Le Langage Java

M. BASTREGHI, P. BETTENS, M. CODUTTI, B. DRION, C. LERUSTE, R. FISSET

Haute Ecole de Bruxelles - Ecole Supérieure d'Informatique

1ère année

Année académique 2003-2004

Professeurs

- M. Bastreghi (MBA) mbastreghi@heb.be
- P. Bettens (PBT) pbettens@heb.be
- M. Codutti (MCD) mcodutti@heb.be
- B. Drion (BDR) bdrion@heb.be
- C. Leruste (CLR) cleruste@heb.be
- R. Fisset (RFS) roger.fisset@wanadoo.be

Table des matières

Table des matières

Table des matières

Table des matières

Présentation

- Les langages de programmation
- **Java** comme langage d'apprentissage
- Les supports du cours
- L'évaluation

Définitions

Langage : "Ensemble de caractères, de symboles et de règles permettant de les assembler, utilisé pour donner des **instructions** à l'ordinateur" (Larousse)

Programme : "Séquence d'**instructions** et de données enregistrées sur un support et susceptible d'être traitée par un ordinateur" (Larousse)

Historique des langages

- Langage machine : ensemble de 0 et de 1
- Langage d'assemblage : abstraction des instructions
- Langage de haut niveau (60') : abstraction des expressions (ex : **`COBOL`**, **`FORTRAN`**)
- Langage structuré (70') : abstraction des structures de contrôle (ex : **`Pascal`**, **`C`**)
- Langage orienté objet (90') : abstraction des données (ex : **`Eiffel`**, **`SmallTalk`**, **`Java`**, **`C++`**)

Historique des langages

- On a aussi :
  - Les langages logiques : **`Prolog`**, . . .
  - Les langages fonctionnels : **`Lisp`**, **`Scheme`**, . . .
- **Une classe de langage est adaptée à une classe de problèmes . . . et ces problèmes évoluent dans le temps . . .**

Les langages orientés objets

- Restés longtemps dans les laboratoires (**`Simula`** en 67, **`Eiffel`**, . . .)
  - sérieux problèmes d'efficacité à l'exécution
  - indigence des environnements de dévelopements
  - rupture avec les méthodes d'analyses, . . .
- Ont explosé il y a une dizaine d'années comme possible réponse à la croissance exponentielle de la complexité des applications
  - réutilisabilité
  - lisibilité

Les langages orientés objets

- Le programme est écrit dans les termes du problème (les objets, regroupés en classes)
- Résolument tournés et associés au processus d'analyse
- Ex : Client, Chambre, Réservation

Tendance actuelle

- Prédominance des langages OO
- Intégration dans un processus de développement Orienté Objet
  - mapping quasi immédiat des résultats de l'analyse (en UML, . . .) vers le langage de programmation

Tendance actuelle

- Mise en évidence accrue
  - de l'importance des bibliothèques (API) : partage de code, réutilisabilité, robustesse
  - de l'architecture et des *templates* de programmations
  - de la gestion temporelle d'une application : maintenance et versions, évolution, documentation, . . .

Les cours de Langage en première

A l'ESI : Historique

- Au début : **PL1**
- 1988 : La nouvelle section industrielle adopte le **C**
- 1998 : Le **C** se généralise à toutes les sections
- 2004 : **Java**
- Et ce ne sera certainement pas le dernier . . .

Ailleurs

- Paul Lambin est passé à **Java** il y a peu
- L'**ULB** est toujours au **C** (**C++**)

Position de Java dans le monde professionnel

Au niveau des **applications locales** :

- Présence (encore ?) faible :
  - Plus lent que le **C/C++**
  - Sécurité, portabilité peu pertinentes
  - Secteur en perte de vitesse
- Exceptions : **IDE Java** sont souvent écrites en **Java** (**Eclipse**, **NetBeans**, **JBuilder**, ...)

Position de Java dans le monde professionnel

Au niveau des **applications Web** :

- Côté Client :
  - **`JavaScript`** n'est pas **`Java`** !
  - *Applets* **`Java`** permettent une extensions des possibilités des navigateurs
- Côté Serveur :
  - Des technologies **`Java`** (**`JSP`** et **`Servlets`**) permettent de générer des pages **`html`** dynamiques (semblable à **`PHP`**)

Position de Java dans le monde professionnel

Au niveau des **applications client-serveur** :

- Secteur de plus en plus important !
- Présence très forte :
  - Architectures nombreuses et abouties
  - Sécurité, portabilité
- Architecture **`Java 2`**, Entreprise Edition (**`j2ee`**)
  - jusqu'il y a peu pratiquement incontournable
  - exemple : **`amazon.com`**
- Concurrence nouvelle de **`Microsoft`** :
  Architecture **`.NET`**

Pourquoi Java ?

- Il a de réelles **qualités pédagogiques**
  (au contraire du **C** et du **C++**)
  - **syntaxe claire et précise**
  - **typage fort**
  - mécanisme de **détection précoce des erreurs**
  - accès simple à de **nombreux concepts fondamentaux** de la programmation moderne : multithreading, mécanisme d'exception, sérialisation, applet, réseau, graphisme, documentation élaborée, . . .

Pourquoi Java ?

- **Economie d'échelle** : pour les autres cours qui peuvent se baser sur la connaissance d'un langage orienté objet moderne
- De plus
  - Il est de la **dernière génération** (au contraire du **C**)
  - Il a du **succès** auprès des entreprises (autant sinon plus que le **C** et le **C++**)

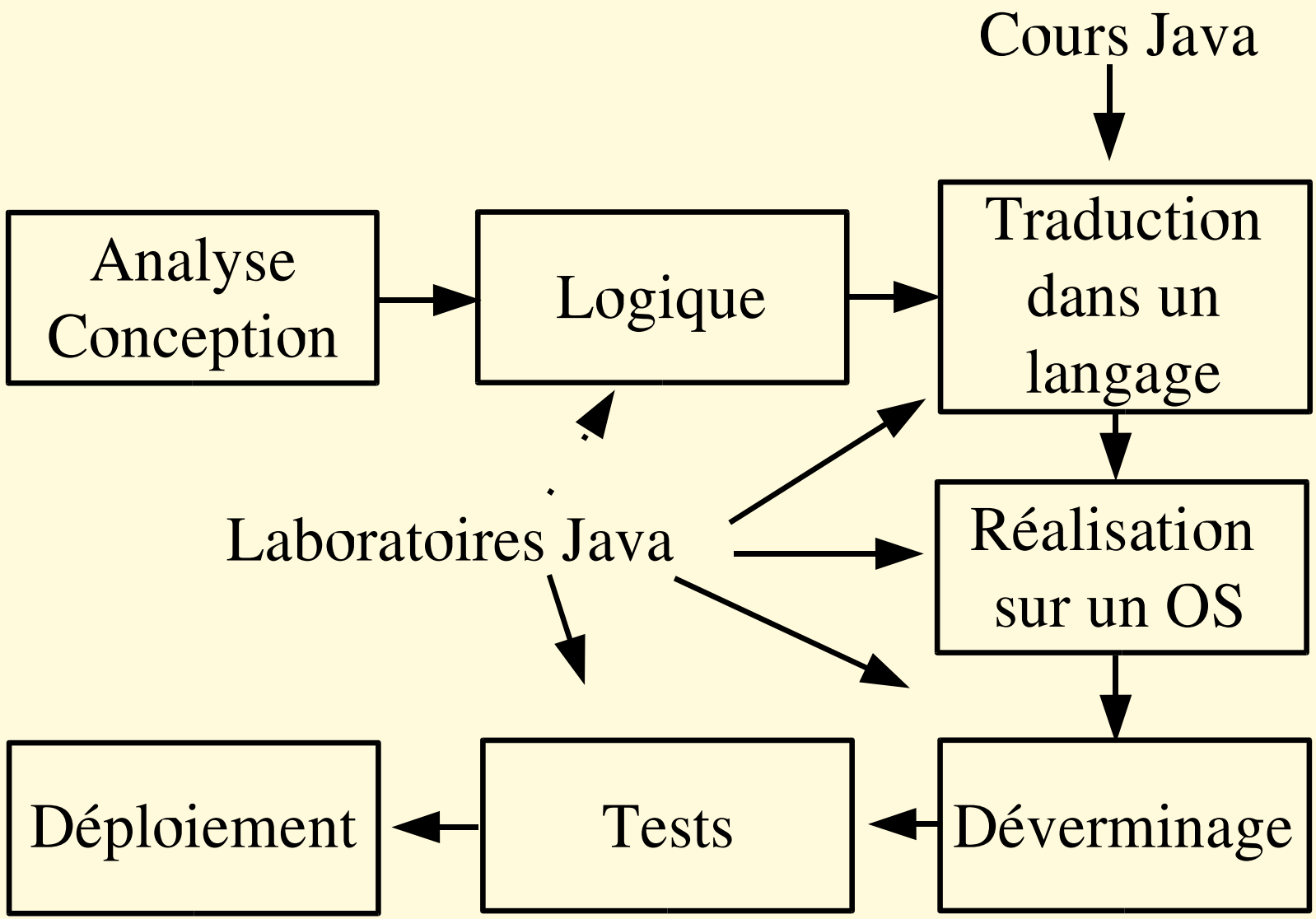
Les autres cours de langage

- Assembleur en première : permet une compréhension bas niveau de la machine
- **`C`** et **`C++`** en deuxième
- **`Cobol`** en 2ème et 3ème (Gestion)

Java et les autres cours

- Les ateliers logiciels (Gestion) : approfondissement de la connaissance du langage et apprentissage de techniques de programmation (API graphiques, *Beans*, *Design Patterns*, . . .)
- Certains cours de réseau : **`Java`** comme support pour les exemples du cours
- Certains cours comme les cours de système restent attachés au **`C`**

Les phases de développement d'un logiciel



Java en première

- Programmation classique et orientée objet (un peu)
- Cours de langage : syntaxe
- Labos
  - Mise en oeuvre de la syntaxe
  - Mise en place d'un style de programmation : choix des noms, mise en forme du code, . . .
  - Mise en place de techniques de bonnes programmation : documentation, . . .
  - Familiarisation avec **`Linux`** : édition, compilation, . . .

Les supports d'apprentissage

- L'école n'édite pas de syllabus pour ce cours
  - Ne nombreux livres dans le commerce
  - Les transparents seront disponibles au fur et à mesure
- Le livre sur lequel nous nous basons le plus est *The Java language specification, second edition* de *The Java Serie*
- Nous n'en aborderons que quelques chapitres
- Les transparents ne sont pas exhaustifs

Les supports d'apprentissage

- L'importance d'un livre d'accompagnement
- Attention : Nombre d'entre-eux se concentrent sur
  - un aspect bien particulier du langage (comme les threads)
  - une API spécifique (comme Swing)

Le site Java de l'Ecole

- Nous maintenons un site pour les étudiants

  http :// java. pit – it .com

- Vous y trouverez
  - Des liens utiles
  - Une liste commentée de livres de référence
  - Les transparents du cours
  - Des infos sur les horaires
  - Des corrigés
  - . . .

Le problème de l'Anglais

- Une connaissance de l'Anglais technique est primordiale
  - Les erreurs seront signalées en Anglais
  - La documentation est essentiellement en Anglais
- Nous essayerons d'introduire chaque nouveau terme dans les deux langues

L'évaluation

Pour le cours

- A Pâques : Interrogation écrite (type QCM) - 1/3 pts
- En juin : Oral sur machine - 2/3 pts
- Que teste-t-on ?
  - Connaître la syntaxe du langage
  - Pouvoir écrire un court programme sans faute
  - Pouvoir écrire de plus gros programmes
  - Savoir programmer proprement

La plateforme **Java**

- Historique de **Java**
- Compilation et interprétation
- La machine virtuelle **Java**
- Les outils de développement

Historique de Java

92 Création chez **SUN** de **oak** destiné aux systèmes embarqués

94 Ce langage se révèle adapté à Internet, encore jeune, grâce aux **applets**. Il devient **Java**

95 **Netscape** annonce qu'il supporte **Java** qui devient populaire

96 Première version stable et gratuite de **JDK**, le kit de développement

98 Sortie de **Java 2**

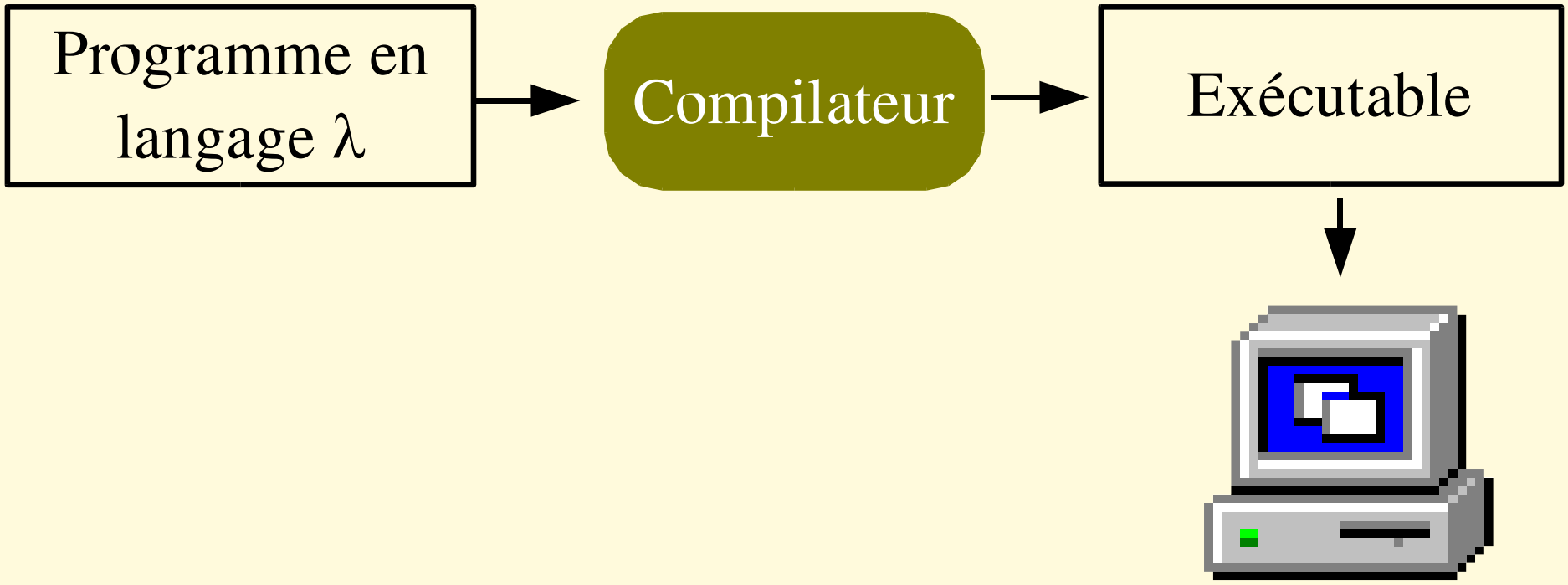
Le problème de la traduction

- Un ordinateur ne comprend que le langage machine
- Comment peut-il comprendre un autre langage (comme **`Java`**) ?
- Nécessité d'une **traduction**

1. **Compilation** : programme traduit d'une traite avant l'exécution
2. **Interprétation** : programme traduit morceau par morceau au moment de l'exécution

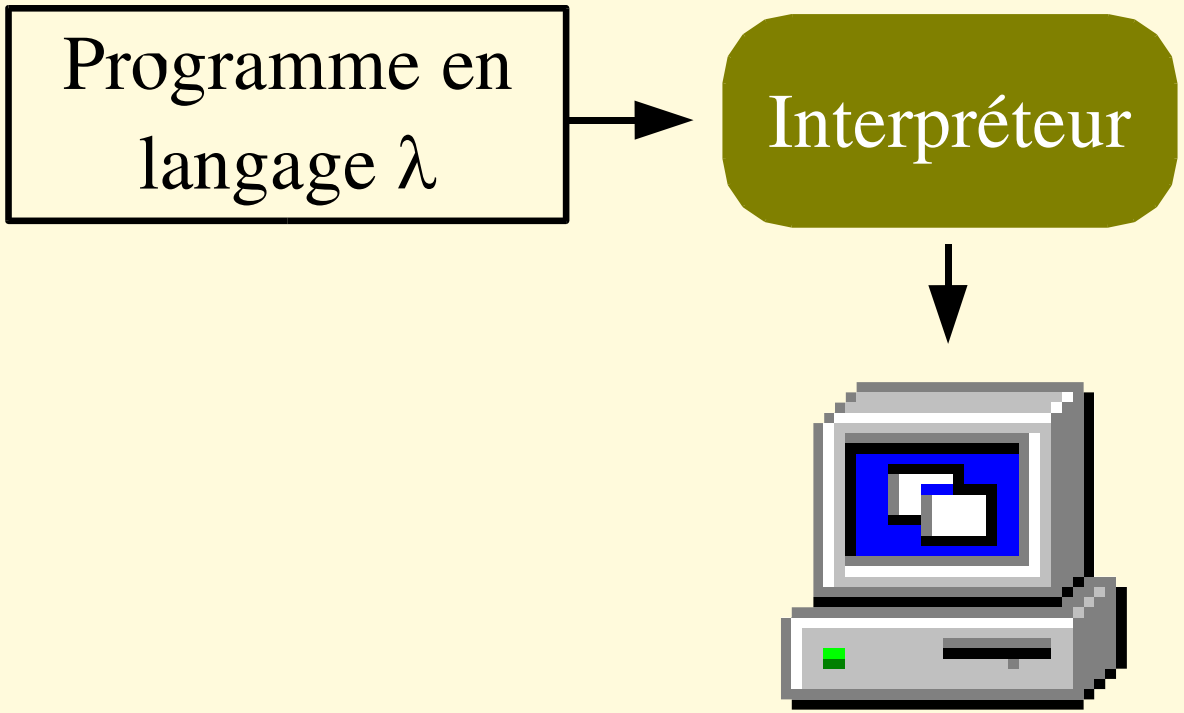
La compilation

Compilation : programme traduit d'une traite avant l'exécution



L'interprétation

Interprétation : programme traduit morceau par morceau au moment de l'exécution



Compilation vs interprétation

- **Compilation**
  - Compilé 1x, exécuté plusieurs fois
  - Nécessite un travail préparatoire
  - Distribution facile mais limitée à un système
- **Interprétation**
  - Plus lent
  - Nécessité d'un interpréteur sur le poste de travail
- **Quel est le choix adopté par Java ?**
  - Approche originale liée à la nécessité de **portabilité** maximale

Le problème de la portabilité

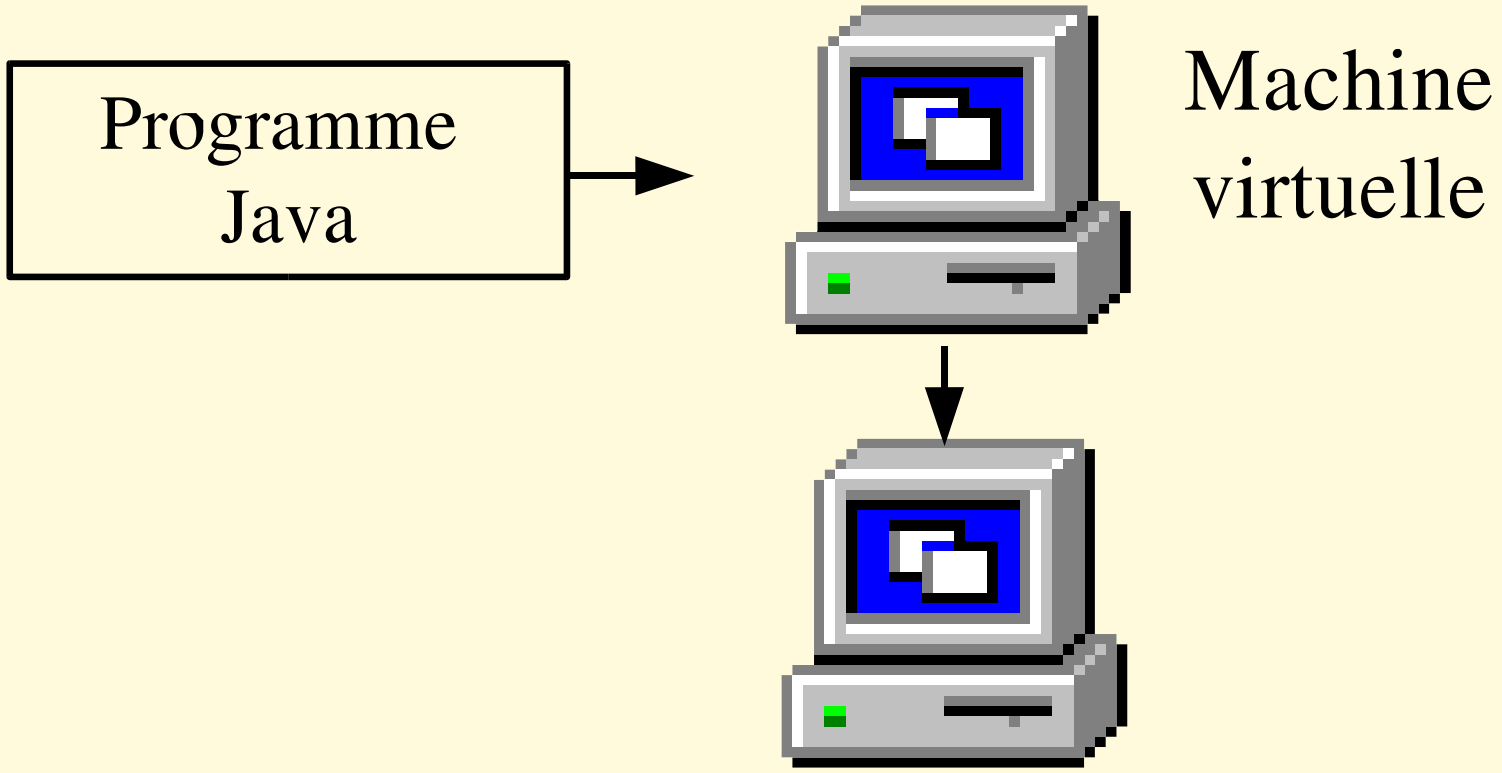
- Avec de nombreux langages (comme **C++**) un programme écrit pour un système (comme **Windows**) ne tourne pas sur un autre système (comme **Linux**)
- On parle du problème de **portabilité**
- Difficulté (lenteur, coût) de développer pour plusieurs systèmes
- Intenable pour **Java** qui s'est développé via les **applets** (système de l'hôte non maîtrisé)

La machine virtuelle

- La réponse de **SUN** a été de développer une machine **virtuelle** (virtual machine)
- Les programmes **Java** sont développés pour cette seule machine
- Mais comment les faire tourner sur un machine réelle ?

La machine virtuelle

- Via un programme qui **émule** la **machine virtuelle Java** (**JVM**)



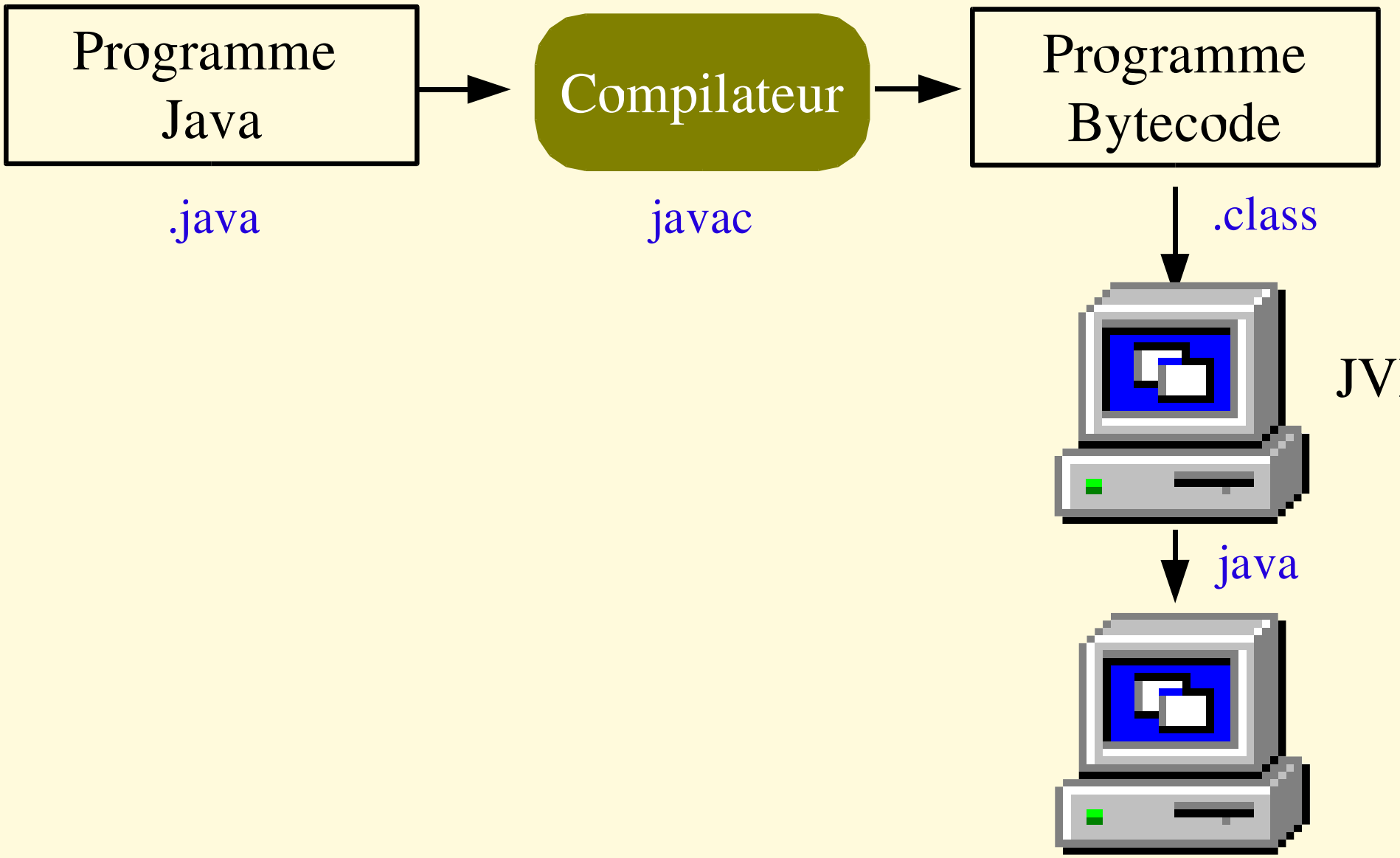
- Phénomène proche de l'interprétation

La machine virtuelle

- Le mécanisme d'interprétation est lent
- En fait, la **JVM** ne comprend pas directement le **Java** mais un langage machine propre, le **Bytecode**
- On a d'abord une phase de compilation du **Java** en **Bytecode**

On a donc une approche mixte compilation / interprétation

La machine virtuelle



La machine virtuelle

- Prenons un exemple

(fichier Hello.java)

```
// Mon premier programme
public class Hello
{
   public static void main(String [ ] args)
   {
      System.out.println("Bonjour␣!");
   }
}
```

La machine virtuelle

- Compilons-le *(extrait d'une console sous **Linux**)*

```
$ javac Hello.java
```

- On obtient la version compilée (Hello.class)
- Il s'agit de **bytecode** illisible par un humain
- Exécutons-le sur la machine virtuelle

```
$ java Hello
Bonjour !
```

Les outils de développement

J2SE : *Java Platform 2, Standard Edition*

- Comprend :
  - **javac** : le compilateur de **Java** vers **bytecode**
  - **java** : la machine virtuelle **Java** qui exécute le bytecode
  - **javadoc** : l'outil de production automatique de documentation
  - ...
- Gratuit, fourni par **SUN**
- Ancien nom : **JDK** (*Java Development Kit*)

Les outils de développement

JRE : *Java Runtime Environment*

- Comprend uniquement ce qui est nécessaire à l'exécution d'applications **Java** (la **JVM** et les bibliothèques de base)
- Accompagne les navigateurs **Web** par exemple

J2EE : *Java 2, Enterprise Edition*

- Solution adaptée au développement d'applications multi-tiers robustes et sécurisées
- Comprend **J2SE** + **(Enterprise) Beans** + **Servlets** + **Sun ONE** + ...

