe d'un type « plus particulier » à un type « plus général »

1. `byte`
2. `short`
3. `int`
4. `long`
5. `float`
6. `double`

Exemples :

```
short s = 6 ; int i = 8 ;
double d = s ; // conversion implicite de 6 en 6.0
long l = s * i ; // convertit s et i en long pour
                 // faire la multiplication
```

Conversion implicite de `char` vers `int`, `long`, `double` et `float`

Conversions explicites

Si la conversion implicite est impossible, on utilise la syntaxe `type var1 = (type) var2` où `type` est le type de la variable `var1`

Exemples :

```
double d = -4.3 ;
float f = (float) d ;
int i = (int) d ; // i vaut -4
```

Conversions explicites

Si la conversion implicite est impossible, on utilise la syntaxe `type var1 = (type) var2` où `type` est le type de la variable `var1`

Exemples :

```
double d = -4.3 ;
float f = (float) d ;
int i = (int) d ; // i vaut -4
```

⚠Pas de conversion entre nombres et booléens