Les outils de développement

Les **IDE** : *Environnement de développement intégré*

- **Sun ONE Studio 5** (payant)
- **JBuilder 9**, par **Borland** (payant)
- **JCreator** light (gratuit), nécessite **J2SE** - Un bon choix pour apprendre, léger et on garde le contrôle sur tout
- **IntelliJ IDEA** (payant) - Un peu plus complexe mais plus puissant (*refactoring* notamment)

La grammaire de **Java**

- La notion de programme
- La notion de grammaire
- Les notations de la grammaire **Java**
- Un exemple simple
- Grammaire lexicale

La notion de programme

- La seule chose dont est capable un ordinateur c'est de réaliser extrémement rapidement des instructions élémentaires : **force brute**
- Toute tâche qu'on veut lui confier doit donc être **préalablement** décrite comme une **suite séquentielle d'instructions** (un programme)
- La première (dernière) instruction exécutée ne sera pas forçement la première (dernière) de la séquence → notion de **point d'entrée** (**de sortie**)

La notion de programme

- Un programme est généralement destiné à être **exécuté plusieurs fois**
  - pour des **données différentes**
  - dans un **environnement différent**
- Il doit pouvoir s'adapter à ce contexte différent

  $\longrightarrow$ notion d'instruction de **test** et de **boucle**

La notion de programme

- En outre un programme doit pouvoir
  - **résister** à des événements imprévus
    - abscence d'un fichier
    - panne réseau, machine, . . .
  - **communiquer** avec son environnement
    - lecture et écriture sur des fichiers
    - requètes au système d'exploitation
  - **utiliser**/**être utilisé** par d'autres programmes

La notion de programme

- La maitrise de tous ces aspects nécessite un langage de programmation **performant, expressif et précis**
- La programmation structurée propose un **choix limité de types d'instructions** pouvant atteindre ces objectifs
- Les langages de haut niveau ont tous intégré les principes de la programmation structurée

La notion de programme

- Mais, en outre, un langage de programmation doit pouvoir
  - **être décrit** auprès des programmeurs
  - faire l'objet d'une **compilation rigoureuse**
- Pour ce faire, on lui associe une **grammaire** permettant de déterminer la conformité de toute suite d'instructions (phrase) écrites dans ce langage

La notion de grammaire

- On peut assimiler un programme **`Java`** à une phrase écrite dans le langage **`Java`**
- Cette phrase est une séquence de mots (*token*)
- Chaque **mot** doit être un mot légal du langage
  - déterminé par une **grammaire lexicale**
- La **séquence de mots** doit être légale dans le langage
  - déterminé par une **grammaire syntaxique**

La notion de grammaire

- Le **nombre** de programmes pouvant être écrits dans un langage est **infini**
- Une grammaire syntaxique est une **description finie** de tous ces programmes
- Le compilateur utilise ces 2 grammaires pour traiter un programme (en 2 phases)
  - une analyse lexicale pour identifier les *tokens* (mots)
  - une analyse syntaxique pour vérifier la séquence des *tokens*

La notion de grammaire

- On peut établir le parallèle avec une **langue naturelle**
  - Un **dictionnaire** pour valider les mots
  - Une **grammaire** pour valider les phrases
- C'est ainsi que *"Le chat est noir"* est reconnu comme une phrase valide de la langue française
- Mais *"Le parapluie mange l'ascenceur"* aussi !

La notion de grammaire

- Il manque la notion de **sémantique**
- Cette notion ne peut pas s'exprimer avec un formalisme aussi rigoureux que les grammaires
- C'est également le cas avec les langages de programmation

Comment fonctionne une grammaire ?

Une grammaire est composée de

- l'**ensemble des mots** pouvant apparaître tels quels (les *symboles* ***terminaux***)
- un ensemble de **règles de production** composées
  - du nom de la règle (*symbole* ***non terminal***)
  - de séquences de symboles produits par la règle
- une règle de production de **départ**

Un exemple de grammaire

- Inventons un nouveau langage, le **langage MU**
- Nous devons indiquer de manière précise quelle phrase est valide dans notre langage
- Cela se fait en donnant sa grammaire

Un exemple de grammaire

Définition de la grammaire du langage MU

- Les 3 symboles terminaux : `M`,`U`,`I`
- Les 3 règles de production

```
start:
      debut MU fin

debut:                          fin:
      debut I                       I fin
      I                             I
```

- La règle de départ : `start`

Un exemple de grammaire

- Quelques **phrases correctes** en langage MU
  - `IMUI`
  - `IIMUI`
  - `IIMUII`
  - `IMUIIIIIIIIIIIIIIIIIIIIIIII`
  - . . .à l'infini

Un exemple de grammaire

- Quelques **phrases incorrectes** en langage MU
  - `MU`
  - `IMUMU`
  - `MUI`
  - `MUIMU`
  - . . .à l'infini

Exercices

- **Exercice** : Ecrire la grammaire d'un langage MU' qui accepte le sous-ensemble des phrases de MU telles que le nombre de `I` en début de phrase soit égal au nombre de `I` en fin de phrase.
  (aide : il faut penser dans l'*autre sens*)

- **Exercice** : Ecrire la grammaire d'un langage MU" qui contient tous les palindrômes à partir des symboles M, U et I.

Notations de la grammaire Java

- Nous ferons souvent référence à la grammaire de **`Java`** afin d'établir précisement ce qui est valide
- Il est important de s'habituer rapidement à la lire
- Pour cela, passons en revue les 8 **conventions** d'écriture que nous respecterons
- 1 : Un symbole **terminal** sera en **`caractère monospacé et gras`**
- 2 : Un symbole **non terminal** sera en *italique*

Notations de la grammaire Java

- 3 : Une **règle de production** aura la forme suivante

---

symbole_non_terminal :

combinaison de symboles

.

.

combinaison de symboles

- Exemple

ArgumentList :

Argument

ArgumentList , *Argument*

Notations de la grammaire Java

- 4 : Lorsque le **développement** d'une règle est **facultatif** nous collerons le postfixe (opt) à la fin de son nom
- Exemple

BreakStatement :
`break` *Identifier* (opt) **`;`**

- Cela permet une écriture plus compacte

Notations de la grammaire Java

- 5 : Lorsque la partie droite d'une règle **dépasse la largeur** d'un slide on peut la continuer à la ligne suivante moyennant une indentation suffisante
- Exemple

ConstructorDeclaration :
ConstructorModifi ers(opt) *ConstructorDeclarator*
Throws(opt) *ConstructorBody* (opt)

Notations de la grammaire Java

- 6 : L'usage de **one of** permet une plus grande concision
- Exemple

ZeroToThree : one of

```
0 1 2 3
```

---

au lieu de

ZeroToThree :

```
0
1
2
3
```

Notations de la grammaire Java

- 7 : L'usage de **but not** et **or** permet d'exclure certaines possibilités
- Exemple

InputCharacter :
UnicodeInputCharacter but not **CR** or **LF**

---

ou encore

Identifier :
IdentifierName but not *Keyword*
or *BooleanLiteral* or *NullLiteral*

Notations de la grammaire Java

- 8 : L'usage d'une **paraphrase** en caractères "helvetica" est parfois utilisé lorsque la liste des alternatives serait trop longue à énumérer
- Exemple

RawInputCharacter :
any Unicode character

Il y en a plus de 65000 !

Notations de la grammaire Java

La **grammaire complète de Java**

- peut-être consultée notamment dans le livre *"The Java Language Specification"*
- ne sera pas vue de manière exhaustive
- son étude se développera sur les trois années du cursus

Notre premier programme

- Reprenons un programme simple

```
// Premier programme
package be.heb.esi.LG1.tutorial.helloWorld;

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

- Décrivons brièvement sa structure et le sens de certains mots

Notre premier programme

Commentaires

- Les caractères // indiquent le début d'un commentaire
- Il se termine à la fin de la ligne
- Il est destiné aux humains qui lisent le programme
- L'ordinateur ne lit pas le commentaire (il serait d'ailleurs incapable de le comprendre)
- Il existe d'autres façons d'insérer un commentaire que nous verrons plus tard

Notre premier programme

package

- Permet de grouper différents morceaux de code sous une appellation commune
- Le nom est *qualifié* ce qui permet
  - un classement raisonné
  - une assurance de noms uniques
- Cette structure se retrouve généralement dans le système de fichiers
- Dans notre exemple le code se retrouvera dans : be\heb\esi\LG1\tutorial\helloWorld

Notre premier programme

class (*classe*)

- Regroupe sous un même nom toute une série de valeurs et de fonctionnalités, c'est à dire du code
- Ici, Principale est le nom de cette *class*
- Un *package* pourra contenir un nombre indéterminé de définitions de *class*
- L'ensemble de ces *class* forme le programme, la librairie que le programmeur développe

Notre premier programme

public (devant **class**)

- Délimite l'usage pouvant être fait de différents éléments d'un programme, ici de la *class*
- Ici, il signifie que la *class* est accessible à tout autre morceau de code
- Des restrictions individuelles peuvent être portées sur chaque valeurs et/ou fonctionnalités définies dans cette *class*

Notre premier programme

Les accolades

- Remarquons que la description de cette *class* commence par une accolade ouvrante et se termine par une accolade fermante
- C'est également le cas pour la fonctionnalité *main*
- De façon générale, cela servira à créer des **blocs d'éléments**

Notre premier programme

main

- La *class Principale* ne comprend aucune valeur et une seule fonctionnalité (*main*)
- Ce nom a un **rôle particulier**
- Lorsqu'on demande l'**exécution** d'un programme, celle-ci commence avec la première instruction décrite dans la fonctionnalité *main*
- Dans notre exemple, elle est composée d'une seule *instruction*

Notre premier programme

public static void main(String [] args)

- **public** précise son utilisabilité par d'autre, obligatoirement public dans ce cas précis
- **static** sera expliqué bien plus tard dans le courant de ce cours
- **void** précise le genre de message retourné par main à la fin de son exécution (ici, rien)
- String [] args sera aussi expliqué plus tard

Notre premier programme

println

- Ecrit la chaine de caractères (*String*) *Hello world* à l'écran de notre ordinateur
- Est disponible à partir de la librairie *System*
- *out* est précisé pour déterminer la sortie de l'impression (ici, la console)

Analyse lexicale de Java

- L'analyse lexicale est la première phase d'analyse d'un programme
- Pour rappel, on y identifie les *tokens* (mots) du langage
- Commençons donc notre étude par cette phase
- Ensuite, nous attaquerons la grammaire syntaxique

Unicode

- **Java** a fait le choix de l'**Unicode** (2.0) pour son jeu de caractères
- Rupture avec l'*occidano-centrisme* habituel
- Les jeux de caractères courants
  - **ASCII** : 7/8 bits
  - **EBCDIC** : 7 bits (**IBM**)
  - **Unicode** : 2 bytes, 16 bits
    (les 128 premiers caractères sont ceux de l'**ASCII**)

Unicode

- Souvent, l'**OS** n'utilise pas le jeu de caractères **Unicode**
- Nécessité d'une traduction du programme avant l'analyse lexicale
  - code de l'**OS** $\longrightarrow$ **Unicode**
- Limitation sur le choix des noms *externes*

Les caractères d'espacements

- Les caractères d'espacements (*WhiteSpace*) n'ont pas de sens en **`Java`**
- Ils peuvent être utilisés librement entre les mots
- Exception : ils reprennent leur sens dans les chaînes (*String*)

WhiteSpace :
the ASCII SP character, also known as "space"
the ASCII HT character, also known as "horizontal tab"
the ASCII FF character, also known as "form feed"
LineTerminator

- Le "form feed" est le saut de page imprimante

Les caractères d'espacements

LineTerminator :
the ASCII LF character, also known as "newline"
the ASCII CR character, also known as "return"
the ASCII CR character followed by the ASCII LF character

- Les *LineTerminator* permettent de numéroter les lignes
- Information fournie dans les messages d'errreurs par exemple
- Cette grammaire montre bien que là où on peut avoir un espace, on peut avoir un passage à la ligne

Le commentaire

- Le commentaire (*comment*) est une partie du programme qui décrit le programme
- Il est destiné au humains qui le lisent
- Il ne modifie en rien ce que fait le programme
- Il est néanmoins crucial dans le développement d'une application
- En Java, il y a 3 sortes de commentaire

Le commentaire

- */* text */*
  - Commentaire traditionnel
  - Tout le texte entre les caractères */** et **/* est ignoré
  - Idem **C** et **C++**
- *// text*
  - Simple ligne de commentaire
  - Tout le texte entre les caractères *//* et la fin de la ligne (*LineTerminator*) est ignoré
  - idem **C++**

Le commentaire

- */** documentation */*
  - Tout le texte entre les caractères */*** et **/* est ignoré
  - Est utilisé pour la production automatique de documentation
  - Traîté par une autre application (**`javadoc`**)

Le commentaire

- Les commentaires ne peuvent pas être imbriqués
  - */∗* et *∗/* n'ont pas de signification spéciale dans un commentaire commencé par *//*
  - *//* , */∗* et */∗∗* n'ont pas de signification spéciale dans un commentaire commencé par */∗* ou */∗∗*
    - Ex : */∗ this comment /∗ // /∗∗ ends here: ∗/*
    - Ex : */∗ commentaire ∗/*erreur *∗/*
- Un commentaire peut êre rencontré n'importe où **entre** les mots

Les tokens (mots du langage)

- Tous les autres caractères vont former les *tokens*, symboles terminaux de l'analyse syntaxique
- On rencontre 5 sortes de tokens

Token : one of
Identifi er *Keyword* *Literal* *Separator* *Operator*

- Les identifiants (*identifier*) servent à nommer des choses, par exemple les variables
  - Cette notion sera développée plus loin dans le cours

Les tokens (mots du langage)

- Les *keyword* (mots-clés ou réservés) sont des noms ayant un sens particulier
  - Expliqués petit à petit tout au long de l'année

Keyword : one of

```
abstract boolean break byte case catch char
class const continue default do double else
extends final finally float for goto if
implements import instanceof int interface
long native new package private protected
public return short static super switch
synchronized this throw throws transient
try void volatile while
```

Les tokens (mots du langage)

- Les littéraux (*literal*) représentent des valeurs
  - Notion développée plus loin dans le cours
- Citons les séparateurs et les opérateurs
  - Expliqués au fur et à mesure dans le cours

Separator : one of

```
( ) { } [ ] ; , .
```

Operator : one of

```
= > < ! ~ ? : == <= >= != && ||
++ -- + - * / & | ^ % << >> >>>
+= -= *= /= &= |= ^= %= <<= >>= >>>=
```

Les tokens (mots du langage)

- L'analyseur lexical reconnait des mots les plus longs possibles
  - `--` est reconnu comme le mot `--` et pas comme les deux mots `-` et `-`
  - `---` est reconnu comme le mot `--` suivi du mot `-` et pas l'inverse

Analyse Lexicale : récapitulatif

- Rappelons les phases de l'analyse lexicale
  - Traduction du jeu de caractères de l'**`OS`** vers **`Unicode`**
  - Identification des espaces
  - Identification des commentaires
  - Identification des *tokens*
- Seuls les tokens passeront à la deuxième phase (l'analyse grammaticale)
- Les commentaires et espaces n'en feront pas partie

Analyse Lexicale : récapitulatif

- Exercice : détailler le comportement de l'analyseur lexical pour notre petit programme

```
// Premier programme
package be.heb.esi.LG1.tutorial.helloWorld;

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

La notion de bloc

- Préliminaires
- Le bloc

Préliminaires

- Attaquons à présent en détail les éléments du langage
- Difficile
  - de tout voir en même temps
  - de choisir par où commencer
- Volonté d'une cohérence avec le cours de logique
  $\Longrightarrow$ commençons par les notions de bloc et de données

La notion de bloc

- Ecrire un programme revient d'une certaine manière à décrire deux choses
  - des **zones de la mémoire** vive où stocker des données (*data*)
  - des **actions** à réaliser sur le contenu de ces zones
- **Java** permet ces descriptions sous forme de blocs rassemblant à la fois l'un et l'autre
- Ainsi un *block* est une séquence d'instructions et de déclarations locales de variable enfermée dans des accolades

La notion de bloc

Block :
{ *BlockStatements*(opt) **}**

BlockStatements :
BlockStatement
BlockStatements BlockStatement

BlockStatement :
LocalVariableDeclarationStatement
Statement

- Exécuté en séquence de la première instruction à la dernière . . .lorsque tout se déroule normalement

Les données

- Les types
- Les littéraux
- Les variables
- Les déclarations

Les types

- **Toute donnée aura un type**
  - Renforce la cohérence sémantique
  - Permet une allocation mémoire adaptée
- Quels types trouve-t-on ?
  - Des types **primitifs prédéfinis** :
    - entier, réel, booléen (logique)
  - Des types **références prédéfinis** :
    - String (chaîne de caractères)
  - Des types **références définis par le programmeur**

Les types primitifs

- Il y a 8 types primitifs

PrimitiveType : one of
NumericType **`boolean`**

NumericType : one of
IntegralType *FloatingPointType*

IntegralType : one of
`byte short int long char`

FloatingPointType : one of
`float double`

Les types numériques entiers

byte, **short**, **int** et **long** (sauf **char** qui est vu à part) :

- Nombres signés (en complément à 2)
- Codés sur (respectivement) 8-bit, 16-bit, 32-bit et 64-bit
- Comprennent donc les valeurs (respectivement)
  - -128 à 127
  - -32768 à 32767
  - -2147483648 à 2147483647
  - -9223372036854775808 à 9223372036854775807

Les types numériques entiers

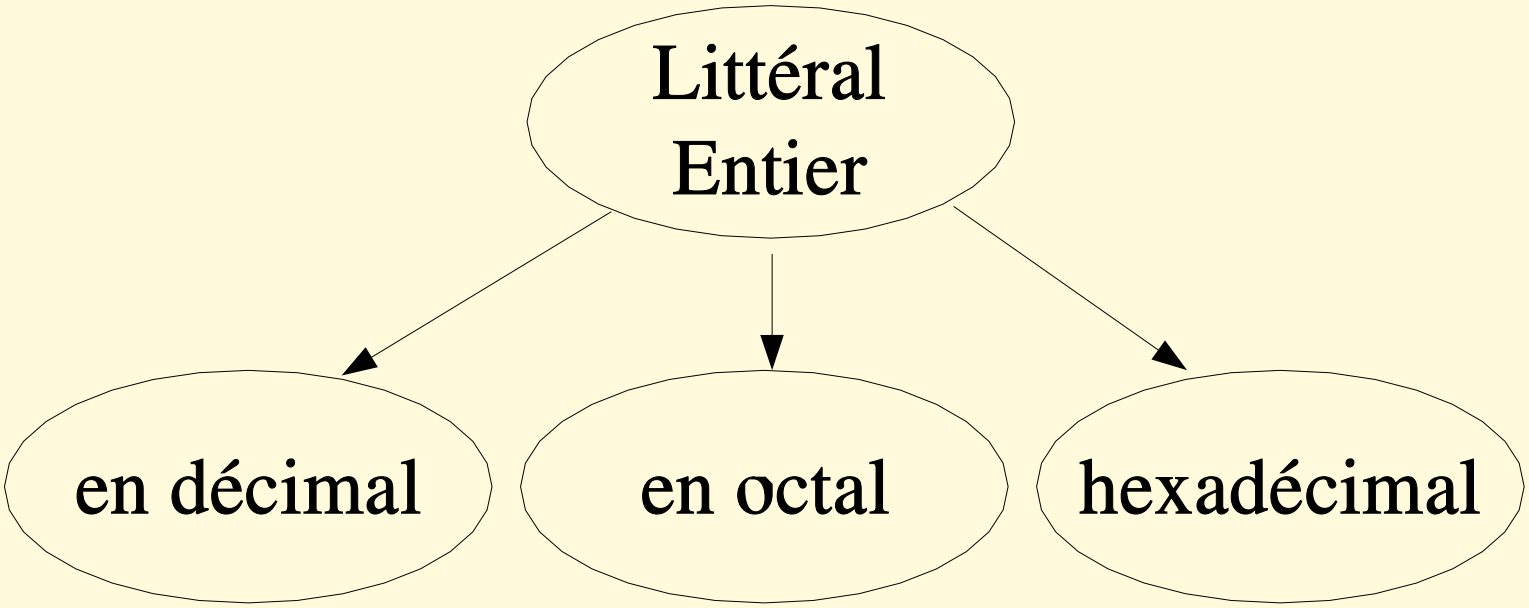
- Il s'agit donc d'une **modélisation** de la notion mathématique d'entier
  - Capacité limitée
  - Possibilité d'obtenir un ***out of range*** lors d'un calcul
- Parmi ces types, **int** est le plus utilisé.

Les littéraux entiers

- ***Littéral*** : représentation d'une valeur
- Pour les entiers, on a

IntegerLiteral : one of
    DecimalIntegerLiteral *HexIntegerLiteral* *OctalIntegerLitera*
DecimalIntegerLiteral :
    DecimalNumeral *IntegerTypeSuffi x* (opt)
HexIntegerLiteral :
    HexNumeral *IntegerTypeSuffi x* (opt)
OctalIntegerLiteral :
    OctalNumeral *IntegerTypeSuffi x* (opt)
IntegerTypeSuffi x : one of
    `l` `L`

Les littéraux entiers



Avec un suffixe facultatif

Les littéraux entiers

- Un ***DecimalNumeral*** (numérique décimal) correspond à la notation conventionnelle des entiers
  - Rien d'autre que les chiffres n'est toléré
  - Pas d'espace, de virgule ou de point pour séparer les milliers par exemple
  - Exemples corrects : 0 1275 1456456421
  - Exemples incorrects :
    12,3 12.3 1,000 1.0 1 0000

Les littéraux entiers

- Un ***HexNumeral*** (numérique hexadécimal)
  - Indiqué en faisant précéder le littéral de 0x ou 0X
  - Comprend, outre les 10 chiffres, les lettres a,b,c,d,e et f (minuscules ou majuscules)
  - Exemples : 0x0 0X1a2B 0x122 0XFFFF
- Un ***OctalNumeral*** (numérique octal)
  - indiqué en faisant précéder le littéral de 0
  - Comprend les chiffres de 0 à 7
  - Exemples : 00 0777 01234567

Les littéraux entiers

- Le suffixe (l ou L) permet de distinguer un littéral de type **int** d'un littéral de type **long**
- Pas de littéral de type **byte** ou **short** ?
  - Non, mais un littéral **int** sera converti automatiquement si nécessaire
- Pas de **littéral négatif** ?
  - A proprement parler, non !
  - Mais l'opérateur unaire – permet de les construire aisément

Le type numérique caractère

char

- Représente les caractères Unicode
- Entier non signé sur 16-bit (le code du caractère)
- Est assimilé à un entier
  - pourra intervenir dans des calculs entiers (déconseillé)

Les littéraux caractères

- On dispose de plusieurs notations pour un caractère

CharacterLiteral :
  ' *SingleCharacter* '
  ' *EscapeSequence* '

SingleCharacter :
  any character but not ' or \ or *LineTerminator*

Les littéraux caractères

- La notation la plus simple est le **caractère entre *quote*** (*SingleCharacter*)
- Exemples : ’a’, ’–’, ’=’
- Cela n’est pas toujours possible
- Contre exemples : ’ ’ ’, ’ \’

Les littéraux caractères

- Il existe des notations spéciales pour certains caractères (*EscapeSequence*)

\n	line feed	\r	carriage return
\t	tabulation	\"	
\b	backspace	\'	
\f	form feed	\\	

- Exemples : '\n' ,'\\' ,'\''

Les littéraux caractères

- De plus, on peut toujours spécifier un caractère via son code Unicode
- Exemples : ’\u0F40’ pour le *KA* tibétain ou ’\u17E0’ pour le chiffre 0 Khmer

Les types numériques à virgule flottante

float, **double**

- Respectent la norme IEEE754
- Codés sur (respectivement) 32-bit, et 64-bit
- On utilisera plus souvent le type **double**

Les types numériques à virgule flottante

- Il s'agit d'une **modélisation** de la notion mathématique de réel
  - Capacité limitée
    - $\longrightarrow$ possibilité de ***out of range*** lors d'un calcul
  - Précision limitée
    - $\longrightarrow$ **imprécision** (parfois grande) lors d'un calcul
    - ex : 10^{-30} est représentable et pourtant $(1 + 10^{-30}) - 1$ donnera 0 et pas 10^{-30}

Les littéraux à virgule flottante

- On dispose d'une notation assez souple

FloatingPointLiteral :

Digits **.** *Digits*(opt) *ExponentPart*(opt) *FloatTypeSuffi x*(opt)

. *Digits ExponentPart*(opt) *FloatTypeSuffi x*(opt)

Digits ExponentPart FloatTypeSuffi x(opt)

Digits ExponentPart(opt) *FloatTypeSuffi x*

ExponentPart :

e *signedInteger*

E *signedInteger*

FloatTypeSuffi x : one of

f F d D

Les littéraux à virgule flottante

partie entière	.	partie décimale	E	exposant	suffixe

- 4 parties optionnelles (mais pas ensembles)
  - Cela doit rester censé
  - On ne peux pas le confondre avec un entier
- En l'absence de suffixe, le littéral est un double
- Exemples : 1.2E3, 1.F, .1, 1e−2d, 1f
- Contre-exemples : 1, .E1, E1

Le type booléen

boolean

- Appelé aussi **logique**
- Possède 2 seules valeurs : **true** (vrai) et **false** (faux)
- Attention : **Java** fait bien la distinction avec les entiers
  - Pas d'assimilation ou de conversion possible
  - Au contraire du **C/C++**

La chaîne de caractères

String

- Permet de représenter une séquence de caractères

StringLiteral :

" *StringCharacters*(opt) "

StringCharacters :

StringCharacter

StringCharacters StringCharacter

StringCharacter :

InputCharacter but not " or \

EscapeSequence

La chaîne de caractères

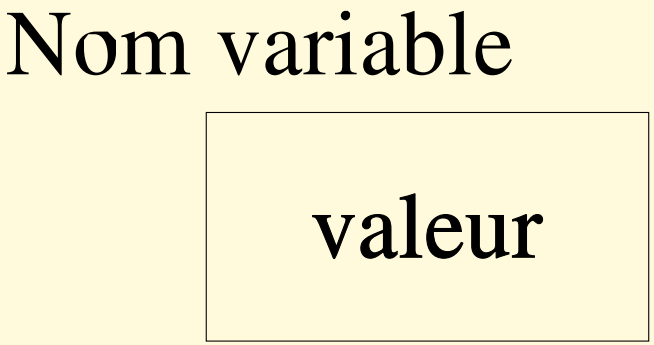
- Pour rappel, *EscapeSequence* est un caractère d'échappement (cf. le type **char**)
- Exemples de String :
  - "Bonjour␣"
  - "'’Un␣peu␣de␣tout'␣:␣\"\n\\"

La variable

- Désignation générique d'un emplacement de la mémoire vive
- Concept équivalent à celui de variable au cours de logique
- **Toute variable possède un type**
- Elle ne pourra contenir que des valeurs de ce type
- L'allocation mémoire diffère entre les variables d'un type primitif et celles d'un type *référence*

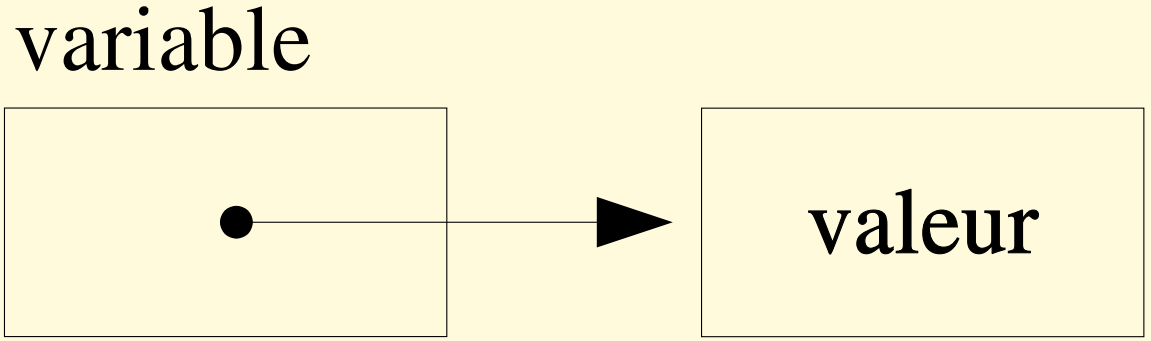
Variable et allocation mémoire

- Pour un type primitif
  - La variable indique directement la zone mémoire où se trouve la valeur



Variable et allocation mémoire

- Pour un type *référence*
  - La variable indique une zone mémoire contenant l'adresse de la zone mémoire contenant la valeur (indirection)



- La différence se fera sentir au moment de l'assignation
- Nous y reviendrons

Déclaration d'une variable locale

- (règles légèrement simplifiées)

LocalVariableDeclarationStatement :
    `final`(opt) *Type VariableDeclarators* **`;`**

VariableDeclarators :
    VariableDeclarator
    VariableDeclarators **`,`** *VariableDeclarator*

VariableDeclarator :
    Identifier
    Identifier **`=`** *Expression*

Déclaration d'une variable locale

- *Identifier* représente le nom de la variable
- *Expression* est l'éventuelle valeur initiale
- Exemples
  - **int** i ;
  - **double** poids, taille ;
  - String nom;
  - **boolean** ok=**true**,fini;
  - **char** lettre , chiffre ='1' ;

    (pas de valeur initiale pour lettre)

Nom d'une variable

- Quel nom peut-on choisir ?
- Nous ferons la différence entre
  - Ce qui est permis par **`Java`**
  - Les conventions supplémentaires que nous vous demanderons de respecter

Nom d'une variable

- Règles imposées par la grammaire
  - L'identifiant a une longueur illimitée
  - Il ne contient que des *lettres* et des *chiffres* ainsi que $ et _ pour des raisons historiques (**C**, **C++**)
  - Il ne commence pas par un chiffre
  - Il ne peut se confondre avec un *keyword* (mot-clé) ou un *litteral* (littéral)
  - Ex valides : nom, Nom, Nom23, Unpeu2touT, _interne, a$b
  - Ex invalides : 2main, le total , **for**, **true**

Nom d'une variable

- Pour **Java**, une lettre est n'importe quel caractère qui est considéré comme une lettre dans au moins une langue
  - Exemples : téléphone, Π
- Pour **Java**, un chiffre est n'importe quel caractère qui est considéré comme un chiffre dans au moins une langue
  - Exemples : on peut utiliser les chiffres arabes actuels (pour autant qu'on arrive à les taper dans l'éditeur)

Nom d'une variable

- Conventions supplémentaires (pas propres à l'école mais utilisées dans le monde entier)
  - Eviter $ et _
  - Une variable commence par une minuscule
  - Si elle contient plusieurs mots accollés, les suivants commencent par une majuscule
  - Les noms sont explicites
  - Les articles sont omis
  - Exemples : annéeNaissance, numéroTéléphone

Nom d'une variable

- Autres conventions (propres à l'école)
  - Certains noms ont un rôle communément accepté : i, j, min, max, pi, nb, . . .
  - Exemples : nbValeurs, minPoids
  - Déclarer toutes les variables en début de bloc
  - Déclarer une variable par ligne
  - Commenter chaque variable

Variable locale et valeur initiale

- Lors de la déclaration, on peut assigner une valeur initiale à la valeur (cf. grammaire)
- Cette valeur est n'importe quelle expression calculable à cet endroit là (à l'exécution)
- Exemple

```
int  poidsKilo  = 20;  //  Un poids en kilos
int  poidsGramme = 1000*poidsKilo; // L'équivalent en grammes
```

Scope (portée) d'une variable locale

- Définition : Le ***scope*** (**portée**) d'une variable indique la portion du programme où elle *existe*
- En **`java`** : Le *scope* d'une variable locale
  - est le *block* dans lequel sa déclaration apparaît
  - et ce dès sa propre initialisation
  - en incluant toutes les éventuelles autres variables déclarées à sa droite

Scope (portée) d'une variable locale

- Exemples :

```
{
    int x = y; // y n'est pas connu
    int y = 1;
}
```

```
{
    int x = 1;
    int y = x; // OK
}
```

Scope (portée) d'une variable locale

```
{
    int x;
    int y = x; // x pas initialisé
}
```

```
{
   int x;
   x = 1;
   int y = x; // OK
}
```

Scope (portée) d'une variable locale

```
{
   int x;
   int y = 1;
   x = y;
}
```

```
{
   int x = 1, y = x; // OK
}
```

```
{
   int x = x; // le x de droite n'est pas encore initialisé
}
```

Constante locale

- La clause **final** permet d'indiquer qu'il ne s'agit pas d'une variable mais d'une constante
- Lorsqu'une valeur lui est donnée pour la première fois, elle ne peut plus changer
- La valeur est donnée
  - Soit à la déclaration
    (cf. *Expression* dans la grammaire)
  - Soit par une assignation ultérieure

Constante locale

- Exemple :

```
final int x = 1;
final int y;
y = 2*x;
x = 2; // Erreur : possède déjà une valeur
y = 3; // Idem
```

- Pourquoi utiliser une constante au lieu d'un littéral ?
  - Une meilleure **lisibilité**
  - Une **modification** plus aisée

Constante locale

- La convention de nom est différente dans le cas des constantes
- On recommande de
  - Tout mettre en **majuscules**
  - Utiliser _ pour séparer les mots
- Exemple

```
final double PI = 3.1415;
final int TAUX_TVA = 21;
```

Les expressions

- Définitions
- Les Expressions *entières*
- Les Expressions *flottantes*
- Les Expressions *caractères*
- Les Expressions *chaînes de caractères*
- Les Expressions *booléennes*

Définitions

- **Expression** : calcul faisant intervenir une ou plusieurs valeurs pour une opération déterminée
- Exemples
  - 1 + 2 : les valeurs sont 1 et 2, l'opération est l'addition
  - − 1 : la valeur est 1, l'opération est la négation
  - 2 : la valeur est 2, il n'y a pas d'opération

Définitions

- La **valeur d'une expression** est le résultat de ce calcul
- Exemples
  - la valeur de l'expression $1+2$ est 3
  - la valeur de l'expression -1 est -1
  - la valeur de l'expression 2 est 2

Définitions

- Le **type d'une expression** est le type de sa valeur
- Exemples
  - le type de l'expression 1+2 est **int**
  - le type de l'expression − 1 est **int**
  - le type de l'expression 2 est **int**
- Quel est le type de l'expression 5/2 ?

Définitions

- **Terminologie** : dans une expression, on appelle
  - **Opérande** : valeur
  - **Opérateur** : opération
- Exemple :

1	+	2
opérande	opérateur	opérande

Définitions

- Un opérateur peut être
  - **unaire** : un seul opérande (ex : -1)
  - **binaire** : deux opérandes (ex : $1+2$)
- Il existe même un opérateur **ternaire**
- Les opérandes sont **évalués de gauche à droite**
- Les opérandes sont **complètement évalués avant** tout calcul lié à leur **opérateur**
  (sauf pour && et || voir plus loin)
- Aura son importance lors d'**effets de bord**

Les expressions entières

- Expressions ne faisant intervenir que des opérandes d'un même type entier : **int**, **long**
- Et pas **short** ou **byte** ?
  - Convertis automatiquement en **int** pour les calculs
- Son type sera celui de ses opérandes
- Opérateurs **unaires**
  - + : ne réalise aucune opération
  - – : réalise l'opposé mathématique classique

Les expressions entières

- Opérateurs **binaires** disponibles :
  - `+` : addition
  - `-` : soustraction
  - `*` : multiplication
  - `/` : division **entière**
    - `1 / 2` vaut `0` `71 / 21` vaut `3`
  - `%` : **modulo**
    - `1 % 2` vaut `1` `71 % 21` vaut `8`

Les expressions entières

- Exemples
  - 1 + 2 vaut 3 (de type **int**)
  - 1L + 2L vaut 3L (de type **long**)
- Peut-on mélanger les types ?
  - Normalement pas
  - mais 1L + 2 sera accepté (conversion implicite)
  - Nous détaillerons ce point plus tard

Division et reste avec des nombres négatifs

- / : la règle des signes s'applique
  - $-5/-3$ vaut 1
  - $5/-3$ vaut -1
- % : respecte la règle $(a/b) * b + (a\%b) = a$
  - En pratique : valeur de même signe que dividende
  - $-5 \% -3$ vaut -2
  - $5 \% -3$ vaut 2
  - $-5 \% 3$ vaut -2

Erreurs de calcul

- La **division par zéro**
  - Lance une *exception* (*ArithmeticException*)
  - Nous verrons ce concept bien plus tard
  - En pratique, pour l'instant, cela **arrête le programme** avec un message explicite
- Le **dépassement de capacité**
  - N'est **pas détecté** par la machine virtuelle
  - Le résultat est tout simplement faux

Les expressions entières

- Les opérandes pouvant intervenir dans une expression peuvent être
  - un **littéral** : cf. exemples précédents
  - une **variable**
    - **int** i = 3; ... i + 2... : expression de valeur 5
    - **int** i = 3; ... i + i ... : expression de valeur 6
  - une **expression**
    - **int** i = 3; ... i + i + 2... : vaut 8

Priorité et associativité

- Problème avec l'expression :

  `int i = 3; ... i + i * 2...`

- Le compilateur va-t-il la comprendre comme
  - (i+i) *2 de valeur 12 ?
  - i + (i *2) de valeur 9 ?
- La stratégie d'évaluation se base sur :
  - la **priorité** d'un opérateur
  - l'**associativité** des opérateurs de même priorité

Tableau des priorités et associativités

- (opérateurs déjà vus)

priorité	opérateur	associativité
grande	$-$, $+$ unaires	$\Longleftarrow$
	$*$, $/$, $\%$	$\Longrightarrow$
faible	$-$, $+$ binaires	$\Longrightarrow$

- Exemple : comment comprendre
  - $3 + 3 * 2 + 1$?
  - $3 + 3 * 2 / - 4 + 5 \% 8$?

Priorité et associativité

- $3 + 3 * 2 + 1$

  $\longrightarrow$ $3 + (3 * 2) + 1$ (priorité)

  $\longrightarrow$ $(3 + (3 * 2)) + 1$ (associativité)

  $\longrightarrow$ $(3 + 6) + 1$ (évaluation)

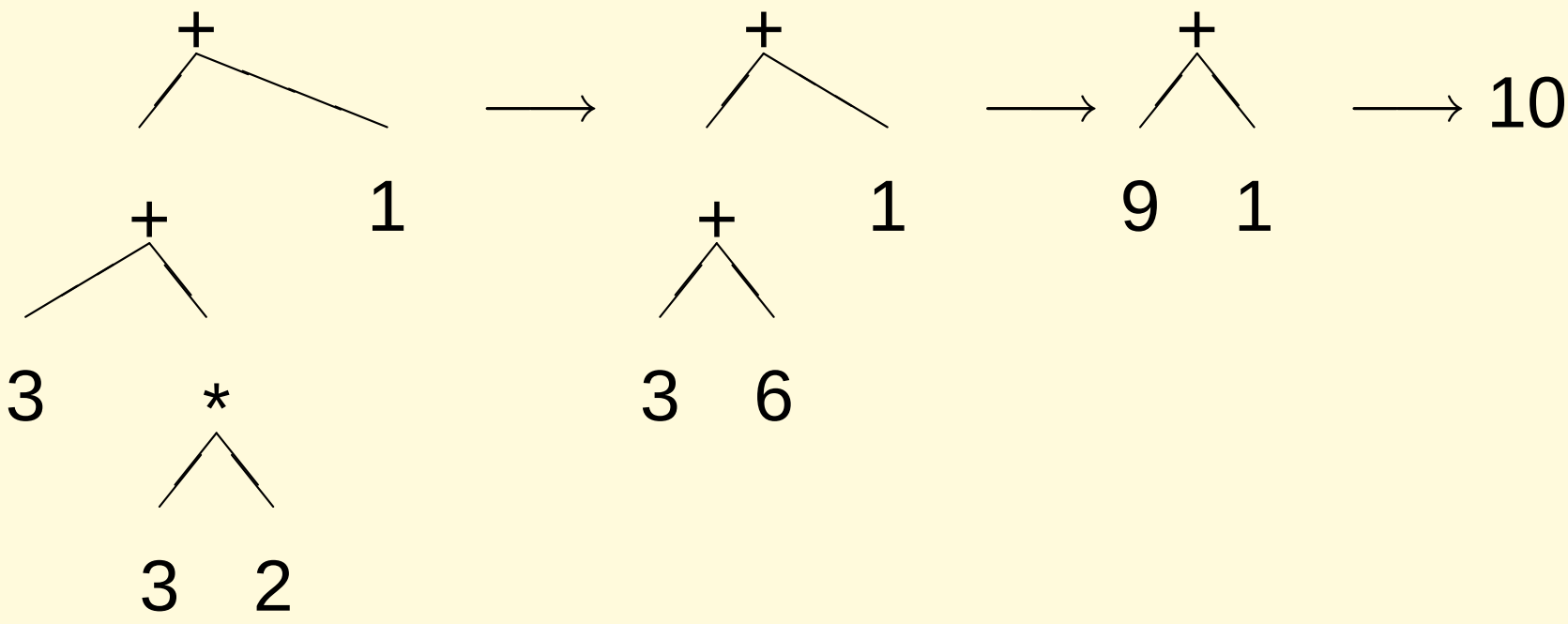
  $\longrightarrow$ $9 + 1$

  $\longrightarrow$ 10

Priorité et associativité

- Représentation sous forme d'arbre de

  $(3 + (3 * 2)) + 1$



Priorité et associativité

- $3 + 3 * 2 / - 4 + 5 \% 8$

$\longrightarrow 3 + (3 * 2 / - 4) + (5 \% 8)$ (priorité)

$\longrightarrow (3 + (3 * 2 / - 4)) + (5 \% 8)$ (associativité)

$\longrightarrow (3 + (3 * 2 / (- 4))) + (5 \% 8)$ (priorité)

$\longrightarrow (3 + ((3 * 2) / (- 4))) + (5 \% 8)$ (associativité)

$\longrightarrow (3 + (6 / (- 4))) + (5 \% 8)$ (évaluation)

$\longrightarrow (3 + (6 / -4)) + (5 \% 8)$

$\longrightarrow (3 + -1) + (5 \% 8)$

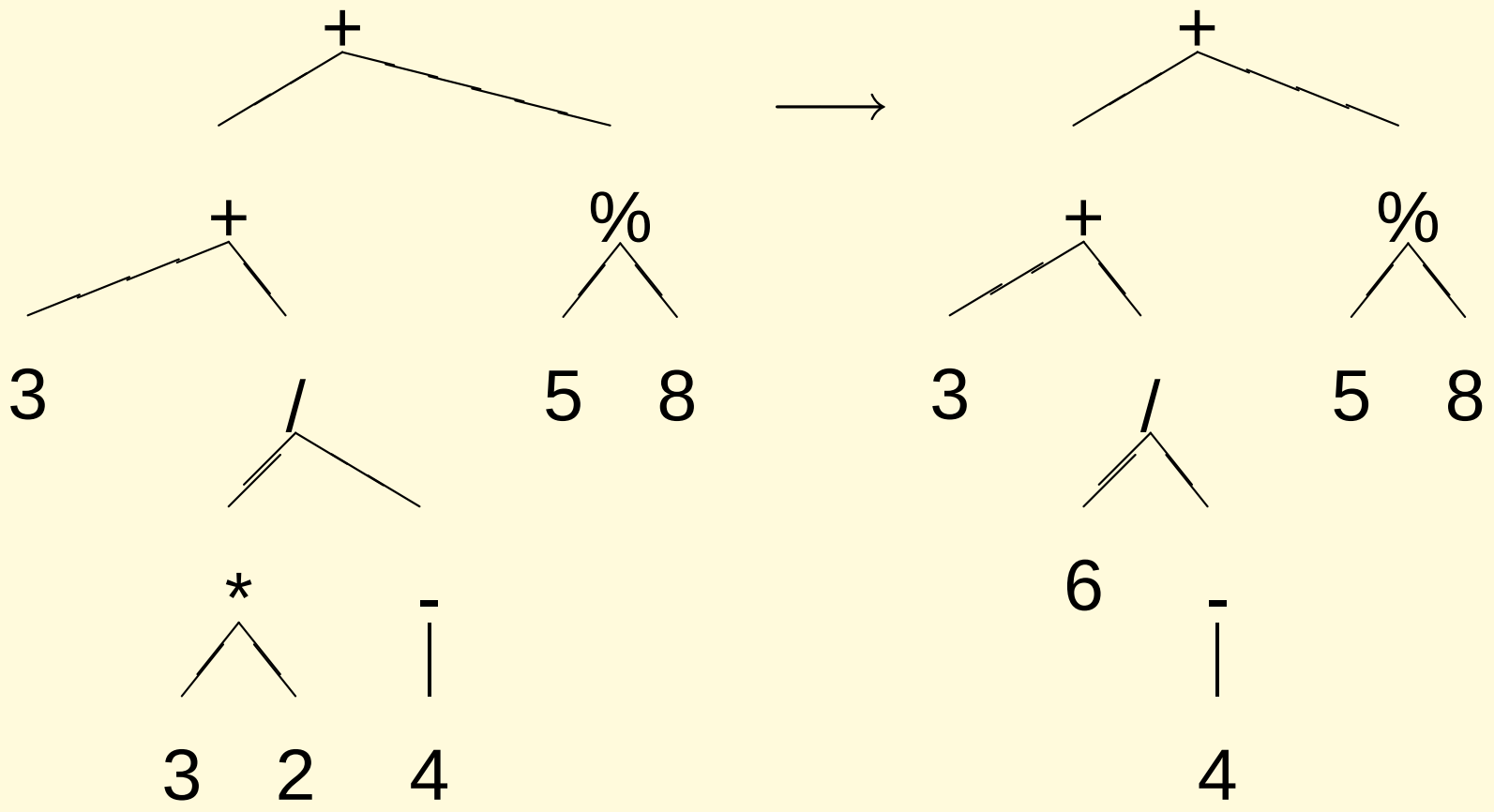
$\longrightarrow 2 + (5 \% 8)$

$\longrightarrow 2 + 5$

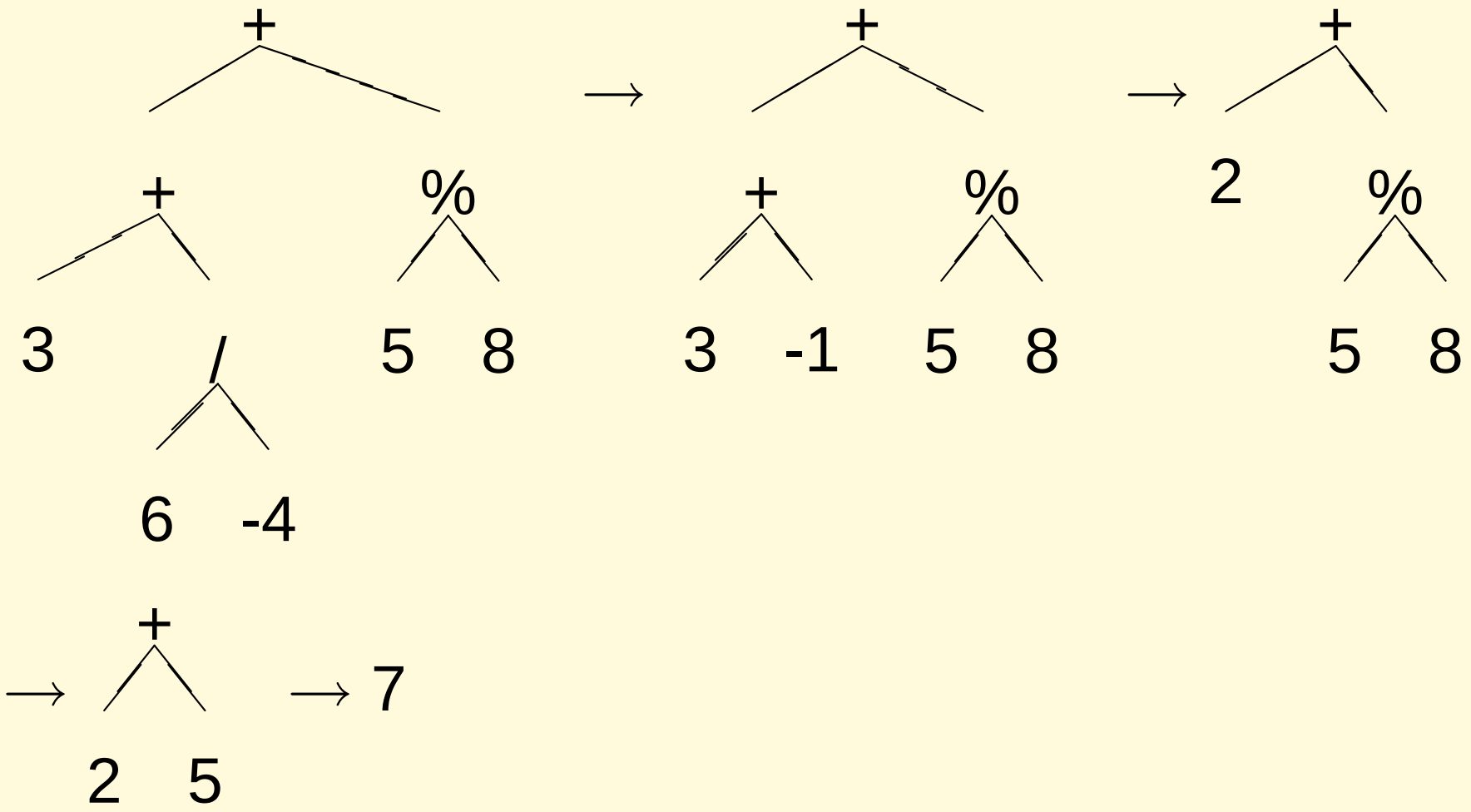
$\longrightarrow 7$

Priorité et associativité

- Arbre de $(3 + ((3 * 2) / (-4))) + (5 \% 8)$



Priorité et associativité



Priorité et associativité

- On peut forcer la stratégie du compilateur en **paranthésant** l'expression
- Le compilateur traite indépendemment chaque expression parenthésée
- Exemple : $(3 + 3) * (2 + 1) \longrightarrow 6 * 3 \longrightarrow 18$
- Moralité : mieux vaut parenthéser complètement
  - **ordre explicite** de l'évaluation
  - **clarté** de l'expression pour les autres utilisateurs

Les expressions flottantes

- Expressions ne faisant intervenir que des opérandes d'un même type flottant : **double**, **float**
- Son type sera celui de ses opérandes
- Opérateurs **unaires**
  - `+` : ne réalise aucune opération
  - `-` : réalise l'opposé mathématique classique

Les expressions flottantes

- Opérateurs **binaires** disponibles :
  - `+` : addition
  - `-` : soustraction
  - `*` : multiplication
  - `/` : division
- Le reste est identique aux entiers
  - Opérande : littéral, variable ou expression
  - Mêmes priorités et associativités
  - Dépassement de capacité non détectée

Les expressions flottantes

- Exemples :
  - 1.0 + 2.3 vaut 3.3 (type **double**)
  - 1.0d – 2.3d vaut –1.3 (type **double**)
  - 7.0f / 3.5 f vaut 2.0f (type **float**)
  - 1. / 3. **ne vaut pas** 0.33333...

    mais 0.33...3

Les expressions caractères

- **Aucun opérateur** n'est spécifiquement dédié à l'usage d'opérandes de type **char**
- On peut utiliser tous les opérateurs sur les entiers (rappel : pas recommandé)

Les expressions de chaînes de caractères

- Ne font intervenir que des opérandes de type String
- La valeur est de type identique : String
- Un seul opérateur (binaire) disponible :
  - + : concaténation de deux chaînes
    - "Hello␣" + "World" vaut valeur "Hello␣World" (de type String)
- Le reste est identique aux entiers
  - Opérande : littéral, variable ou expression
  - Mêmes priorités et associativités que les autres +

Les expressions booléennes

- Ne font intervenir que des **boolean**
- La valeur est de type identique
- Opérateur unaire :
  - ! : **NON** logique (*NOT* en anglais)
- Opérateurs binaires :
  - && : **ET** logique (*AND* en anglais)

    vrai $\Leftrightarrow$ les deux opérandes sont vrais
  - || : **OU** logique (*OR* en anglais)

    vrai $\Leftrightarrow$ un au moins des deux opérandes est vrai

Les expressions booléennes

- && : table de vérité

(ET)	**true**	**false**
true	**true**	**false**
false	**false**	**false**

- Particularité : si l'opérande de gauche est **faux**, l'opérande droit **ne sera pas évalué** et le résultat sera **false**

Les expressions booléennes

- || : table de vérité

(OU)	true	false
true	true	true
false	true	false

- Particularité : si l'opérande de gauche est **vrai**, l'opérande droit **ne sera pas évalué** et le résultat sera **true**

Tableau des priorités et associativités

- (opérateurs déjà vus)

priorité	opérateur	associativité
grande	$-$, $+$ unaires, $!$	$\Longleftarrow$
	$*$, $/$, $\%$	$\Longrightarrow$
	$-$, $+$ binaires	$\Longrightarrow$
	$\&\&$	$\Longrightarrow$
faible	$\|\|$	$\Longrightarrow$

Les expressions booléennes

- Exemples :
  - **true** && **false** || **true**
    - → (**true** && **false**) || **true**
    - → **false** || **true**
    - → **true**
  - **false** || **false** && ! **true**
    - → **false** || (**false** && ! **true**)
    - → **false** || (**false** && **false**)
    - →**false** || **false**
    - → **false**

Les expressions booléennes

- Exemples :
  - **true || false && true**
    → **true || (false && true)**
    → **true**
  - **false && (true || false)**
    → **false**
  - **!(true || false) && true**
    → **!true && true**
    → **false && true**
    → **false**

Résumé

- Voici les règles qui résument tout ceci

Expression :
    ConditionalOrExpression

ConditionalOrExpression :
    ConditionalAndExpression
    ConditionalOrExpression **||** *ConditionalAndExpression*

ConditionalAndExpression :
    AdditiveExpression
    ConditionalAndExpression **&&** *AdditiveExpression*

(. . .)

Résumé

AdditiveExpression :

MultiplicativeExpression

AdditiveExpression **+** *MultiplicativeExpression*

AdditiveExpression **-** *MultiplicativeExpression*

MultiplicativeExpression :

UnaryExpression

MultiplicativeExpression ***** *UnaryExpression*

MultiplicativeExpression **/** *UnaryExpression*

MultiplicativeExpression **%** *UnaryExpression*

(. . .)

Résumé

UnaryExpression :

+ *UnaryExpression*

- *UnaryExpression*

! *UnaryExpression*

Literal

(*Expression* **)**

Identifi er

- Ces règles de production font clairement apparaitre la priorité et l'associativité des opérateurs
- Cependant pas suffisantes pour intégrer toutes les contraintes additionelles que nous avons décrites

Résumé

- Exercices : ces expressions sont elles correctes avec les contraintes décrites à ce jour ? Si non, pourquoi ? Si oui, quelle valeur ?
  - "turlu" + "tutu"
  - "turlu" + '2'
  - 1 + 1.2
  - **true** || **false** && **false** || **true**
  - 1 + 2 || 2 + 1

Lien avec la logique

- Avertissement
- L'assignation
- Les expressions relationnelles
- Structures alternatives
- Structures répétitives

Avertissement

- Notions utiles pour coder les exercices vus en logique
- Permet d'être prêts pour les laboratoires
- Superficiel mais on y reviendra

L'assignation

=

- Permet de réinitialiser le contenu d'une variable au cours de l'exécution du programme
- Forme générale

Assignment :
Identifier **=** *Expression* **;**

- A ne pas confondre avec l'opérateur d'égalité utilisé en mathématique
  - Exemple : `i = i + 1;` est valide

L'assignation

- *Identifier* : nom de variable préalablement déclarée et toujours en vie
- *Expression*
  - **Type** : même que celui de la variable (cette contrainte sera ultérieurement relachée)
  - **Valeur** : va écraser l'ancienne valeur contenue dans la variable

Les expressions relationnelles

- Composées d'opérateurs permettant de ***comparer*** des valeurs
- Ont une **valeur booléenne**
- Abondamment utilisées dans les conditions des structures alternatives et répétitives

Les expressions relationnelles

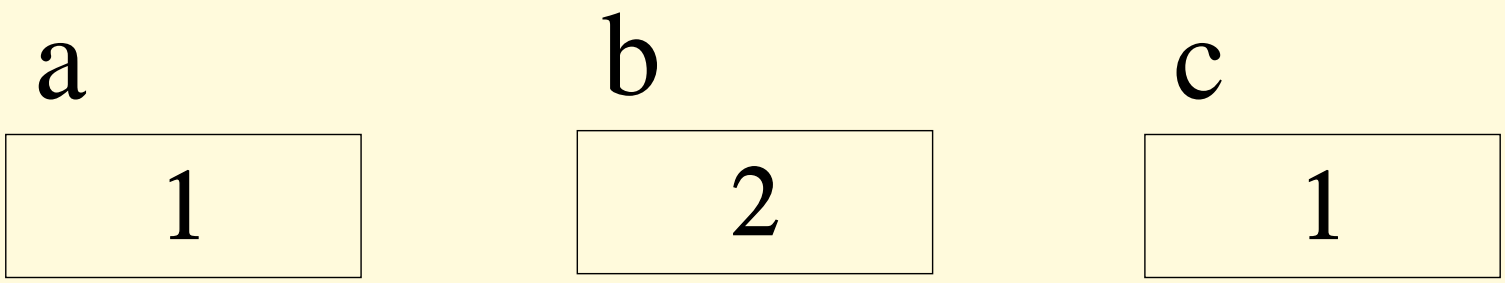
- On distingue
  - Les opérateurs d'**égalité** : == !=
    - a == b : vrai si a **égale** b
    - a != b : vrai si a est **différent** de b
  - Les opérateurs de **comparaison** : < > <= >=
    - a < b : vrai si a est **plus petit** que b
    - a > b : vrai si a est **plus grand** que b
    - a <= b : vrai si a est **plus petit ou égal** à b
    - a >= b : vrai si a est **plus grand ou égal** à b

Les opérateurs d'égalité

- Les deux opérandes doivent être de **même type**
  (on relachera cette contrainte un peu plus tard)
- Le **résultat** est du type **boolean**
- S'appliquent à **tous les types**
- **Attention** : le sens est différent pour un
  - type **primitif** : les valeurs sont comparées
  - type **référence** : les *références* sont comparées

Les opérateurs d'égalité

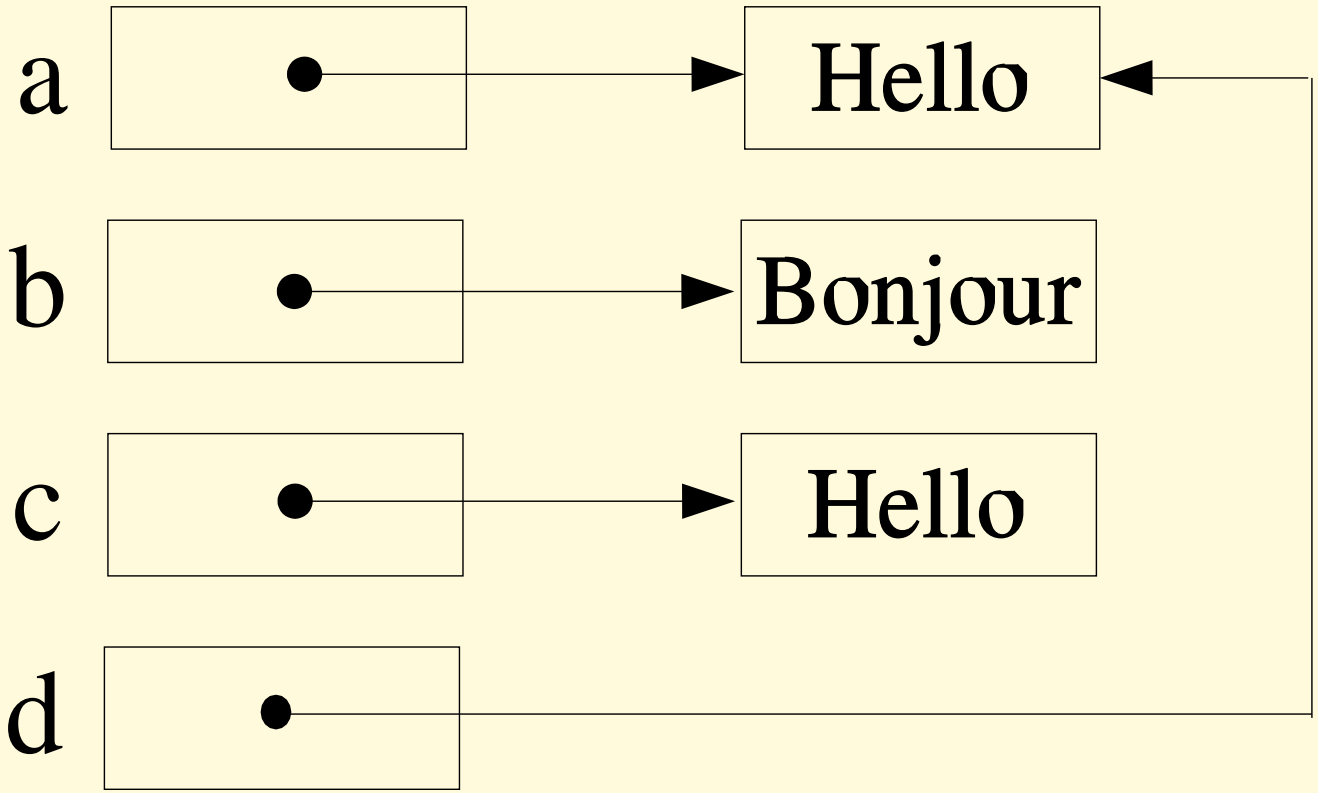
- Egalité des types primitifs



- On a : a==c mais a!=b et b!=c

Les opérateurs d'égalité

- Egalité des types références



- On a : a!=b, a!=c mais a==d

Les opérateurs d'égalité

- Particularité du type String

```
{
   String  s1 = "Hello";
   String  s2 = "Hello";
   System.out.println(s1==s2);
}
```

- Ecrit **true**
- Pourquoi ?
  - Le compilateur se rend compte que le texte est identique et **réutilise** le même espace mémoire

Les opérateurs d'égalité

- Particularité du type String

```
{
   String  s1 = "Hello";
   String  s2 = "Hel";
   s2 = s2 + "lo";
   System.out.println(s1==s2);
}
```

- Ecrit **false**
- Pourquoi ?
  - L'égalité n'apparaît que lors de l'exécution
  - La machine virtuelle n'économise pas cet espace

Les opérateurs de comparaison

- Les deux opérandes doivent être de **même type**
  (on relachera cette contrainte un peu plus tard)
- Le **résultat** est du type **boolean**
- S'appliquent aux **types numériques** uniquement
  - Exemple : **true** < **false** n'est pas accepté
  - Exemple : "Absolu" < " Relatif " non plus
    (on pourra comparer l'ordre lexicographique de 2 String mais par une autre écriture)

Tableau des priorités et associativités

- (opérateurs déjà vus)

priorité	opérateur	associativité
grande	−, + unaires, !	⟸
	∗, /, %	⟹
	−, + binaires	⟹
	<, >, <=, >=	⟹
	==, !=	⟹
	&&	⟹
faible	||	⟹

Priorité et associativité

- Exemples
  - 1+3==2+2 est compris comme (1+3)==(2+2)
    $\longrightarrow$ 4==4 $\longrightarrow$ **true**
  - a<b==c<d est compris comme (a<b)==(c<d)
- **Attention** : ce sont des **opérateurs binaires**
  - 1==1==1 est incorrect car compris comme
    (1==1)==1 $\longrightarrow$ **true**==1 (erreur : types différents)
  - 1<2<3 est incorrect car compris comme
    (1<2)<3 $\longrightarrow$ **true**<3 (erreur : types différents)

Les instructions de choix

- L'instruction *Si-Alors* (*If-Then Statement*)
- Permet de **conditionner l'exécution d'une instruction** à la vérification d'une condition
- Forme générale

IfThenStatement :
`if (` *Expression* `)` *Statement*

- *Expression* : à valeur booléenne

Les instructions de choix

- Exemple

```
package be.heb.esi.lg1.cours;
public class Test
{
   public static  void main(String[]  args)
   {
      int  nb = −15;
      if  (nb < 0)     //  prend la  valeur  absolue de nb
         nb = −nb;
   }
}
```

- Remarquer l'indentation !

Les instructions de choix

- Exemple

```
package be.heb.esi.lg1.cours;
public class Test
{
   public static  void main(String[]  args)
   {
         int  âge = 35;
         if  (âge > 25 && âge < 60)
            System.out.println(" Tarif ␣Senior");
   }
}
```

- On ne pourrait pas écrire 25<age<60

Les instructions de choix

- Exemple

```
package be.heb.esi.lg1.cours;
public class Test
{
   public static  void main(String[]  args)
   {
         int  âge = 35;
         if  (âge > 25)
            if  (âge < 60)
               System.out.println(" Tarif ␣Senior");
   }
}
```

- equivalent car If-Then est aussi une instruction

Les instructions de choix

- Si il y a plusieurs instructions soumises à la même condition, on les met dans un **bloc** qui est aussi une instruction

```
if (condition)
{
    instruction_1
     ...
    instruction_n
}
```

Les instructions de choix

- Exemple

```
public class Test
{
   public static  void main(String[]  args)
   {
         double tarif=0.0;
         int  âge = 35;
         if  (âge > 25 && âge < 60)
         {
            tarif   = 7.50;
           System.out.println(" Tarif ␣Senior␣=␣" + tarif );
         }
   }
}
```

Les instructions de choix

- Est bien différent de

```
public class Test
{
   public static  void main(String[] args)
   {
         double tarif=0.0;
         int  âge = 35;
         if  (âge > 25 && âge < 60)
            tarif  = 7.50;
           System.out.println(" Tarif ␣Senior␣=␣" + tarif );
   }
}
```

- Indentation pas significative pour le compilateur

Les instructions de choix

- L'instruction *Si-Alors-Sinon* (*If-Then-Else Statement*)
- Permet d'exécuter deux instructions différentes en fonction d'une condition
- Forme générale

IfThenElseStatement :
`if (` *Expression* **`)`** *Statement* **`else`** *Statement*

- *Expression* : à valeur booléenne

Les instructions de choix

- Exemple

```
package be.heb.esi.lg1.cours;
public class Test
{
   public static void main(String[] args)
   {
      int entier1 = 35;
      int entier2;
      // entier2 prend la valeur absolue de entier1
      if (entier1 < 0)
         entier2 = -entier1;
      else
         entier2 = entier1;
   }
}
```

Les instructions de choix

- Si il y a plusieurs instructions soumises à condition, on les met dans un **bloc**

```
if (condition)
{
    instruction_1
     ...
    instruction_m
}
else
{
    instruction_1
     ...
    instruction_n
}
```

Les instructions de choix

```
public class Test
{
   public static  void main(String[] args)
   {
      int  age = 35;
      fbat   tarif =0.0;
      if  (age <= 25)
      {
         tarif  = 5.00f;
         System.out.println(" Tarif ␣Etudiant␣=␣" + tarif );
      }
      else
      {
         tarif  = 7.5f;
         System.out.println(" Tarif ␣Senior␣=␣" + tarif );
      }
   }
}
```

Problème du dangling else

- Que va imprimer le code suivant ?

```
public class Test
{
   public static void main(String[] args)
   {
      int entier1 = 15, entier2 = 18, entier3 = 14;

      if (entier1 > entier2)
         if (entier1 < entier3)
            System.out.println("entier1 < entier3");
      else
         System.out.println("entier1 < entier2");
   }
}
```

Problème du dangling else

- Rien ! Pourquoi ?
- **Règle** : le **else** appartient au premier **if** rencontré en remontant le code
- L'indentation n'y joue aucun rôle
- Contournement : Utiliser les blocs

Problème du dangling else

```
package be.heb.esi.lg1.cours;
public class Test
{
   public static void main(String[] args)
   {
      int entier1 = 15, entier2 = 18, entier3 = 14;
      if (entier1 > entier2)
      {
         if (entier1 < entier3)
            System.out.println("entier1 < entier3");
      }
      else
         System.out.println("entier1 < entier2");
   }
}
```

Exercice récapitulatif

Ecrire en Java un programme qui calcule le maximum de 3 entiers

Les instructions de choix

- Il existe également une instruction proche du *Selon que*
- Nous la verrons plus tard

Les instructions répétitives

- L'instruction *Tant-que* (*While Statement*)
- Ce qui correspond au *Tant que* de la logique
- Permet de **répéter une instruction** tant qu'une condition est vérifiée
- Forme générale

WhileStatement :
`while (` *Expression* **`)`** *Statement*

- *Expression* : à valeur booléenne

Exemple 1

```
package be.heb.esi.lg1.cours;
public class Test
{
   public static  void main(String[]  args)
   {
      int  i  = 1;
      while (i  < 10)
      {
         System.out.print( i );
         i  =  i  + 1;
      }
   }
}  //  Imprime: 123456789
```

Exemple 2

```
package be.heb.esi.lg1.cours;
public class Test
{
   public static  void main(String[]  args)
   {
      int  i  = 1;
      while (i  < 10)
      {
         i  =  i  + 1;
         System.out.print( i );
      }
   }
}  //  Imprime: 2345678910
```

Exemple 3

```
package be.heb.esi.lg1.cours;
public class Test
{
   public static  void main(String[]  args)
   {
      int  i  = 1;
      while (i   < 10 ||   i % 2 == 0)
      {
         i  =  i  + 1;
         System.out.print( i );
      }
   }
} //  Imprime: 234567891011
```

Les instructions répétitives

- Remarque : si la **condition** du *while* n'est **pas modifiée** dans le corps du *while* alors elle aura toujours la même valeur
  - si elle est **fausse on n'entre pas** dans la boucle
  - si elle est **vraie** on entre dans la boucle et on en sort plus : **on cycle**

```
while (true) System.out.println("Je␣cycle␣!");
```

Les instructions répétitives

- L'instruction *Pour* (*For Statement*)
- Correspond (plus ou moins) au *Pour* de la logique
- En logique, on écrit

```
Pour i = début à fin par pas faire
  instruction
Fin pour
```

- En **Java**, cela donne

```
for ( int i = début; i <= fin ; i = i + pas)
    System.out.print( i );
```

Exemple 1

```
package be.heb.esi.lg1.cours;
public class Test
{
   public static  void main(String[]  args)
   {
      int  debut = 1;
      int  fin  = 10;
      int  pas = 1;
      for  (  int  i  =  début;  i <=  fin ;  i  =  i  +  pas)
         System.out.print(i );
   }
} //  Imprime: 12345678910
```

Exemple 2

```
package be.heb.esi.lg1.cours;
public class Test
{
   public static  void main(String[]  args)
   {
      int  i ;
      int  nb = 3;
      for( i  = 1;  i  <= 10;  i  =  i  + 1)
      {
         System.out.print(nb);
         System.out.print(" X " + i + " = ");
         System.out.println(nb * i );
      }
   }
}
```

Exemple 2

Ecrit

```
3 X 1 = 3
3 X 2 = 6
3 X 3 = 9
3 X 4 = 12
3 X 5 = 15
3 X 6 = 18
3 X 7 = 21
3 X 8 = 24
3 X 9 = 27
3 X 10 = 30
```

for et while

- Quand utilise-t-on un **while** plutôt qu'un **for**
  - Le **for** intègre dans son entête l'initialisation du compteur, le test et l'incrémentation
    - Utilisé lorsque ces trois étapes sont simples et que les variables début, fin et pas sont connues avant le début de la boucle
    - **Accroit la lisibilité** du programme
  - On utilisera un **while** dans les autres cas

Lire au clavier

- **Java** ne fournit **pas de mécanisme simple** pour lire au clavier (entrée standard)
- Pourquoi ?
  - **Java** est un langage moderne pour des **applications modernes** (graphiques)
  - Les lectures se feront dans des **champs de saisie graphiques**
  - Ecrire s'utilise plus fréquemment, notamment en phase de test $\longrightarrow$ simplicité pour écrire à la console

Lire au clavier

- Toutefois : **`Java`** permet de lire au clavier et fournit même un mécanisme riche et puissant pour ce faire (détaillé plus tard)
- En attendant, on fournit une classe qui permet de lire au clavier tout en **cachant la complexité** du mécanisme
  - Package : `be.heb.esi.lg1.util`, classe : `Lire`

Exemple

```
package be.heb.esi.lg1.cours;
import be.heb.esi.lg1. util .Lire ;
public class Test
{
   public static  void main(String[] args)
   {
      int  nb;
      nb = Lire .intData ();      //  On a aussi double , ...
      if  (nb % 2 == 0)
            System.out.println("Ce␣nombre␣est␣pair");
      else
            System.out.println("Ce␣nombre␣est␣impair");
   }
}
```

Lire au clavier

- Ce programme ne peut fonctionner chez vous que si vous installez la classe *Lire*
- Nous vous expliquerons comment faire lors des laboratoires
- La classe est disponible sur le site
- Avec une documentation
  (au format **`HTML`**, générée par **`javadoc`**)

Les conversions

- Présentation
- Les groupes de conversions
- Les contextes de conversions

Présentation

- Les valeurs des expressions ne sont pas toujours du type requis pour pouvoir être utilisées
- Peuvent être **explicitement converties** en un autre type par le programmeur : **casting**
- Ou **implicitement** par le compilateur : **conversion** et **promotion**
- Règle générale : toute modification devrait être explicitée par le programmeur

Présentation

- **6 groupes** (sortes) de conversions
  - Elargissante de type primitif
  - Arrondissante de type primitif
  - Elargissante de type référence
  - Arrondissante de type référence
  - Conversion de chaîne de caractères
  - Conversion identique
- Apparaissent dans **5 contextes** différents

Conversion élargissante de type primitif

(*widening primitive conversion*)

- Vers un type plus général **sans perte d'information**

byte

↪ **short** **char**

↪ **int** ↩

↪ **long**

↪ **float**

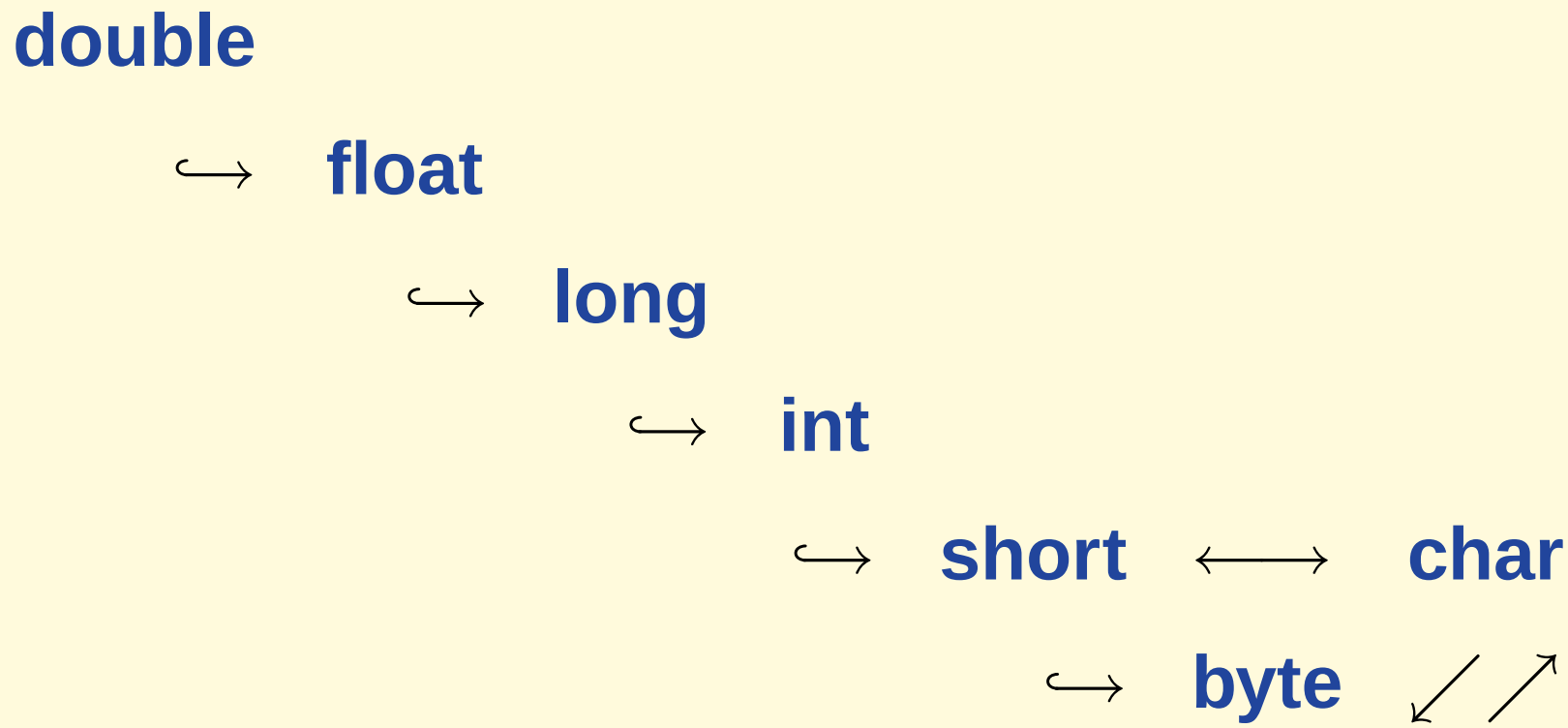
↪ **double**

Conversion élargissante de type primitif

- Pourquoi le **char** est-il converti en **int** ?
- Le passage **entier vers réel** peut entraîner une **perte de précision** (pas d'erreur générée)

Conversion arrondissante de type primitif

(*narrowing primitive conversion*)



Conversion arrondissante de type primitif

- On retrouve toutes les conversions qui ne sont pas élargissantes
- Il peut y avoir **perte d'information** et/ou de précision (aucune exception à l'exécution)
- Passage de **float** vers **long** : on perd les décimales (pas d'arrondi)

Conversion de type référence

- Conversion de **type référence**
  - Elargissante
  - ou arrondissante
- Seront vues plus tard

Conversion de chaîne de caractères

- Toute valeur de tout type peut être convertie en chaîne de caractères
- Même pour les types définis par l'utilisateur
- La chaîne représentera au mieux la valeur

Conversion identique

- Tout type peut être **transformé en lui même**
- Permet essentiellement d'expliciter le type d'une valeur

Les contextes de conversions

- 5 contextes de conversions
  - **La promotion numérique** (*numeric promotion*)
  - **L'assignation** (*assignment conversion*)
  - **L'appel de méthode** (*method invocation conversion*) (plus tard dans le cours)
  - **La chaîne de caractères** (*String conversion*)
  - **Le *casting*** (*casting conversion*)
- A chaque contexte : un ou plusieurs types de conversions

La promotion numérique

- Apparaît dans les expressions
- Adapte le type des opérandes à ceux attendus par l'opérateur
- Permet la conversion
  - identique
  - élargissante

La promotion numérique

- Opérateur unaire : **byte**, **short**, **char** $\longrightarrow$ **int**
- Exemples

```
short s = 1;
byte b = 2;
char c = '3';
 ... s ... // type short
 ... + s ... // s converti implicitement en int
 ... – b ... // b converti implicitement en int
 ... + c ... // c converti implicitement en int
 ... – 'A' ... // 'A' converti implicitement en int
```

La promotion numérique

- Opérateur binaire :
  - Si le type le plus large est **double**, **float**, **long** $\longrightarrow$ on élargit le deuxième opérande au même niveau
  - Sinon on élargit les deux opérandes à **int**
- Exemples

```
short s = 1;
char c = '3';
int  i  = 4;
 ... s+i  ...   // s converti implicitement en int
 ... s+5L ...   // s converti implicitement en long
 ... c-s  ...   // c et s convertis implicitement en int
... 1-1L  ...   // 1 converti implicitement en long
```

La promotion numérique

- Exemples

```
System.out.println (7/2)        // imprime 3
System.out.println (7./2)       // imprime 3.5
System.out.println (7/2.)       // imprime 3.5
System.out.println  (7./2.)     // imprime 3.5
System.out.println(7+2)         // imprime 9
System.out.println(7.+2)        // imprime 9.0
```

La conversion à l'assignation

- Adapte le type de l'expression au type de la variable
- Permet la conversion
  - identique
  - élargissante
  - arrondissante (sous certaines conditions)

La conversion à l'assignation

- Conversion arrondissante lors de l'assignation
  - La variable est de type **byte**, **short**, ou **char**
  - L'expression est
    - constante
    - de type **byte**, **short**, **char** ou **int**
    - sa valeur est représentable dans le type de la variable
  - Si ce n'est pas vérifié, une erreur est générée à la compilation

La conversion à l'assignation

- Exemples

```
long l1 = 12; // élargissante
long l2 = 'a'+1; // élargissante
short s1 = 12; // arrondi int -> short
short s2 = 1+2; // idem
byte b1 = 123245; // erreur compilation (trop grand)
byte b2 = s1+1; // erreur compilation (pas constant)
byte b3 = 21L; // erreur compilation (long pas admis)
```

Le casting

- C’est la **conversion explicite**
- Valeur convertie dans le type explicité par le casting

Casting :
(*Type* **)** *Expression*

- Autorise toutes les conversions vues sauf la conversion de chaînes de caractères
- Grammaticalement, c’est un opérateur

Tableau des priorités et associativités

- (opérateurs déjà vus)

priorité	opérateur	associativité
grande	(), −, + unaires, !	⟸
	∗, /, %	⟹
	−, + binaires	⟹
	<, >, <=, >=	⟹
	==, !=	⟹
	&&	⟹
faible	||	⟹

Le casting

- Exemples

```
int  entier  = ( int ) 5;  // conversion identique

int  entier  = ( int ) 5 l ;  // conversion arrondissante

int  entier  = ( int) 120000000000000000000l;
     //  erreur  à  la  compilation  (integer  number too large)
     //  Le nombre n'est pas représentable en tant que long

int  entier  = ( int) 1200000000000L;
     //  Accepté mais valeur de entier  abérrante
```

Le casting

```
double réel = (double) 12; //conversion élargissante

String  mot = (String ) 12;
      //  erreur  à  la  compilation  :
      //  la  conversion de chaîne n'est pas admise par casting

String  mot = (String )  "mot";  //  OK: conversion identique

boolean b = (boolean) 1;
      //  erreur  compilation  :  pas de conversion vers boolean
```

Le casting

```
int  entier  = (byte) 500−400;
     // 500 converti  en byte −> accepté mais résultat inattendu
     //  ()   prioritaire   par rapport à −

byte entier  = (byte) 500−400;
     //  résultat  du calcul  int  ne peut être  converti  en byte

byte entier  = (byte) (500−400); // OK

int  entier  = (byte) (190−(byte)100);
     //  100 :  conversion arrondissante en byte (casting)
     //  puis  promu en int pour le  calcul
     //  90 :  conversion arrondissante en byte (casting)
     //  puis  promu en int pour l'assignation
```

Le contexte des chaînes de caractères

- Opérateur + avec un opérande de type String

  $\Rightarrow$ l'autre opérande converti en String

- Exemples

```
System.out.println("1+1␣=␣"+2);
String  s = "Pi␣=␣" + 3.1415;
```

- Que donnera ceci ?

```
System.out.println("1"+2+3);
System.out.println(1+2+"3");
```

Récapitulatif

- Récapitulons les conversions et leur contexte

	élargis.	arrondi	chaîne	identique
promotion num.	√			√
assignation	√	√ (*)		√
chaîne			√	√
casting	√	√		√

(*) : sous certaines conditions

Récapitulatif

- Que se passe-t-il avec le programme suivant ?

```
public class Test
{
   public static  void main(String [ ]  args)
   {
      System.out.println ( ( double) 3/2 );
      System.out.println ( ( double ) (3/2) );
      int  i1  = ( double) 3/2;
      int  i2  = ( int ) (( double) 3/2);
      System.out.println ( "i1=" + i1  + ",␣i2=" + i2 );
      System.out.println ( "i1+i2=" + i1  + i2 );
   }
}
```

Les tableaux

- Présentation
- Déclaration
- Création
- Initialisation
- Accès aux éléments
- Assignation en bloc
- Taille

Présentation

- Nécessité de pouvoir manipuler un **nombre de données supérieur à un**
  - Ex : toutes les factures d'un client déterminé
- Nombre de données
  - **Statique** : constante à la compilation (prévoir un tableau suffisament grand pour tous les cas)
  - **Dynamique** : valeur déterminée à l'exécution
- Cours de logique : uniquement tableaux statiques
- **Java** gère les tableaux dynamiques

Présentation

- Un tableau contient un nombre **déterminé** de composants (éléments) de **même type**
- Si éléments de type T $\Rightarrow$ type du tableau = T[]
- Exemples
  - **int** [] est le type *tableau d'entiers*
  - **char**[] est le type *tableau de caractères* (différent de String)
  - String [] est le type *tableau de chaînes de caractères*

Présentation

- Nombre d'élements = **taille** (de 0 à $+\infty$)
  (on dit aussi **longueur**)
- La **taille** du tableau ne fait **pas partie du type**
- Un tableau est de type **référence**
- Exemple : t de type **int** [] de taille 3

Tableau à plusieurs dimensions

- Le type *tableau* peut être le type des éléments d'un tableau
- Permet d'avoir l'équivalent de tableaux à plusieurs dimensions
- Exemples
  - **int** [][] est un tableau de tableaux d'entiers
  - **int** [][][] est un tableau d'entiers de dimension 3
- **Dimension** = nombre de couples de crochets ouvrant-fermant apparaissant dans son type

Tableau à plusieurs dimensions

- Exemple : t un tableau de 3 tableaux de 2 entiers (type : **int** [][])

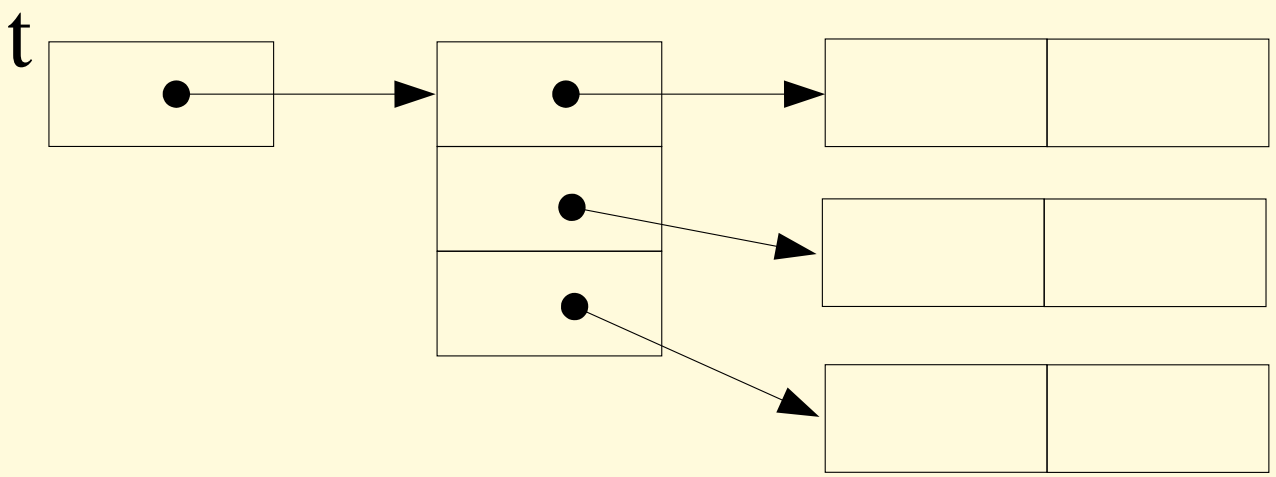
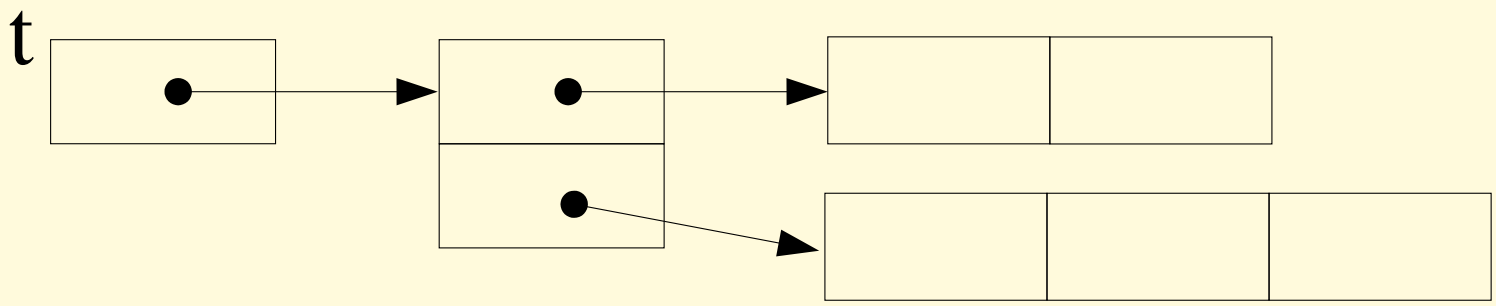


Tableau à plusieurs dimensions

- **Différence avec les tableaux de logique ?**
  - Chaque élément d'un tableau à deux dimensions est un tableau indépendent
  - La taille ne fait pas partie du type
  - $\Rightarrow$ Chaque élément peut être d'une **taille différente**



Déclaration

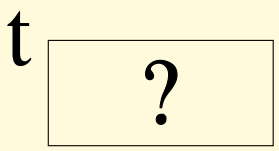
- Démarche **similaire aux autres déclarations** de variables vues à ce jour
- Exemples
  - `int [] entiers ;`

    entiers pourra contenir une référence vers un tableau d'entier
  - `short [][] shortss;`

    shortss pourra contenir une référence vers un tableau de tableaux d'entiers courts

Déclaration

- Un style ***C++****-like* existe aussi et sera vu plus tard
- La déclaration ne **réserve** que la place nécessaire à une **référence**
- Exemple : **int** [] t



Création dynamique

- Avant de pouvoir manipuler le tableau, il faut le **créer**
- Cela se fait par le mot clé **new**
- Exemple :

```
int [] t;
t = new int[3]; // crée un tableau de 3 entiers
```

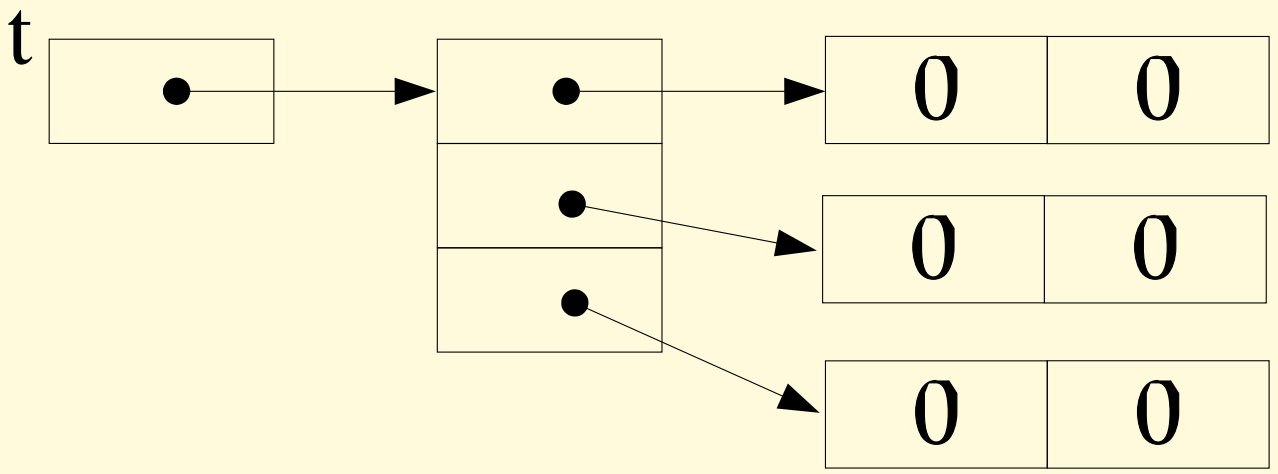
Création dynamique et valeur par défaut

- Lors de la création d'un tableau, les éléments sont initialisés à une **valeur par défaut**
  - Numérique : 0
  - Booléen : **false**
  - Référence : **null** (**référence vers *rien***)
- On peut aussi créer à la déclaration
  - Exemple : `int [] t = new int[3];`

Création dynamique

- Idem pour les tableaux à plusieurs dimensions
- Exemple :

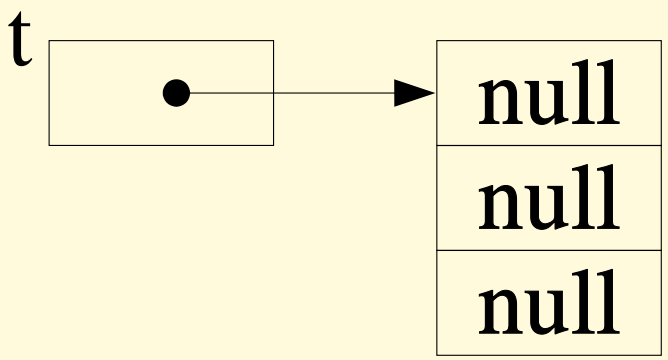
```
int [][] t;
t = new int[3][2];
```



Création dynamique

- On peut omettre les dernières tailles
- Le tableau est créé en partie
- Il faut au moins la première
- Exemple :

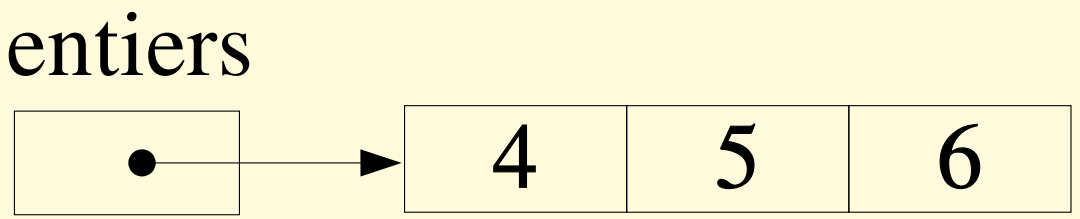
```
int [][] t;
t = new int [3][];
```



Initialisation

- On peut aussi créer le tableau en **donnant (toutes) ses valeurs**
- Dans ce cas, on ne spécifie pas les tailles (elles sont déduites)
- Exemple :

```
int []  entiers ;
entiers  =  new int [] {4,5,6};
```



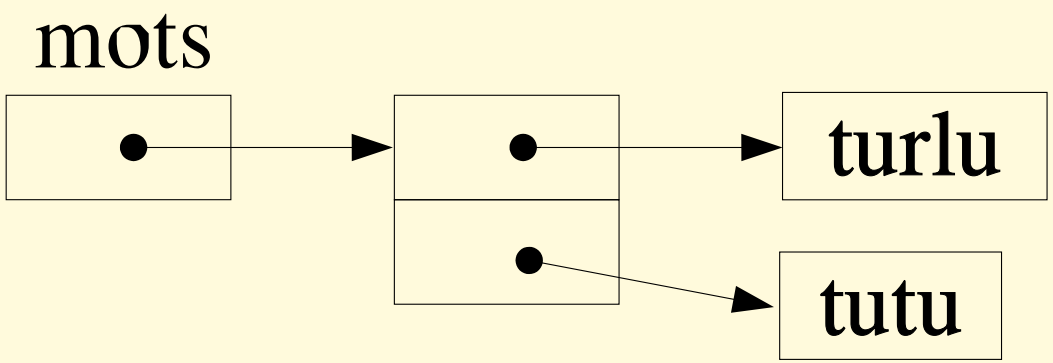
Initialisation

- Si les valeurs sont données à la déclaration, on peut simplifier l'écriture

```
int [] t1 = new int [] {4,5,6};   // OK
int [] t2 = {4,5,6};   // Ecriture abrégée acceptée
int [] t3;
t3 = {4,5,6};   // Erreur à la compilation
```

Initialisation

- Exemple : String [] mots = {"turlu","tutu"};
  - mots contient une référence vers un tableau de String de longeur 2



Initialisation

- Syntaxe : initalisation de tableau (*array*)

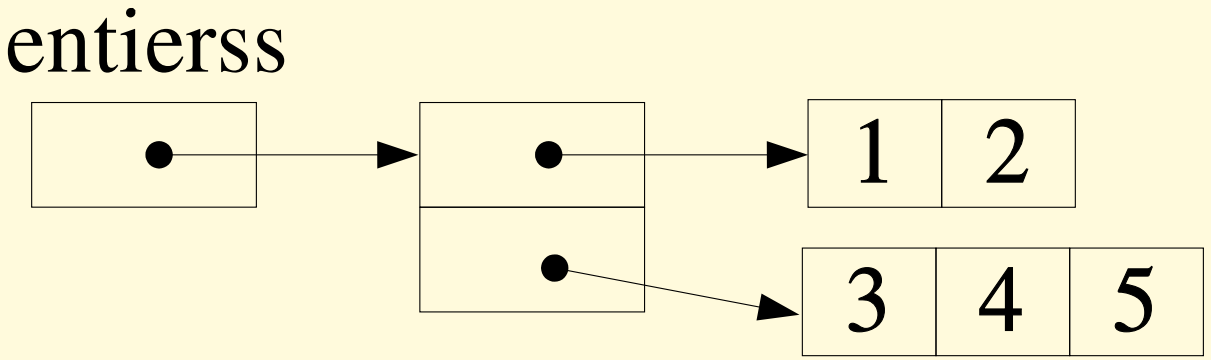
ArrayInitializer :
  { *VariableInitializers*(opt) **,**(opt) **}**

VariableInitializers :
  VariableInitializer
  VariableInitializers **,** *VariableInitializer*

VariableInitializer :
  Expression
  ArrayInitializer

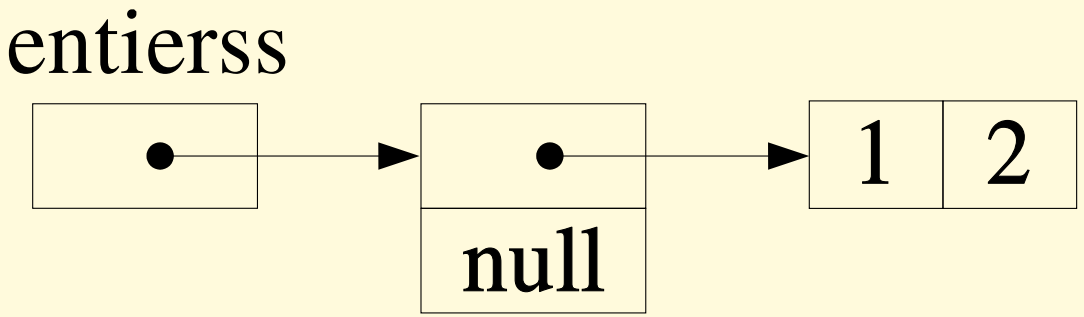
Initialisation

- Remarques
  - La liste des initialisations peut-être terminée par une virgule : sans effet (et sans intérêt)
  - Un initialisateur de tableau peut en contenir un autre : pour les tableaux de tableaux
- Exemple : `int [][] entierss = {{1,2},{3,4,5}};`



Initialisation

- Exemple : `int [][] entierss = {{1,2}, null};`



- Autre écriture

```
int [][] entierss;
entierss = new int [][] {{1,2}, null};
```

Création

ArrayCreationExpression :

new *TypeName DimExprs Dims*(opt)

new *TypeName Dims ArrayInitializer*

DimExprs :

DimExpr

DimExprs DimExpr

DimExpr :

[*Expression* **]**

Dims :

[ ]

Dims **[ ]**

Création

- On ne peut spécifier les valeurs que si on ne donne aucune taille
- Le nombre de dimensions doit correspondre au type
- Exemples :
  - `int [][] t = new int[2];` est incorrect
  - `int [][] t = new int [] {1,2};` aussi
  - `int [][] t = new int [] {{1},{2}};` aussi
  - `int [][] t = new int [3][2] {1,2};` aussi
  - `int [][] t = new int [3][2] { null , null };` aussi

Création

- La taille doit être un **int**
- Eventuellement après une promotion numérique
- Donc pas de **long** : pas de tableau de plus de 5 milliards d'éléments ;-)
- La **taille** ne peut **pas** être **négative**
  - sinon une exception est lancée
  - pas vérifié à la compilation même si constante

Création

- La taille peut être **nulle**
  - Exemple : `int [][] t = new int [0][2];`
  - Peut avoir un sens comme cas limite
- La taille est une expression générale (pas forçément une constante connue à la compilation)
- Exemple

```
int taille = Lire.intData();
int [] t2 = new int[ taille ];
```

Accès aux éléments

- Composants indicés de **0 → taille du tableau - 1**
- **On ne peut pas choisir** comme en logique
- Accès en faisant suivre le nom du tableau par une **expression entière entourée de crochets**
- Exemple :

```
int [] entiers = {7,14,0};
int entier = entiers [0]; // entier vaut 7
```

Accès aux éléments

- Exemple :

```
int []  entiers  = {7,14,0};
entiers [1] = 85;
int  entier  =  entiers [1];  // entier  vaut 85
```

- Exemple :

```
int  [][]  entierss  = {{1,2},{3,4,5}};
int []  entiers  =  entierss [1];
    //  entiers  :  référence vers  le  tableau d'entiers  {3,4,5}
int  entier  =  entiers [1];  //  entier  vaut 4
```

Accès aux éléments

- Si l'indice n'est pas valide (par rapport à la taille du tableau), la **JVM** lance une exception (**IndexOutOfBoundException**)
- Exemple :

```
int [] entiers = {7,14,0};
int i1 = entiers[-1]; // erreur à l'exécution
int i2 = entiers [3]; // erreur à l'exécution
int i3 = entiers [0.0]; // erreur à la compilation
```

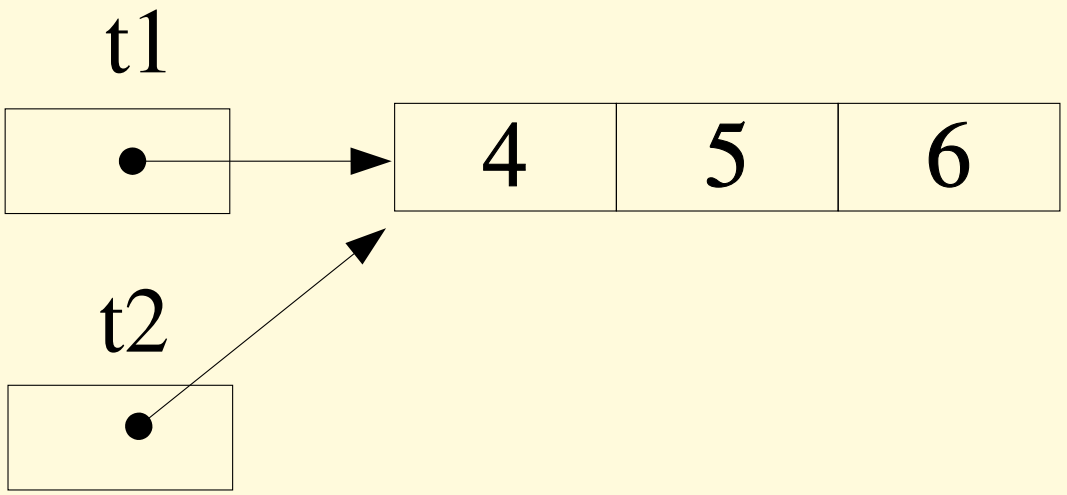
Accès aux éléments

- La **JVM** lance aussi une exception si le tableau est à **null** (**NullPointerException**)
- Exemple :

```
int [][] entierss = {{1,2}, null};
int [] entiers = entierss [1]; // OK
int entier = entierss [1][0]; // erreur à l'exécution
```

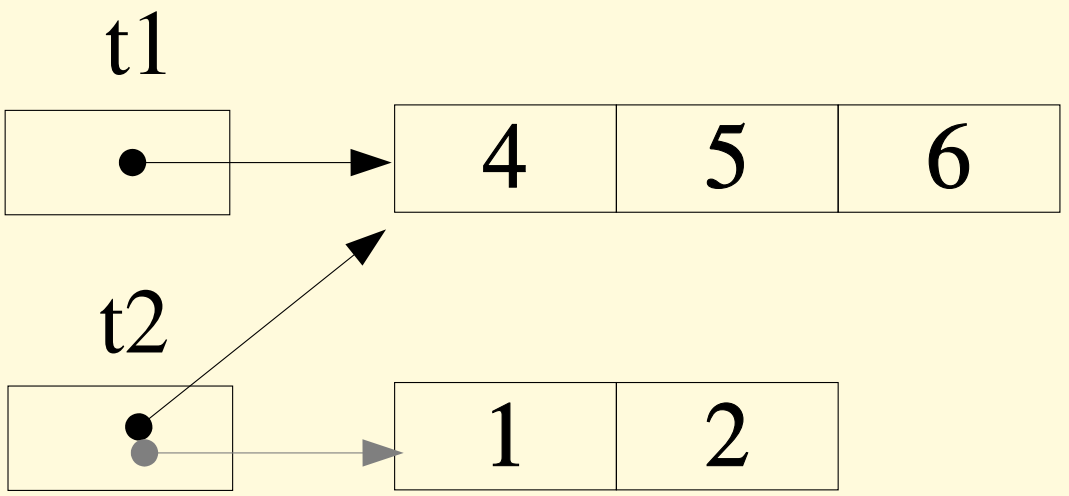
Assignation en bloc

- Un tableau est de type référence
- **L'assignation** d'un tableau à un autre **copie la référence** (et pas le tableau)
- Exemple : `int [] t1 = {4,5,6}, t2 = t1;`



Assignation en bloc

- Exemple : **int** [] t1 = {4,5,6}, t2 = {1,2}; t2 = t1;



- L'ancien tableau n'est plus référencé
- La place mémoire sera récupérée si nécessaire (par le *garbage collector*)

Assignation en bloc

- Comme dans toute assignation, **il faut que les types correspondent**
- Exemple :

```
int []  t1  = {4,5,6};
boolean[] t2 = t1 ;  //  erreur  à  la  compilation
```

- Exemple :

```
int  [][]  t1   = {{4,5,6},{1,2}};
int []  t2  =  t1 [0];   //  t2 ={4,5,6};
```

Assignation en bloc

- **Pas de conversion implicite** pour tout le tableau
- Exemple :

```
short s = 1;
int  i = s; // OK
short[] ss  = {4,5,6};
int []  is = ss; // erreur à la compilation
int  i2 = ss [1]; // OK
```

Taille

- La taille du tableau s'obtient par :

  nom_tableau.length

- Exemple :

```
public class Test
{
   public static  void main(String [ ]  args)
   {
      int []  entiers   = {4,5,6};
      int   taille   =  entiers .length;
      System.out.println( taille  );  //   écrit  3
   }
}
```

Taille

- Si plusieurs dimensions
  - Taille (potentiellement) différente d'un élément à l'autre
  - $\Rightarrow$ Spécifier l'élément
  - Exemple :

```
int [][] t = {{1,2},{2,3,4}};
System.out.println( "Nombre␣d'éléments␣=␣"
      + ( t [0]. length + t [1]. length ) );
```

Exemple

- Initialisation des composants d'un tableau d'entiers à la valeur de leur indice

```
package be.heb.esi.lg1.tutorials.tableaux;

public class InitialisationTableau {
    public static void main(String[] args){
        int [] tableau = new int[10];
        for(int i = 0; i < tableau.length; i = i + 1)
            tableau[i] = i;
    }
}
```

Exemple

- Parcours des composants d'un tableau d'entiers suivant l'ordre ascendant des indices

```
package be.heb.esi.lg1.tutorials.tableaux;

public class SimpleParcoursAscendant{
      public static  void main(String[]  args){
            int []  tableau   = {1,2,3,4,5,6,7,8,9,10};
            for(int  i  = 0;  i  < tableau.length;  i  =  i  + 1)
                  System.out.println(tableau[i ]);
      }
}
```

Exemple

- Parcours des composants d'un tableau d'entiers suivant l'ordre descendant des indices

```
package be.heb.esi.lg1.tutorials.tableaux;

public class SimpleParcoursDescendant{
      public static  void main(String[]  args){
            int []  tableau   = {1,2,3,4,5,6,7,8,9,10};
            for(int  i  =  tableau.length  –  1;  i  >= 0;  i  =  i  –  1)
                  System.out.println(tableau[ i ]);
      }
}
```

Exemple

- Que va imprimer le code suivant ?

```
package be.heb.esi.lg1.tutorials.tableaux;

public class ParcoursLigneParLigne{
   public static void main(String[] args){
        String [][] tableau= {{"00","01","02","03","04"},
                              {"10","11","12","13","14"}};
        for(int i = 0; i < tableau.length; i = i + 1)
          for(int j = 0; j < tableau.length; j = j + 1)
            System.out.println(tableau[i][j]);
     }
}
```

Exemple

- Parcours des composants d'un tableau d'entiers à deux dimensions ligne par ligne

```
package be.heb.esi.lg1.tutorials.tableaux;

public class ParcoursLigneParLigne{
   public static void main(String[] args){
        String [][] tableau= {{"00","01","02","03","04"},
                              {"10","11","12","13","14"}};
        for(int i = 0; i < tableau.length; i = i + 1)
          for(int j = 0; j < tableau[i].length; j = j + 1)
            System.out.println(tableau[i][j]);
     }
}
```

Exemple

```
int [][] t = new int [3][2];
```

Pourrait s'écrire

```
int [][] t;
t = new int [3][];
for (int i=0; i<3; i=i+1)
   t[i] = new int[2];
```

Exemple

- Création d'un tableau triangulaire

```
package be.heb.esi.lg1.tutorials.tableaux;

public class TableauTriangulaire{
   public static void main(String[] args){
        int [][] t;
        t = new int [3][];
        for (int i=0; i<3; i=i+1)
             t[i] = new int[i+1];
     }
}
```

Exemple : Triangle de Pascal

```
public class TrianglePascal{
    public static void main(String[] args){
        int [][] pascal; // le triangle de Pascal
        int taille ;      // taille du triangle
         taille = Lire.intData();
        pascal = new int[ taille ][];
        for (int i=0; i< taille ; i=i+1)
        {
            pascal[i ] = new int[i+1];
            pascal[i ][0] = 1;
            pascal[i ][ i ] = 1;
            for(int j=1; j<i ; j=j+1)
                pascal[i ][ j ] = pascal[i−1][j−1]+pascal[i−1][j ];
        }
    }
}
```

Exemple : lecture d'un vecteur

```
public class LectureVecteur
{
    public static void main(String[] args)
    {
        int [] vecteur;      // le vecteur à lire
        int    taille ;      // taille du vecteur
         taille  = Lire.intData();
        for (int i=0; i< taille ; i=i+1)
            vecteur[i] = Lire.intData();
    }
}
```

- Attention : ceci n'est pas correct. Pourquoi ?

Exemple : lecture d'un vecteur

```
import be.heb.esi.lg1. util .Lire ;
public class LectureVecteur
{
    public static void main(String [] args)
    {
        int [] vecteur ;       // le vecteur à lire
        int     taille ;       // taille du vecteur
         taille  = Lire .intData ();
        vecteur = new int[ taille ];
        for ( int i=0; i< taille ; i=i+1)
             vecteur[ i ] = Lire .intData ();
    }
}
```

Exemple : décalage circulaire

- Ecrire le programme qui décale les éléments d'un vecteur (vers la droite, circulairement et en place)
- Exemple :

$$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix} devient \begin{pmatrix} 3 & 1 & 2 \end{pmatrix}$$

Exemple : décalage circulaire

```
public class DecalageCirculaire
{
    public static void main(String[] args)
    {
        int [] vecteur;      // le vecteur à manipuler
        int    caseSauvée; // sauvée lors du décalage
        int     taille ;      // taille du vecteur
        // ici : création et initialisation du vecteur
        caseSauvée = vecteur[vecteur.length-1];
        for (int i=vecteur.length-1; i>0; i=i-1)
            vecteur[i] = vecteur[i-1];
        vecteur[0] = caseSauvée;
    }
}
```

Exemple : déplacement de colonnes

- Ecrire le programme qui décale les colonnes d'un tableau (vers la droite, circulairement et en place)
- Exemple :

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} devient \begin{pmatrix} 3 & 1 & 2 \\ 6 & 4 & 5 \\ 9 & 7 & 8 \end{pmatrix}$$

Exemple : déplacement de colonnes

```
public class DecalageColonnes
{
    public static void main(String[] args)
    {
        int [][] matrice;      // la matrice à manipuler
        int      caseSauvée; // sauvée lors du décalage
        // ( ici : code pour initialiser la matrice)
        for (int i=0; i<matrice.length; i=i+1)
        {
            caseSauvée = matrice[i][matrice[i].length-1];
            for (int j=matrice[i].length-1; j>0; j=j-1)
                matrice[i][j] = matrice[i][j-1];
            matrice[i][0] = caseSauvée;
        }
    }
}
```

Exemple : déplacement de lignes

- Ecrire le programme qui décale les lignes d'un tableau (vers le bas, circulairement et en place)
- Exemple :

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} devient \begin{pmatrix} 7 & 8 & 9 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

Exemple : déplacement de lignes

```
public class DecalageLignes
{
    public static void main(String[] args)
    {
        int [][] matrice;        // la matrice à manipuler
        int []   ligneSauvée; // la dernière ligne de la matrice

        // ( ici : code pour initialiser la matrice)
        ligneSauvée = matrice[matrice.length-1];
        for (int i=matrice.length-1; i>0; i=i-1)
            matrice[i] = matrice[i-1];
        matrice[0] = ligneSauvée;
    }
}
```

Les fonctions

- Présentation
- Définition d’une fonction
- Appel d’une fonction
- Passage de paramètre
- La fonction **main**
- Les conversions

Présentation

- Pourquoi découper un code en fonctions ?
  - **Réutilisation**
  - Code identique dans différents endroits du programme
  - Code déjà écrit *ailleurs*
  - **Scinder la difficulté**
  - Faciliter le déverminage
  - **Accroître la lisibilité**
  - Diviser le travail

Présentation

- Comment découper un code en fonctions ?
  - Résout un **sous-problème** bien précis
  - Est **fortement documentée**
  - Est la plus **générale possible**
  - Tient sur une page
  - N’a pas d’effet de bord

Présentation

- Une fonction est une **boîte noire**
  - On lui fournit une/des données
  - Elle retourne une donnée
- Etapes de la vie d'une fonction
  - **Definition** d'une fonction
  - **Appel** (demande d'exécution d'une fonction)
  - Ordre d'apparition dans le code source est sans importance

Etapes de la vie d'une fonction

- Definition d'une fonction
- On décrit la fonction en choisissant :
  - son nom
  - ses paramètres formels
    - nom
    - type
    - ordre
  - son type de valeur de retour
  - son code

Etapes de la vie d'une fonction

- Appel (demande d'exécution d'une fonction)
  - on appelle une fonction en invoquant
    - son nom
    - une liste correspondante de paramètres effectifs
  - on récupère éventuellement la valeur de retour de la fonction

Définition d'une fonction

- Syntaxe (*version simplifiée pour la présentation*)

FunctionDeclaration :
`public static` *ResultType Identifier*
`(` *FormalParameterList* (opt) **`)`** *Block*

- **`public`** et **`static`** : détaillés plus tard
- *Identifier* : nom de la fonction
  - **Mêmes règles** que pour une **variable**
  - **Décrit** au mieux **ce que fait** la fonction
  - On peut réutiliser un nom de variable déjà utilisé (espaces de noms différents)

Définition d'une fonction

- *ResultType* : type de la valeur que va retourner la fonction

---

ResultType :
  Type
  `void`

  - Tous les types sont admis
  - **`void`** : la fonction ne retourne rien
  - Techniquement, **`void`** est un type sans valeur

Définition d'une fonction

- *FormalParameterList* : liste de paramètres formels

FormalParameterList :
FormalParameter
FormalParameterList **,** *FormalParameter*

FormalParameter :
final(opt) *Type Identifi er*

- Rappelons qu'elle est optionelle
- **final** : rend le paramètre constant
- Tous les noms des paramètres formels doivent être différents

Définition d'une fonction

- La grammaire montre bien qu'on doit répéter le type même si il est partagé par les paramètres
  - **int** i , **int** j est OK
  - **int** i , j n'est pas permis
  - Différence par rapport à la déclaration de variable

Définition d'une fonction

- *Block* : Le texte de la fonction
  - Se présente sous forme d'un bloc d'instructions (même si une seule instruction)
  - Les paramètres formels peuvent y être utilisés comme n'importe quelle autre variable déclarée dans le bloc

Définition d'une fonction

- L'instruction **return** signale la fin de l'exécution du code

---

returnStatement :
`return` *Expression*(opt) **`;`**

  - On revient au code appelant
  - On retourne la valeur de l'expression suivant le **`return`**
  - Lorsque le *ResultType* vaut **`void`**, **`return`** n'est suivi d'aucune expression

Définition d'une fonction

- Exemples

```
public static  int  add ( int  opérandeGauche,
                          int  opérandeDroite )
{
      int  somme = opérandeGauche + opérandeDroite;
      return somme;
}
```

```
public static  int  add( int  opérandeGauche,
                         int  opérandeDroite )
{
      return opérandeGauche + opérandeDroite;
}
```

Définition d'une fonction

```
public static void hello ( String nom )
{
    System.out.println("Bonjour␣" + nom + "!");
    return ;
}
```

```
public static int addLire ()
{
    return Lire.intData () + Lire.intData ();
}
```

```
public static void entête () {
    System.out.println("HEB−ESI␣:␣Laboratoires␣Java");
    return ;
}
```

Définition d'une fonction

- L'instruction **return**
  - Peut apparaître n'importe où dans le texte de la fonction
  - Normalement, au moins une fois
    - On conseille **exactement une fois**
  - Si pas présent, **return**; implicite en fin de block
    - ($\Rightarrow$ valable uniquement si retour de type **void**)

Définition d'une fonction

- Exemple : calcul du maximum de 2 nombres

```
public static int max ( int nb1, int nb2 )
{
      if (nb1 > nb2)
            return nb1;
      else
            return nb2;
}
```

Définition d'une fonction

- Autre solution pour le même problème

```
public static int max ( int nb1, int nb2 )
{
      if (nb1 > nb2)
            return nb1;
      return nb2;
}
```

Définition d'une fonction

- On préférera

```
public static int max ( int nb1, int nb2 )
{
      int max=0; // le maximum des nombres nb1 et nb2
      if (nb1 > nb2)
            max = nb1;
      else
            max = nb2;
      return max;
}
```

Définition d'une fonction

- Le compilateur peut détecter que le **return** ne serait pas rencontré dans certains cas
- Exemple

```
public static int max ( int nb1, int nb2 )
{
      int max; // le maximum des nombres nb1 et nb2
      if (nb1 > nb2)
            return nb1;
      else
            max = nb2; // détecte une erreur à la compilation
}
```

Appel d'une fonction

- Pour demander l'exécution d'une fonction (l'invoquer)
  - On donne son nom
  - Suivi d'une liste de paramètres **effectifs** correspondant aux paramètres formels
    - en ordre
    - en type
- C'est une expression qui vaut la valeur retournée par l'exécution de la fonction

Appel d'une fonction

- Exemples
  - max(1,2) : vaut 2
  - **int** m = max(1,2) : m reçoit 2
  - **int** max = max(1,2) : espaces de nom différents
  - **int** a=1; **int** max = max(a,2) : max vaut 2
  - **int** a=1; a = max(a+1,1) : a vaut 2
  - **int** a=1; a = max(a,2)+1 : a vaut 3
  - max(max(1,2),3) : vaut 3
  - max(max(Lire.**int**(),Lire.**int**()), Lire.**int**())

Un exemple complet

```
package be.heb.esi.lg1.cours;
import be.heb.esi.lg1. util . Lire ;
// Calcul du maximum d'un ensemble d'entiers positifs
// entrés au clavier . On termine par la sentinelle −1
public class MaxEntiers
{
    // max est le maximum des entiers nb1 et nb2
    public static int max ( int nb1, int nb2 )
    {
        int max=0;
        if (nb1 > nb2)
            max = nb1;
        else
            max = nb2;
        return max;
    }
```

Un exemple complet

```
    public static void main ( String [] args )
    {
        int max;  //  Le max (courant) des nombres lus
        int nbLu; //  Chacun des nombres lus

        nbLu = Lire.intData ();
        max = nbLu;
        while( nbLu != -1 )
        {
            max = max(max,nbLu);
            nbLu = Lire.intData ();
        }
        System.out.println("max = "+max);
    }
}
```

Un exemple complet

- Dans l'exemple précédent
  - que manque-t-il aux commentaires ?
  - comment pourrait-on améliorer le programme ?

Passage de paramètres

- En logique, il y a trois sortes de paramètres
  - En entrée
  - En sortie
  - En entrée-sortie
- Qu'en est-il en **`Java`** ?
  - Tous les **passages de paramètres** se font **par valeur**

Passage de paramètres

- Chaque paramètre effectif est **copié** dans le paramètre formel correspondant (comme une assignation)
- Dès lors,
  - Pour un type primitif, la **valeur est copiée** ($\approx$ paramètre en entrée)
  - Pour un type référence, la **référence est copiée** ($\approx$ paramètre en entrée-sortie)

Passage de paramètres : exemple

```
public class PassageValeur
{
    public static void fois2 ( int nb )
    {
        nb = nb * 2;
    }

    public static void main ( String [] args )
    {
        int entier = 1;
        fois2 ( entier );
        System.out.println ("entier = "+entier); // 1
    }
}
```

Passage de paramètres : exemple

```
public class PassageValeur2
{
    public static void fois2 ( int [] nbs )
    {
        nbs[0] = nbs[0] * 2;
    }

    public static void main ( String [] args )
    {
        int [] entiers = {1,3,5};
        fois2(entiers);
        System.out.println("entier = "+entiers[0]); // 2
    }
}
```

Un exemple complet

- Dans l'exemple précédent
  - quel commentaire donner à cette fonction ?
  - quel bug y a-t-il ?

Passage de paramètres : exemple

- Que penser de ceci ?

```
public static void f( int [] tab )
{
    int taille = tab.length;
    int [] copie = new int[ taille ];
    for(int i=0; i< taille ; i=i+1)
        copie[i ] = tab[ taille −1−i];
    tab = copie;
}
```

Passage de paramètres

- En logique, on ne peut pas retourner de tableau
- En **Java**, **tout type valide peut être le type de retour**
  - $\Longrightarrow$ même les types références
  - $\Longrightarrow$ même les tableaux

Passage de paramètres

```
public class TableauEnRetour
{
    public static int [] créerTableau( int taille , int valeur )
    {
        int [] entiers = new int[ taille ];
        for(int i=0; i< taille ; i=i+1)
            entiers[ i ] = valeur;
        return entiers
    }

    public static void main ( String [] args )
    {
        int [] tableau = créerTableau(10,13);
    }
}
```

La fonction main

- La fonction ***principale*** de la classe
- Exécuter une classe ≡ exécuter sa fonction main
- L'entête est imposé
  - **public static void** main(String[] args)
  - note : args est libre (argument formel)
- Ex : **public static int void** main() {}
  - Est acceptée à la compilation
  - N'est pas reconnue comme la fonction principale à l'exécution

La fonction main

- Lorsque une classe **`Java`** est exécutée, elle reçoit des **paramètres du système**
- Toujours sous la forme d'un tableau de String
- On dit aussi **argument** plutôt que paramètre
- Si aucun argument, le tableau est vide

La fonction main : exemple

```
// Inverse les arguments passés par le système
public class InverserArguments
{
    public static void main ( String [] args )
    {
        for ( int i=args.length−1; i>=0; i=i−1 )
            System.out.print(args[i]+"␣");
        System.out.println("");
    }
}
```

```
$ java InverserArguments Phrase à lire à l ' envers
envers l ' à lire à Phrase
```

La fonction main

- Rappel : tous les arguments sont passés comme des chaînes même si ils pourraient être vus comme des entiers
  - Exemple

```
$ java  InverserArguments 1 + 2 = 3
3 = 2 + 1
```

- L'espace sert à délimiter les arguments
  - Exemple

```
$ java  InverserArguments 1 + 2=3
2=3 + 1
```

La notion de signature

- **Signature d'une fonction** : son nom + sa liste de paramètres formels
- Le type de retour n'intervient pas dans la signature
- Règle : **On ne peut pas déclarer dans une même classe deux fonctions ayant la même signature**
- Pourquoi ? Le compilateur ne pourrait pas facilement savoir laquelle choisir lors d'un appel

Exemples

```
public static  double
      add( double opérandeGauche, double opérandeDroite) {
            return opérandeGauche + opérandeDroite;
}

public static  int  // OK
      add( int  opérandeGauche, int opérandeDroite) {
            return opérandeGauche + opérandeDroite;
}

public static  long // Pas OK
      add( int  opérandeGauche, int opérandeDroite) {
            return opérandeGauche + opérandeDroite;
}
```

Exemples

```
public static  int  //  OK
      add( short opérandeGauche, byte opérandeDroite) {
            return opérandeGauche + opérandeDroite;
}

public static  int  //  OK
      add( int  opérandeGauche, long opérandeDroite) {
            return opérandeGauche + opérandeDroite;
}

public static  long //  OK
      add( long opérandeGauche, int opérandeDroite) {
            return opérandeGauche + opérandeDroite;
}
```

Conversions

- Rappelons les 5 contextes de conversions
  - La promotion numérique
  - L'assignation
  - **L'appel de méthode**
  - La chaîne de caractères
  - Le *casting*
- Il est temps de détailler celle intervenant dans les appels de méthodes

Conversions

- Nous avons dit que les paramètres effectifs devaient correspondre en type aux paramètres formels
- En fait, certaines conversions peuvent être appliquées implicitement
  - La conversion **élargissante**
  - La conversion **identique**

Conversions

- Exemple, avec la fonction suivante

```
public static  int  add( int  opérandeGauche,
                         int  opérandeDroite)
{
        return opérandeGauche + opérandeDroite;
}
```

- Les arguments lors de l'appel peuvent être **byte**, **short**, **char** ou encore **int**

Conversions

- Si il existe plusieurs fonctions avec le même nom, il peut y avoir plusieurs candidats pour un appel
- Exemple

```
public static int f( long op1, long op2 ) {...}
public static int f( int op1, int op2 ) {...}
```

- Quelle fonction est choisie avec cet appel ?

```
short s1=1, s2=2:
f(s1,s2);
```

Conversions

- La règle indique que le compilateur choisit la méthode la plus **spécifique**
- Une méthode est plus spécifique si les arguments sont plus ***petits*** (en terme de conversion élargissante)
- Dans notre exemple, il choisira la fonction

```
public static int f( int op1, int op2 ){...}
```

Exemple

```
public static  double
      add( double opérandeGauche, double opérandeDroite) {
            return opérandeGauche + opérandeDroite;
}

public static  int
      add( int  opérandeGauche, int opérandeDroite) {
            return opérandeGauche + opérandeDroite;
}
```

- add(3.,2.) retourne 5.0
- add(3,2) retourne 5
- add(3.,2) retourne 5.0

Conversions

- Parfois, il n'y a pas de méthode plus spécifique
- Exemple

```
public static int f( int op1, long op2 ) {...}
public static int f( long op1, int op2 ) {...}
```

- Ceci est accepté mais certains appels seront ambigus (erreur à la compilation)
  - f(1,2L) // 1ère méthode
  - f(1L,2) // 2ème méthode
  - f(1,2) // ambigu

Conversion

- Et en ce qui concerne le **type de retour** ?
- Le type de l'expression accompagnant l'instruction **return** doit pouvoir être ramené au *ResultType*
- Par les **mêmes conversions que** lors d'une **assignation**
  - conversion identique
  - conversion élargissante
  - conversion arrondissante (sous les mêmes conditions $\Rightarrow$ se rencontrera peu)

Conversion

- Exemple

```
public static long
      add( int  opérandeGauche, int opérandeDroite) {
            return opérandeGauche + opérandeDroite;
}
```

- Accepté car la somme est un **int** qui peut être élargi en **long**

Exemples récapitulatifs

```
public class Fonction{

    public static void main(String[] args){
        int paramètreEffectif1 = 2;
        int paramètreEffectif2 = 5;
        int res = modif(paramètreEffectif1,paramètreEffectif2);
        System.out.println(res);                    // imprime: 4
        System.out.println(paramètreEffectif1 );    // imprime: ?

        int [] tableau = {5,6,8,2,4};
        printTab(tableau);     // imprime: le tab vaut : 5 6 8 2 4
        reset(tableau);
        printTab(tableau);     // imprime: le tab vaut : 0 0 0 0 0
    }
```

Exemples récapitulatifs

```
public static int modif(int opérandeGauche,
                        int opérandeDroit)
{
    opérandeGauche = opérandeGauche – 3;
    return opérandeGauche + opérandeDroit;
}

public static void reset(int [] tab){
    for(int i = 0; i < tab.length; i = i + 1)
        tab[i] = 0;
    return;
}
```

Exemples récapitulatifs

```
    public static void printTab( int [] tab )
    {
        System.out.print("le␣tab␣vaut:␣");
        for(int i = 0; i < tab.length; i = i + 1)
            System.out.print(tab[i] + "␣");
        System.out.print("\n");
        return;
    }
}
```

Conclusion

- Insistons : Une **fonction**
  - **fait une et une seule chose**
  - possède un **nom explicite**
  - est **fortement documentée**
- Exercice : commenter le programme précédent

Introduction aux objets

- Introduction
- Classe et instance
- Propriétés et fonctionnalités
- Constructeur

Introduction

- Nous avons évoqué **différents moyens de stocker des valeurs** de différents types et de les manipuler
  - types : int, double, char, boolean, String, tableaux, etc...
  - stockage : variables
  - manipuler : assignation, boucle, choix, fonction

Introduction

- Les types vus jusqu'à présent couvrent des **informations peu complexes** se décrivant par une seule valeur
  - age du capitaine
  - compteur
  - ...

Introduction

- **La plupart des informations** qu'un programmeur devra manipuler sont cependant **beaucoup plus sophistiquées**
- D'autant plus qu'il s'agira de **notions de la vie quotidienne**
  - date
  - client
  - livre
  - imprimante
  - ...

Introduction

- **Langage procédural** : notion de **structure** répond partiellement à la complexité des informations
- **L'approche OO** : **place les données au centre** du design du langage
- Description via 2 points de vues différents
  - bottom/up : développement historique des langages OO
  - top/down : langage OO vu comme langage d'implémentation des spécifications OO issues de l'analyse

Enseignements de l'analyse

- Votre cours d'analyse met en évidence l'efficacité d'une méthode de description d'un système complexe comme une bibiothèque par l'identification d'entités (objets)
  - clients
  - livres
  - bibliothécaires
  - prets
  - . . .

Classe et instance

- Ces objets, bien que nombreux, se rassemblent en un nombre limités de types (**classes**) dont ils constituent des occurences (**instances**)
- Au même titre que
  - 2 est une instance du type primitif int
  - 'f' une instance du type primitif char
- On dira donc que la **classe Client est le type des clients** ou encore que **les clients sont des instances de la classe Client**

Les propriétés

- Tous ces clients partagent les mêmes propriétés (**attributs**), ils ont tous
  - un nom
  - une adresse
  - une liste de livres empruntés
  - ...
- Mais tous ces clients se distinguent par **des valeurs différentes** de ces propriétés :
  - le nom d'un client est "Al Katabi" l'autre est "Van Piperzeel"

Les fonctionnalités

- Tous ces clients partagent les mêmes fonctionnalités (**méthodes**), ils peuvent tous :
  - emprunter un livre
  - donner leur nom
  - restituer un livre
  - . . .

La classe

- Rassemblons tout ce que les **clients ont en commun** (leur type) en une **classe** Client
- Exemple de définition de la classe Client

```
class Client {
    String nom;
    String adresse;
}
```

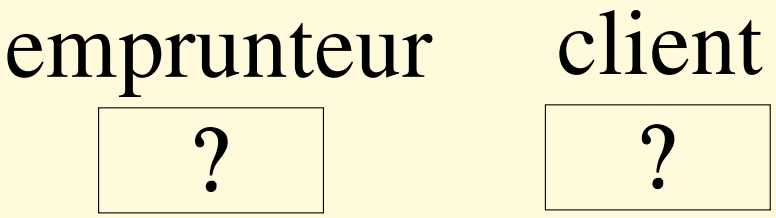
La classe

- En **Java**, toutes les **classes** seront des types **références**
- $\Rightarrow$ Une déclaration de variable de type Client
  - **n'entrainera pas la création d'un objet** client
  - mais la **réservation de l'espace mémoire nécessaire pour stocker une référence** vers un objet de ce type

La classe

- Exemples de déclarations

```
Client  emprunteur;
Client   client ;
```



La classe

- Ce que nous avons fait pour les clients, nous pourrions le faire pour les livres ainsi que pour les prêts

```
class Pret{
        String  dateEmprunt;
        String  dateRetour;
        boolean estPayé;
        Livre  livreEmprunté;
        Client  emprunteur;
}
```

```
class Livre{
        String   titre ;
        String  état ;
}
```

Les méthodes

- Les clients ne partagent pas que des **propriétés**, ils partagent aussi des **fonctionnalités** telles que :
  - faire un pret de livre : emprunter un livre
  - terminer un pret de livre : ramener un livre
- La définition générale d'un client (sa classe) comprendra
  - les propriétés des clients (on dit **attributs**)
  - les fonctionnalités propres aux clients (on dit **méthodes**)

La classe

- On complètera donc la définition de la classe Client de la façon suivante

```
class Client {
        String  nom;
        String  adresse;
        boolean emprunte(String date, Livre livre ) {
                //   ...
        }
        boolean ramene(Livre livre) {
                //   ...
        }
}
```

La classe

- Les effets de ces fonctionnalités sont dépendantes de chaque client
- Différence avec les fonctions **static** (relèvent seulement de la classe où elles sont définies)
- Exemple

```
class Client {
        String nom;
        String adresse;
        static int nombreAttributsClient() {return 2;}
        // ...
}
```

Instanciation

- Nous avons **défini** des classes
- A présent, comment créer une **instance**, une **occurence** de la classe Client ?
  - ou encore créer un **objet** Client pour adopter la terminologie OO
- Ce que tous les clients ont en commun : leurs attributs et fonctionnalités
- Créer un objet : retenir ce qui le distingue des autres clients, les valeurs de ses attributs

Instanciation

- Il y a 2 aspects
  - **allocation de l'espace mémoire** nécessaire pour stocker les valeurs particulières des attributs de l'objet
  - **initialisation** de ces différents **attributs**
- Cela se fait par
  - L'utilisation du mot clef **new** (allocation mémoire)
  - Suivi de l'invocation d'un **constructeur** de la classe (**initialisation** des différents attributs)

Constructeur

- **Constructeur**
  - Méthode particulière que l'on retrouve dans la définition d'une classe
  - Son nom est le même que celui de sa classe
  - On peut en définir autant qu'on le désire (signatures différentes)
  - Initialise l'objet
  - Pas de valeur de retour

Constructeur

- Exemple : Ajoutons deux constructeurs à la classe Client
  - Un constructeur sans paramètre (n'initialise rien)
  - Un constructeur avec le nom et l'adresse du client en paramètres

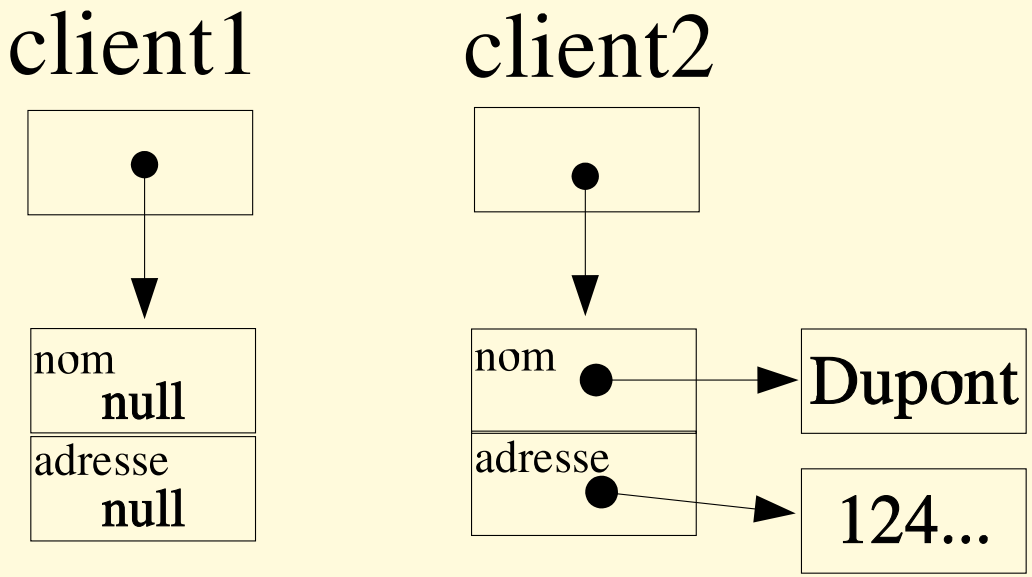
Constructeur

```
class Client {
        String nom;
        String adresse;
        Client () {}
        Client (String nomClient, String adresseClient) {
                nom = nomClient;
                adresse = adresseClient;
        }
        boolean emprunte(String date, Livre livre ) {
                //  ...
        }
        boolean ramene(Livre livre) {
                //  ...
        }
}
```

Instanciation

- Nous disposons à présent de tout ce qu'il faut pour créer un objet Client et stocker sa référence

```
Client  client1  = new Client();
Client  client2  = new Client( "Dupont",
        "124, av des Atrébates 1780 Wemmel Belgique");
```



Accès aux attributs

- Comment **accéder aux attributs** de client1 ?
- En faisant suivre l'objet de l'attribut de la manière suivante

```
String  nomClient;
nomClient = client1.nom;
```

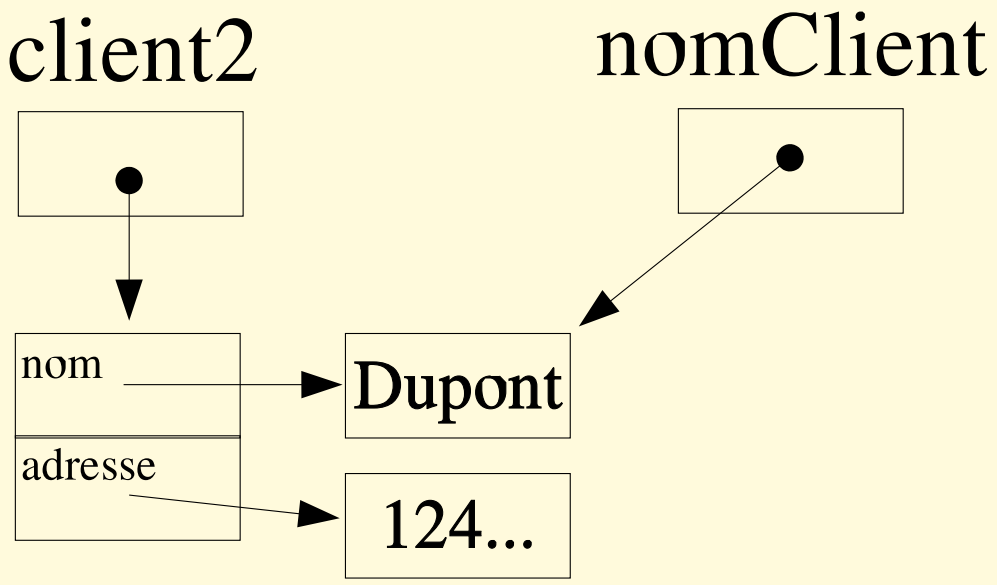
- De la même manière on peut assigner "Tintin" comme nom de client1

```
client1 .nom = "Tintin";
```

Accès aux attributs

- A la différence de client1, client2 a vu ses attributs initialisés lors de sa création

```
String  nomClient;
nomClient = client2.nom;
```



Exemple : suite

- Ce que nous avons fait pour Client peut l'être également pour Livre et Pret

```
class Livre{
        String   titre ;
        String  état;
        Livre () {
        }
        Livre(String   titreLivre  ) {
                titre  =  titreLivre ;
               etat  =  "neuf";
        }
}
```

Exemple : suite

```
class Pret{
      String  dateEmprunt;
      String  dateRetour;
      boolean estPayé;
      Livre  livreEmprunté;
      Client  emprunteur;
      Pret () { }
      Pret(String  dateEmpruntPret, Livre livreEmpruntePret,
                  Client  emprunteurPret) {
            dateEmprunt = dateEmpruntPret;
            estPayé = false;
            livreEmprunté = livreEmpruntePret;
            emprunteur = emprunteurPret;
      }
}
```

Exemple : suite

- Imaginons à présent le scénario suivant

```
Livre  livre  = new Livre( "Les hauts de Hurlevent" );

Client  client  = new Client( "Dupont",
        "124, av des Atrébates 1780 Wemmel Belgique");

Pret  pret  = new Pret("28/11/2003", livre ,  client );

String  nomEmprunteur = pret.emprunteur.nom;
     //  vaut "Dupont"
```

Exemple : suite

- Reprenons la définition de Client
  - Stockons les prêts d'un client parmis ses attributs
  - Complétons la méthode emprunte()

Exemple : suite

```
class Client
{
      String  nom;
      String  adresse;
      Pret []  prets;
      int  nombrePrets;

      Client () {
            prets  =  new Pret[20];
            nombrePrets = 0;
      }

      Client(String  nomClient, String  adresseClient) {
            nom = nomClient;
            adresse = adresseClient;
            prets  =  new Pret[20];
            nombrePrets = 0;
      }
```

Exemple : suite

```
    boolean emprunte(String date, Livre livre ) {
        boolean empruntAccepté = false;
        if (nombrePrets < prets.length)
        {
            prets[nombrePrets] = new Pret(date, livre , this );
            nombrePrets = nombrePrets + 1;
            empruntAccepté = true;
        }
        return empruntAccepté;
    }

    boolean ramene(Livre livre ) {...}
}
```

Exemple : suite

- Imaginons à présent le scénario suivant

```
Livre   livre  = new Livre( "Les␣hauts␣de␣Hurlevent" );
Client   client  = new Client( "Dupont",
        "124,␣av␣des␣Atrébates␣1780␣Wemmel␣Belgique");
Pret  pret  = new Pret("28/11/2003", livre ,  client );
 client .emprunte("18/11/2003",livre);
```

- Que valent ?
  - client .nombrePrets
  - client .prets[0]
  - client .prets[1]
  - client .prets[0]. livre .etat

Conclusion

- Les notions de classe, d'objet, d'attribut, de méthode et de constructeur ayant été présentées intuitivement, nous les redéfinirons de façon formelle au prochain cours

Classe et objet

- Déclaration d’une classe
- Attribut
- Méthode
- Accès
- Constructeur
- Exemple récapitulatif

Déclaration d'une classe

- Revient à définir un **nouveau type référence**
- Toujours visible dans son propre package
- Sauf spécification contraire, pas visible de l'extérieur de son package

Déclaration d'une classe

- Syntaxe

ClassDeclaration :
ClassModifiers(opt) **`class`** *Identifier* *Super*(opt) *Interfaces*(opt)
ClassBody

ClassModifiers :
ClassModifier
ClassModifiers ClassModifier

ClassModifier : one of
`public abstract final`

Déclaration d'une classe

- **`class`** : mot clef
- *Identifier*
  - syntaxe : mêmes règles que pour les variables locales
  - conventions : pas de majuscule sauf premier caractère de mots collés
  - pas de classes synonymes dans un même package

Déclaration d'une classe

- *Identifier*
  - pas d'import d'une classe ayant le même nom

```
package Corentin;
import java. util .Vector ; // erreur de compilation
public class Vector {...}
```

```
package Corentin;
import java. util .*; // pas d'erreur de compilation
public class Vector {...}
```

Déclaration d'une classe

- *ClassModifiers*
  - **`public`** : rend une classe visible de l'extérieur de son package
  - **`abstract`** :
    - classe ne pouvant faire l'objet d'une instanciation
    - et/ou classe partiellement définie
  - **`final`** : classe ne pouvant être héritée (voir plus loin)

Déclaration d'une classe

- *ClassModifiers*
  - liste sans virgule
  - le compilateur refuse les répétitions

Déclaration d'une classe

- *Super*
  - optionnel
  - détermine la classe héritée
  - voir plus loin dans le cours
- *Interface*
  - optionnel
  - détermine les interfaces implémentées
  - voir plus loin dans le cours

Déclaration d'une classe

ClassBody
{ *ClassBodyDeclarations*(opt) }

ClassBodyDeclarations :
ClassBodyDeclaration
ClassBodyDeclarations ClassBodyDeclaration

```
class MyClass {} //ok

class MyClass {
    ClassBodyDeclaration_1
     ...
    ClassBodyDeclaration_n
}
```

Déclaration d'une classe

- Le corps d'une classe contient 4 sortes d'éléments

ClassBodyDeclaration :
- *ClassMemberDeclaration*
- *StaticInitializer*
- *ConstructorDeclaration*

ClassMemberDeclaration :
- *FieldDeclaration*
- *MethodDeclaration*

Les attributs

- Les attributs

FieldDeclaration :
FieldModifi ers(opt) *Type VariableDeclarators* **;**

FieldModifi ers :
FieldModifi er
FieldModifi ers FieldModifi er

FieldModifi er : one of
public protected private final
static transient volatile

Les attributs

- La visibilité d'une variable (attribut, propriété)
  - rien (package) : ne pourra être accédée qu'à partir du code de son propre package
  - **`public`** : pourra être accédée à partir du code pour lequel sa classe est visible
  - **`protected`** : ne pourra être accédée qu'à partir du code de sa propre classe ou d'une classe héritante (voir plus loin)
  - **`private`** : ne pourra être accédée qu'à partir du code de sa propre classe

Les attributs

- **`static`**
  - la variable est associée à la classe
  - n’est pas dupliquée pour chaque objet
  - on parle alors de variable de classe (opposé aux variables d’instances)

Les attributs

- **`static`**
  - initialisée lors du chargement de la classe
  - les variables d'instance ne sont pas initialisée lors du chargement de la classe mais, plus tard lors de la création de chaque objet
  - on appelle souvent cette initialisation *valeur par défaut* de l'attribut

Les attributs

- Autres modifieurs
  - **`final`**
    - la valeur de la variable ne pourra être modifiée une fois initialisée
    - nom en majuscule
  - **`transient`** : voir plus loin
  - **`volatile`** : voir plus loin

Les attributs

- Exemple

```
class Groupe
{
        static  int  nombreGroupes = 0;
        int  nombreEleves = 0;
        String  professeur = null;
}
```

Les attributs

- La création d'un objet se traduit par la réservation de l'espace mémoire nécessaire pour stocker les variables
- La variable entière nombreGroupes n'existe qu'en un seul exemplaire créé et initialisé lors du chargement de la classe

Les attributs

- Contrairement aux variables locales, les attributs ne doivent pas être initialisées explicitement avant leur usage

numérique	0
booléen	false
référence	null

Les attributs

- Pour la valeur initiale, les règles sont les mêmes que pour les variables locales

```
class Test {
      fbat  f = j ;    //  erreur
      int  j  = 1;
      int  k = k+1;    //  erreur
}
```

```
class Test {
      fbat  f = j ;    //  pas d'erreur
      static  int  j  = 1;
}
```

Les méthodes

- Reprenons

ClassBodyDeclaration :
ClassMemberDeclaration
StaticInitializer
ConstructorDeclaration

ClassMemberDeclaration :
FieldDeclaration
MethodDeclaration

- syntaxiquement, les méthodes sont identiques aux fonctions (sans le modificateur static)

Les méthodes

MethodDeclaration :
MethodHeader MethodBody

MethodHeader :
MethodModifi ers(opt) *ResultType Identifer*
(*FormalParameterList*(opt) **)** *Throws*(opt)

MethodModifi ers :
MethodModifi er
MethodModifi ers MethodModifi er

MethodModifi er : one of
`public protected private abstract`
`static final synchronized native`

Les méthodes

- La visibilité d'une méthode : idem attributs
  - rien (package) : ne pourra être accédée qu'à partir du code de son propre package
  - **`public`** : pourra être accédée à partir du code pour lequel sa classe est visible
  - **`protected`** : ne pourra être accédée qu'à partir du code de sa propre classe ou d'une classe héritante (voir plus loin)
  - **`private`** : ne pourra être accédée qu'à partir du code de sa propre classe

Les méthodes

- **`static`**
  - la méthode est associée à la classe
  - on parle alors de méthode de classe en opposition aux méthodes d'instance
- **`final`** : voir plus loin
- **`abstract`** :
  - méthode sans partie *MethodBody*
  - sa classe doit être déclarée abstract également
  - la méthode ne peut être ni **`private`** ni **`static`** ni **`final`**

Accès

- Accès à un membre (attribut ou méthode) **de classe**
  - préfixer le nom du membre par le nom de la classe où il a été défini
  - ex : `Lire.intData()`
- Accès à un membre (attribut ou méthode) **d'instance**
  - préfixer le nom de la variable par un objet de la classe où il a été défini
  - ex : `tab.length`

Accès

- Lorsqu'une variable locale ou un paramètre formel d'une méthode ou d'un constructeur est synonyme d'un attribut de la classe, on distingue l'attribut en le préfixant de **this**

```
class Client {
        String  nom;
        String  adresse;
        Client(String  nom, String adresse) {
                this.nom = nomClient;
                this.adresse = adresseClient;
        }
}
```

Constructeur

- Reprenons

ClassBodyDeclaration :
  ClassMemberDeclaration
  StaticInitializer
  ConstructorDeclaration

ConstructorDeclaration :
  ConstructorModifi er (opt) *ConstructorDeclarator*
    Throws (opt) *ConstructorBody*

ConstructorModifi er : one of
  `public protected private`

Constructeur

- *ConstructorModifier* : mêmes conséquences que pour les méthodes
- *Throws* : voir plus loin dans le cours
- Nom du constructeur

ConstructorDeclarator :
SimpleTypeName **(** *FormalParameterList* (opt) **)**

Constructeur

- Corps du constructeur

ConstructorBody :
{ *ExplicitConstructorInvocation*(opt) *Statements*(opt) **}**

ExplicitConstructorInvocation :
this (*ArgumentList*(opt) **) ;**
super (*ArgumentList*(opt) **) ;**

- **this** : pour invoquer un autre constructeur de la même classe
- **super** : pour invoquer un constructeur de la classe héritée (voir plus loin)

Constructeur

- si pas de *ExplicitConstructorInvocation*
  - $\Rightarrow$ appel implicite : **super**()
  - Erreur si pas de constructeur vide (car redéfini)

Constructeur

- Déroulement d'une instanciation de classe

1. l'espace mémoire nécessaire pour stocker tous les attributs de l'objet est alloué
2. les attributs sont (éventuellement) initialisés
3. le corps du constructeur est exécuté
   (a) invocation du constructeur explicite ou implicite
   (b) exécution de la séquence d'instructions
4. le résultat du constructeur est la référence vers cet espace mémoire

Exemple récapitulatif

```
package be.heb.esi.lg1.tutorials.Classe;

class Point {
    static Point POINTORIGINE = new Point(0,0);
    int x = 0;
    int y = 0;

    public Point(int x, int y){
        this.x = x;
        this.y = y;
    }

    public Point(int x){
        this.x = x;
    }
```

Exemple récapitulatif

```
    public boolean isOnVerticalOf(Point point){
        return this.x == point.x;
    }

    public String  toString (){
        return "x= " + this.x + " y= " + this.y;
    }
}
```

Exemple récapitulatif

```
class Test {
   public static  void main(String[] args){
         Point  pointOrigine = Point.POINTORIGINE;
         Point  point = new Point(1,2);
         System.out.println(point.x);
         System.out.println(point.toString ());
         System.out.println(point);
         point = new Point(3);
         System.out.println(point);
   }
}
```

L'héritage

- Héritage simple
- La classe Object
- Polymorphisme
- Classe abstraite
- Héritage multiple
- Interfaces
- Classes internes

Héritage simple

- L'héritage permet
  - La réutilisation de code
  - Un ajustement des propriétés

ClassDeclaration :
ClassModifiers(opt) **`class`** *Identifier* *Super*(opt)
Interfaces(opt) *ClassBody*

Super :
`extends` *ClassType*

- Permet à une classe de **partager** les définitions de membres de la classe dont elle hérite

Héritage simple

- Exemple

```
class Point2D {
        int  x= 0;
        int  y= 0;

        String  toString () {
                return "x=" + "␣et␣y=" + y;
        }
}
```

```
class Point3D extends Point2D {
        int  z= 0;
}
```

Héritage simple

```
System.out.println(new Point2D());     //  x=0 et y=0
System.out.println(new Point3D());     //  x=0 et y=0
System.out.println(new Point3D().z);   //  0
```

- La méthode `toString()` héritée de la classe Point2D ne convient pas !
- Solution : la réécrire
- Cachera la précédente définition : **Overriding**

Héritage simple

```
class Point3D extends Point2D{
      int  z= 0;
      String  toString () {
            return "x=" + "␣et␣y=" + y + "␣et␣z=" + z;
      }
}
```

```
System.out.println(new Point2D()); //  x=0 et  y=0
System.out.println(new Point3D()); //  x=0 et  y=0 et  z=0
```

Héritage simple

- La méthode originale est toujours accessible grâce à **super**

```
class Point3D extends Point2D{
      int  z= 0;
      String  toString () {
            return super.toString() +  "␣et␣z=" + z;
      }
}
```

```
System.out.println(new Point2D()); //  x=0 et  y=0
System.out.println(new Point3D()); //  x=0 et  y=0 et  z=0
```

Polymorphisme

- On dit que Point3D **spécialise** la classe Point2D
- A ce titre : tout objet de la classe Point3D est aussi un objet de la classe Point2D
- On pourra donc écrire

```
Point2D point2D= new Point3D();
```

- Ce seront cependant les méthodes définies dans la classe Point3D qui seront utilisées :
- `point2D.toString ();  // x=0 et y=0 et z=0`
- Cette propriété s'appelle le **polymorphisme**

Polymorphisme

- Remarque

```
class Point3D extends Point2D{
        int  z= 0;
        String  toString () {
                return super.toString() + "␣et␣z=" + z;
        }
        void foo () {}
}
```

- point2D.foo(); : provoque une erreur à la compilation
- ((Point3D)point2D).foo(); : est accepté

La classe Object

- La syntaxe impose 0 ou une seule classe héritée
- Aucune classe héritée signifie implicitement que la classe hérite de la classe Object
- Dès lors toutes les classes héritent directement ou indirectement de la classe Object
- API de la classe objet : cf. documentation

Classe abstraite

- Bénéficier de classes déjà réalisées est un avantage
- Même quand ces classes ne peuvent pas être complètement définies
- Par exemple :
  - la nécessité d'une méthode est identifiée mais seule sa signature peut être précisée
  - son implémentation dépendra de chaque cas particulier d'utilisation

Classe abstraite

- **Java** offre la possibilité de définir partiellement des classes
  - Certaines méthodes, déclarées abstraites (abstract), ne sont pas suivies de leur définition
- Le compilateur imposera que la classe soit elle aussi déclarée abstract
- Peuvent être héritées par d'autres classes
  - Leurs méthodes abstraites *peuvent* recevoir une implémentation (mécanisme d'*overriding*)

Classe abstraite

- Exemple

```
abstract class Pile {
      int  nombreItem= 0;
      int  size;

      Pile(int  size) {
            this.size= size;
      }
      abstract void push(Object item);
      abstract Object pop();
      boolean isEmpty() {
            return nombreItem == 0;
      }
}
```

Classe abstraite

```
class MyPile extends Pile {
    Object [] container;

    MyPile(int size ) {
        super(size);
        container= new Object[size];
    }
    void push(Object item) {
        if (nombreItem < container.length) {
            container[nombreItem]= item;
            nombreItem = nombreItem + 1;
        }
    }
    Object pop() {
        if (nombreItem == 0) return null;
        nombreItem = nombreItem – 1;
        return container[nombreItem];
    }
}
```

Classe abstraite

Remarque :

- Assez naturellement, une classe abstraite ne pourra pas faire l'objet d'une instanciation
- L'existence d'un objet ne disposant pas de la définition de toutes ses méthodes est considérée comme non significative

Héritage multiple

- Héritage disponible dans beaucoup de langages (**C#**, **Java**)
- Pas en **Java**
- En théorie très séduisant mais pose des difficultés
  - difficulté de gérer les conflits de définitions
  - difficulté de garder un code lisible
- En pratique, pas très utile pour les problèmes courants
- **Java** propose un autre mécanisme : l'**interface**

Interface

- **Interface** ≡ classe où toutes les méthodes sont abstraites
- Plus un ensemble de contraintes qu'un bénéfice en terme d'implémentation
- Utilisé lorsque deux blocs de code développés indépendamment et devant le rester sont amenés à échanger de l'information
- L'échange se fait au travers d'un ensemble de fonctions prédéfinies (l'interface) que l'un des blocs s'engage à invoquer et l'autre à implémenter

Interface

- Exemple

```
abstract class Enumeration {
        abstract boolean hasMoreElement();
        abstract Object nextElement();
}
```

- Une classe est autorisée à hériter(on dira **implémenter**) un nombre indéterminé d'interfaces
- Pas de risque de conflit de définition

Interface

- Exemple

```
interface TextListener {
   void textValueChanged(TextEvent e);
}
```

```
class StrangeObject extends Object
                              implements Enumeration, TextListener
{
      boolean hasMoreElement(){...}
      Object nextElement (){...}
      void textValueChanged(){...}
}
```

Classe interne

- On peut définir des classes à l'intérieur d'autres classes
- Réintroduit une sorte d'héritage multiple
- Mais on maîtrise les conflits de définition de méthodes
- C'est lié à la visibilité et la portée

Classe interne

- Dans la plupart des langages : le bloc a son propre espace de nom
  - tout nom de variable a une portée débutant à sa déclaration et se terminant à la fin du bloc contenant immédiatement cette même déclaration
  - reste vrai pour tous les blocs imbriqués dans cette portée
  - la visibilité permet de cacher un nom préalablement défini dans un bloc englobant avec une nouvelle définition et un nouveau contenu

Classe interne

- Approche essentiellement des langages non orientés (code important)
- Inutile en OO (code éclaté)
- L'unité logique de définition de nom ne sera plus le bloc mais la classe

Classe interne

- Exemple

```
class Inner {
      Inner () {
            new C();   //  remarquons que l'appel se fait
                       //  avant la  défi nition  de la  classe
      }

      class C {
            boolean isWonderfull() {return true;}
      }
}
```

Classe interne

- Exemple

```
class Inner {
      int  i=0;
      Inner () {
            System.out.println( i );                    // imprime 0
            System.out.println(new C().i );             // imprime 1
      }

      class C {
            int  i= 1;
            boolean isWonderfull() {return true;}
      }
}
```

Classe interne

- Exemple

```
class Inner {
    Inner () {
        System.out.println(isWonderfull ());          //  false
        System.out.println(new C().isWonderfull ());  //  true
    }

    class C {
        boolean isWonderfull() {return true;}
    }

    boolean isWonderfull() {return false;}
}
```

Classe interne

- Exemple

```
class Inner {
    Inner () {
        System.out.println(isWonderfull ());          // false
        System.out.println(new C().isWonderfull ());  // false
        System.out.println(new C().trans ());         // false
    }

    class C {
        boolean trans() {return isWonderfull ()}
    }

    boolean isWonderfull() {return false;}
}
```

Classe interne

- Ces exemples démontrent que les méthodes d'une classe englobante restent accessibles aux classes englobées
- Réintroduit donc une forme dégénérée d'héritage multiple (limitée à deux)
- Permet d'éviter les conflits de définitions entre les deux classes héritées
- Atelier Logiciel 1 : utilisation particulièrement efficace de cette forme d'héritage multiple

Compléments

- Dans les notions vues en début d'année nous avons parfois
  - laissé certaines constructions de côté (ex : **switch**)
  - simplifié certaines notions (ex : **for**)
- Nous y revenons maintenant
- On y intègre les notions vues par la suite (ex : les classes)

Compléments

- Lien entre bloc et instruction
- Les variables locales
- Les instructions
- Les classes locales
- Les expressions

Lien entre bloc et instruction

- Toutes les instructions (*statement*) constituant un programme sont contenues dans au moins un bloc (*block*)
- Un bloc est exécuté de **la première à la dernière** instruction de ce bloc
- Les instructions sont exécutées pour leur **effet** mais n'ont **pas de valeur**

Lien entre bloc et instruction

Block :
{ *BlockStatements*(opt) **}**

BlockStatements :
BlockStatement
BlockStatements *BlockStatement*

BlockStatement :
ClassDeclaration
LocalVariableDeclarationStatement
Statement

- Ces trois dernières alternatives peuvent donc apparaître intermixées

Les variables locales

- Nous avons déjà abondamment vu la déclaration des variables locales
- Reprécisons
  - la notion de portée (scope)
  - Le problème de l'initialisation
  - Les contraintes de visibilité
  - La grammaire

Les variables locales

- Le scope d'une variable locale est, à partir de sa déclaration, le reste du bloc où elle est définie y compris le code imbriqué et le reste de sa déclaration

```
public class MyClass1 {
   public MyClass1() {
      int  i = 5;
      for(int  j=0; j  < 2;  i=i+1) {
            j= j*i ;  // ok
      }
   }
}
```

Les variables locales

```
public class MyClass1 {
    public MyClass1() {
        int  i=0,  j= i ;          // ok
    }
}
```

```
public class MyClass1 {
    public MyClass1() {
        int  i=j ,  j = 0;         // non
    }
}
```

Les variables locales

- Avant de pouvoir être utilisée, le compilateur s'assure qu'une variable locale a bien été explicitement initialisée

```
public class MyClass1 {
   public MyClass1() {
      int  i ,  j=i;          // compile time error
   }
}
```

Les variables locales

```
public class MyClass1 {
   public MyClass1() {
      int  i ,  j=0;
      if  ( j==0)
         i=1;
      System.out.println( i );  // compile time error
   }
}
```

Les variables locales

- Une variable locale ne peut-être définie dans le scope d'une autre variable locale de même nom
- (nous verrons une exception avec les classes locales)

```
public class MyClass1 {
   public MyClass1() {
      int i;
      for(int j = 0; j < 2; i++) {
         int i;                    // compile time error
      }
   }
}
```

Les variables locales

```
public class MyClass1 {
   public MyClass1() {
      int i;
      for(int i = 0; i < 2; i++); // compile time error
   }
}
```

Les variables locales

- Ceci est également vrai pour les paramètres
- La portée d'un paramètre est toute la méthode où elle intervient
- Pas question d'y redéfinir une variable de même nom

```
public fct(int var) {
    int var; // compile time error
    char var; // compile time error
}
```

Les variables locales

- Une variable locale peut toujours être définie dans le scope d'un attribut de même nom
- Dans ce cas la variable locale cache l'attribut
- Ce dernier peut être malgré tout invoqué en préfixant le nom de la variable par **this.** (ou le nom de sa classe s'il s'agit d'un attribut **static**)

Les variables locales

```
public class MyClass1 {
   int  i = 1;
   public MyClass1() {
      System.out.println( i );          // affi che : 1
      int  i=0;
      System.out.println( i );          // affi che : 0
      System.out.println(this. i );     // affi che : 1
   }
}
```

Les variables locales

```
public class MyClass2 {
   static  int  i = 1;
   public MyClass2() {
      System.out.println( i );          // affi che : 1
      int  i=0;
      System.out.println( i );          // affi che : 0
      System.out.println(MyClass2.i);       // affi che : 1
   }
}
```

Les variables locales

- Revoyons la grammaire

LocalVariableDeclarationStatement :
    LocalVariableDeclaration **;**

LocalVariableDeclaration :
    final(opt) *Type VariableDeclarators*

VariableDeclarators :
    VariableDeclarator
    VariableDeclarators **,** *VariableDeclarator*

VariableDeclarator :
    VariableDeclaratorId
    VariableDeclaratorId **=** *VariableInitializer*

Les variables locales

VariableDeclaratorId :
Identifi er
VariableDeclaratorId **[ ]**

VariableInitializer :
Expression
ArrayInitializer

- Remarquons la forme *C like* de la déclaration de tableau qui s'additionne à la forme Java
- **int** [][] tableau ≡ **int** [] tableau[] ≡ **int** tableau [][]
- Les formes mixtes et *C like* sont tolérées mais déconseillées

Les instructions

- L'instruction vide
- Les expressions-instructions
- L'instruction étiquettée
- L'instruction break
- L'instruction continue
- L'instruction switch
- L'instruction for

Les instructions

Statement :

- *Block*
- *EmptyStatement*
- *LabeledStatement*
- *BreakStatement*
- *ContinueStatement*
- *ExpressionStatement*
- *IfThenStatement*
- *IfThenElseStatement*
- *SwitchStatement*
- *WhileStatement*
- *DoStatement*
- *ForStatement*
- *ReturnStatement*

- *SynchronizedStatement*
- *ThrowStatement*
- *TryStatement*

L'instruction vide

- Cette instruction ne fait rien et se déroule toujours bien

EmptyStatement :
    ;

Les expressions instructions

- Nous avons déjà parlé d'**instruction** et d'**expression**
- En fait, certaines expressions peuvent devenir des instructions
- Vous en avez déjà rencontrées
  - ex : une assignation est une expression-instruction

Les expressions instructions

- Référons nous à la grammaire

ExpressionStatement :
StatementExpression **;**

- Une expression devient une instruction si elle est suivie de ;
- En Java, une instruction n'a pas de valeur
- Dès lors, sa valeur est perdue

Les expressions instructions

- Toutes les expressions ne peuvent pas devenir des instructions

---

StatementExpression :

Assignment
PreIncrementExpression
PreDecrementExpression
PostIncrementExpression
PostDecrementExpression
MethodInvocation
ClassInstanceCreationExpression

L'assignation

- Nous vous avons présenté l'**assignation** comme étant une instruction
- C'est en fait une **expression**
- Elle a donc une valeur :
  - La valeur de la variable assignée (après assignation)
- Une assignation peut donc intervenir comme élément d'une autre expression
- Elle a une priorité faible
- Elle est associative de *droite à gauche*

L'assignation

- Ceci explique pourquoi on peut écrire

```
i = j = k = l = 0;
i = ( j = i+j ) + 1;
while( ( i=i −1) != 0 ) ;
f( i=1,j=0);
```

- Par contre, ceci est incorrect

```
i = j = k = l = 0                  // erreur compilation
while( i=i −1 ) ;                  // erreur compilation
while( ( i=i −1) != 0 ; ) ;        // erreur compilation
```

- N'abusons pas de cette possibilité : moins grande lisibilité

L'assignation

- Que peut-on trouver comme partie gauche d'une assignation ?

LeftHandSide :
- *Identifier*
- *FieldAccess*
- *ArrayAccess*

- Exemples : `brol`, `brol.champ`, `brol[i]`, `brol[i].champ`, `brol.champ[i]`, ...

L'assignation

- Il existe des formes simplifiées de l'assignation
- var += expr équivaut à var = var + expr
- Ex : i+=1 pour incrémenter i
- Peut être utilisé pour tous les opérateurs arithmétiques vus
- Ex : d∗=2 pour multiplier d par 2
- Que penser de ?

```
i = 2;
i = i = ( i∗=2) + 1;
(i+1) −= 2;
```

Post et pré incrémentation/décrémentation

- L'opérateur ++ permet d'incrémenter une variable
- Peut être promu en instruction
- Ex : i++; ≡ i+=1; ≡ i=i+1;
- Il existe aussi −− pour décrémenter une variable
- Peut se placer avant ou après la variable
- Ex : i−−; ≡ −−i; ≡ i−=1; ≡ i=i−1;
- Il existe tout de même une différence

Post et pré incrémentation/décrémentation

- Différence entre i++ et ++i
- Placés **devant** la variable : la valeur de celle-ci est incrémentée de 1 **avant** d'être utilisée
- Placés **derrière** la variable : la valeur de celle-ci est incrémentée de 1 **après** avoir été utilisée
- Exemples

```
int i = 5;
i = i++;
i = ++i;
i = i++ + ++i;
i = (i++)++;      // Erreur !
i = 2++;          // Erreur !
```

Post et pré incrémentation/décrémentation

- Comment comprendre ceci ?

```
i = i++ + i ;      // OK
i = i+ ++ i ;      // OK
i = i+++1;         // OK
i = i++ + ++ i ;   // OK
i = i+++++i;       // Erreur !
```

- Aide : penser au fonctionnement du compilateur

Tableau des priorités et associativités

- De la plus grande à la plus petite
- Associativité de gauche à droite sauf pour les assignations

postfi xes unaires	(params), ., expr++, expr−−
préfi xes unaires	++expr, −−expr, −, +, !
création et de cast	**new**, (type)
multiplicatif	∗, /, %
additif	−, +
relationnels	<, >, <=, >=
égalité	==, !=
et	&&
ou	||
assignations	=, +=, −=, ∗=, /=, %=

Ordre d'évaluation

- Comment comprendre ceci ?

```
i = 2;
i = (i=3) * i;
i = i * ++i;
```

- C'est un problème épineux dans beaucoup de langages
- En Java, l'**évaluation est garantie de gauche à droite** pour les opérandes d'un opérateur binaire

Ordre d'évaluation

- C'est aussi le cas pour les paramètres d'un appel de méthode

```
i = 2;
f(i++,--i); // équivaut à f(2,2)
f(--i,i++); // équivaut à f(1,1)
```

Les expressions instructions

- Où en sommes-nous ?

StatementExpression :

- *Assignment*
- *PreIncrementExpression*
- *PreDecrementExpression*
- *PostIncrementExpression*
- *PostDecrementExpression*
- *MethodInvocation*
- *ClassInstanceCreationExpression*

Appel de méthode

- L'**appel de méthode** est un autre cas d'expression-instruction
- Explique pourquoi ceci est valable

```
a = f (1);    // l'appel est une expression
f (1);        // ici aussi mais ensuite promu en instruction
```

- Notons que le type de retour est quelconque pas seulement void
- La valeur est alors **perdue**

Instanciation

- L'**instanciation** est un dernier cas d'expression-instruction
- Exemple

```
new Brol();
```

- On instancie la classe (l'objet est créé) mais sans utiliser la référence à l'objet
- C'est d'un intérêt limité

Les instructions

- L'instruction vide
- Les expressions-instructions
- L'instruction break
- **L'instruction étiquettée**
- L'instruction continue
- L'instruction switch
- L'instruction for

Etiquette

- Toute instruction peut recevoir une **étiquette** (*Label* en anglais)

LabeledStatement :
Identifi er **:** *Statement*

- N'est connue que dans l'instruction qui la suit
- Permettra de quitter brutalement (**break**) ou de réitérer (**continue**) l'instruction

L'instruction `break`

- Pour **arrêter brutalement une instruction**
- Forme générale

BreakStatement :
`break` *Identifier* (opt) **`;`**

- Identifier : étiquette préalablement définie
- Etiquette précisée : passe le contrôle à l'instruction suivant l'instruction étiquetée par *Identifier*
- Pas d'étiquette, passe le contrôle à l'instruction suivant la première instruction **while**/**for**/**do**/**switch** englobante

L'instruction break

- Exemples

```
int nb;
entrée : do {
        nb = Lire.intData();
        if (nb>0)
            break entrée;
        System.out.println("Mauvais nb. Recommencer.");
}
```

```
int nb;
do {
        nb = Lire.intData();
        if (nb>0) break;
        System.out.println("Mauvais nb. Recommencer.");
}
```

L'instruction break

- Exemple

```
int  i = 1;
lab1  : { if  ( i  == 1)  break lab1 ; System.out.println (2);}
System.out.println (3);          // affi che : 3
```

- Comment comprendre ceci ?

```
int  i = 1;
lab1  :  if  ( i  == 1)  break lab1 ; System.out.println (2);
System.out.println (3);
```

```
int  i = 1;
if  ( i  == 1)  break lab1 ; System.out.println (2);
System.out.println (3);
```

L'instruction `continue`

- Pour une **nouvelle itération d'une boucle**
- Forme générale

ContinueStatement :
`continue` *Identifi eropt* (opt) **`;`**

- Etiquette précisée
  - Doit être une répétitive
  - Passe le contrôle à l'**itération suivante**
- Pas d'étiquette : passe le contrôle à l'**itération suivante** de la première instruction répétitive englobante

L'instruction continue

- Exemples

```
for (int i=0; i<10; i++)
{
    if (i%2==0) continue;
    System.out.println(i);
}
```

```
blci : for (int i=0; i<10; i++) {
    blcj : for (int j=0; j<10; j++) {
        if ( (i*j)%2==0 ) continue bcli;
        System.out.println(j);
    }
    System.out.println(i);
}
```

L'instruction `continue`

- Les instructions **break** et **continue** sont à utiliser avec parcimonie
- Souvent, cela n'aide pas à la lecture du code

Le selon que

- Il existe une instruction proche du **selon que** : **switch**
- Sorte de **if-then-else**
  - généralisé à un nombre indéterminé d'alternatives
  - avec des conditions restreintes

Le selon que

- Exemple :

```
switch (jour ) {
   case 0 : System.out.println("Lundi" ); break;
   case 1 : System.out.println("Mardi" ); break;
   case 2 : System.out.println("Mercredi" ); break;
   case 3 : System.out.println("Jeudi" ); break;
   case 4 : System.out.println("Vendredi" ); break;
   case 5 : System.out.println("Samedi"); break;
   case 6 : System.out.println("Dimanche"); break;
}
```

Le selon que

- La grammaire est assez longue

SwitchStatement :
`switch (` *Expression* **`)`** *SwitchBlock*

SwitchBlock :
`{` *SwitchBlockStatementGroups*(opt)
SwitchLabels(opt) **`}`**

SwitchBlockStatementGroup :
SwitchLabels BlockStatements

SwitchLabel :
`case` *ConstantExpression* **`:`**
`default :`

Le selon que

SwitchBlockStatementGroups :
  SwitchBlockStatementGroup
  SwitchBlockStatementGroups SwitchBlockStatementGroup

SwitchLabels :
  SwitchLabel
  SwitchLabels SwitchLabel

BlockStatements :
  BlockStatement
  BlockStatements BlockStatement

Le selon que

- L'expression à évaluer est de type limité : **char**, **byte**, **short** ou **int**
- Les expressions constantes sont toutes différentes
- **`default`** peut être omis
- Si il apparaît, il doit être unique
  (on recommande de le mettre à la fin)

Le selon que

- Cette instruction a la propriété de **percolation**
- Si une entrée est vérifiée, toutes les instructions relatives aux entrées suivantes seront aussi exécutées

```
switch (nbFois) {
   case 3 : System.out.println("Hello");
   case 2 : System.out.println("Hello");
   case 1 : System.out.println("Hello");
}
```

- Une instruction **break** peut forcer l'arrêt

Le selon que

- Plusieurs **case** peuvent être associés

```
switch(mois) {
   case 1:
   case 3:
   case 5:
   case 7:
   case 8:
   case 10:
   case 12: System.out.println("31 jours"); break;
   case 4:
   case 6:
   case 9:
   case 11: System.out.println("30 jours"); break;
   case 2:  System.out.println("28 jours"); break;
   default: System.out.println("numéro incorrect");
}
```

Le selon que

- Reprenons un point de grammaire

SwitchBlockStatementGroup :
SwitchLabels BlockStatements

- Indique que les cas sont en fait des blocs
  - On peut y déclarer des variables
  - Elles seront locales à ce bloc
    - Utiles si le code dans du case est important
    - Mais dans ce cas, il y a probablement mieux qu'un **switch**

Le selon que

- Attention, le **switch** n'est pas toujours le plus élégant ou le plus efficace
- Exemple

```
switch (jour ) {
   case 0 : System.out.println("Lundi" ); break;
   case 1 : System.out.println("Mardi" ); break;
    ... }
```

- peut s'écrire

```
String []  nomJours = {"Lundi","Mardi"  ,...};
System.out.println (  nomJours[jour] );
```

Le for

- A la différence du **while** et du **do**, présente l'avantage syntaxique de ramener dans l'en-tête de l'instruction les trois phases d'une boucle de contrôle :
  - l'initialisation
  - le test
  - l'incrémentation (au sens large)
- N'a de sens que si ces trois phases restent simples à exprimer
- Sinon, on préférera utiliser **while** ou **do**

Le for

ForStatement :
`for (` *ForInit*(opt) **`;`** *Expression*(opt) **`;`** *ForUpdate*(opt) **`)`**
Statement

ForInit :
StatementExpressionList
LocalVariableDeclaration

ForUpdate :
StatementExpressionList

StatementExpressionList :
StatementExpression
StatementExpressionList **`,`** *StatementExpression*

Le for

- La phase d'initialisation
  - N'accepte qu'une seule déclaration
  - Ou une liste de *StatementExpression* séparés par une virgule
  - Pas question d'y mettre un block ou une instruction d'itération
  - Exemples

```
for( int i=0; ...
for( i=0, j=n ; ...
for( int i=0, j=n ; ...   // Erreur !
```

Le for

- La phase d'incrémentation
  - est encore plus limitée puisqu'elle ne peut inclure aucune déclaration
  - Exemple

```
for ( i=0, j=n−1; i<n; i++, j −− ) ...
```

- La phase de test (Expression)
  - Doit être de valeur booléenne.
  - Est optionnelle et dans ce cas vaut **true**

Le for

- Ordre d'exécution de l'instruction pour
  1. la phase d'initialisation : *ForInit*
  2. l'évaluation du test : *Expression*
  3. si le test vaut true alors
     (a) le corps du for : *Statement*
     (b) la phase d'incrémentation : *ForUpdate*
     (c) retour au début de l'étape 3
  4. sinon, on passe le contrôle à l'instruction suivant immédiatement le for

Le for

- Il est intéressant de vérifier que l'on peut aisément traduire une boucle for en une boucle while :

```
ForInit
while(Expression) {
   Statement
   ForUpdate
}
```

- où les listes d'instructions de ForInit et ForUpdate sont transformées en une séquence d'instructions

Le for

- Exemples

```
for (;;) System.out.println("Je␣ne␣me␣fatigue␣pas␣!");
```

```
int [] tab = {1,2,3,4,5};
int i,j;
int n = tab.length;
for(i=0,j=n−1; i<n/2; i++,−−j) {
   int t = tab[i];
   tab[i] = tab[j];
   tab[j] = t;
}
```

Compléments

- Lien entre bloc et instruction
- Les variables locales
- Les instructions
- **Les classes locales**
- Les expressions

Les classes locales

- Une déclaration de classe peut également se faire dans un *block*
- En général le corps d'une fonction
- Permet de définir une classe pour un besoin local
- Sera utilisé en Atelier Logiciel

Les classes locales

- La portée (*scope*) d'une classe locale est le reste du *block* et sa propre déclaration
- Exemples

```
public class MyClass1 {
    public MyClass1() {
        new C();                    // compile time error
        class C{};
    }
}
```

Les classes locales

```
public class MyClass1 {
   public MyClass1() {
      class C{};
      new C();                    // ok
   }
}
```

```
public class MyClass1 {
   public MyClass1() {
      class C{};
      {
            new C();              // ok
      }
   }
}
```

Les classes locales

- On ne peut pas redéfinir une classe locale dans le même bloc et au même niveau

```
public class MyClass1 {
   public MyClass1() {
      class C{};
      class C{};              // compile time error
   }

public class MyClass1 {
   public MyClass1() {
      class C{};
      {
            class C{};        // ok
      }
   }
```

Les classes locales

- La définition d'une classe locale cache toute classe de même nom définie dans le code englobant

```
public class MyClass1 {
   public MyClass1() {
      class C{};
      {
            class C{};
            new C();        // instancie  la  deuxième classe
      }
   }
```

Les classes locales

- Rappelons-nous une exception avec les variables locales
- Une variable locale peut-être définie dans le scope d'une autre variable locale de même nom si elle appartient à une nouvelle classe locale
- Dans ce cas la nouvelle variable locale cache la précédente

Les classes locales

```
public class MyClass1 {
   public MyClass1() {
      int  i = 1;
      class C {
               int  foo (){ int  i = 0;return  i ;};
      }
      System.out.println(new C().foo ());
   }
   public static  void main(String [] args){
      new MyClass1();                    //  affi che : 0
   }
}
```

Compléments

- Lien entre bloc et instruction
- Les variables locales
- Les instructions
- Les classes locales
- **Les expressions**

L'expression conditionnelle

- Il existe un opérateur proche du **if-then-else**
- condition ? casVrai : casFaux
- La valeur de cette expression est
  - casVrai si condition est vraie
  - casFaux si condition est fausse
- Toutes les alternatives doivent être de même type
- Exemple

```
int abs(int n) {
    return n<0 ? -n : n;
}
```

L'expression conditionnelle

- La syntaxe précise est

ConditionalExpression :
ExpressionExceptConditionalExpression **?**
Expression **:** *ExpressionExceptConditionalExpression*

- L'expression conditionnelle ne brille pas par sa clarté dès que le test est un peu compliqué (à usage limité donc !)

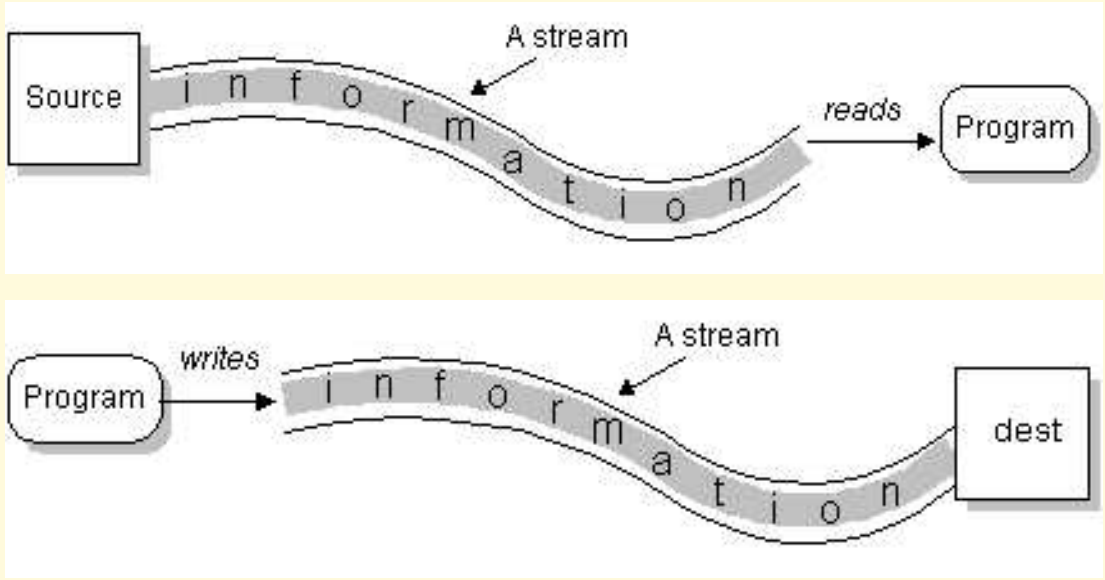
Tableau des priorités et associativités

- De la plus grande à la plus petite
- Associativité de gauche à droite sauf pour les assignations

postfi xes unaires	(params), ., expr++, expr−−
préfi xes unaires	++expr, −−expr, −, +, !
création et de cast	**new**, (type)
multiplicatif	∗, /, %
additif	−, +
relationnels	<, >, <=, >=
égalité	==, !=
et	&&
ou	||
conditionnel	?:
assignations	=, +=, −=, ∗=, /=, %=

Les entrées-sorties

- Ecrire et lire se font en direction ou à partir de supports quelconques et hétérogènes :
  - fichiers, mémoire, socket, application, . . .
- Encapsulé dans le concept de **stream** (**flot**)

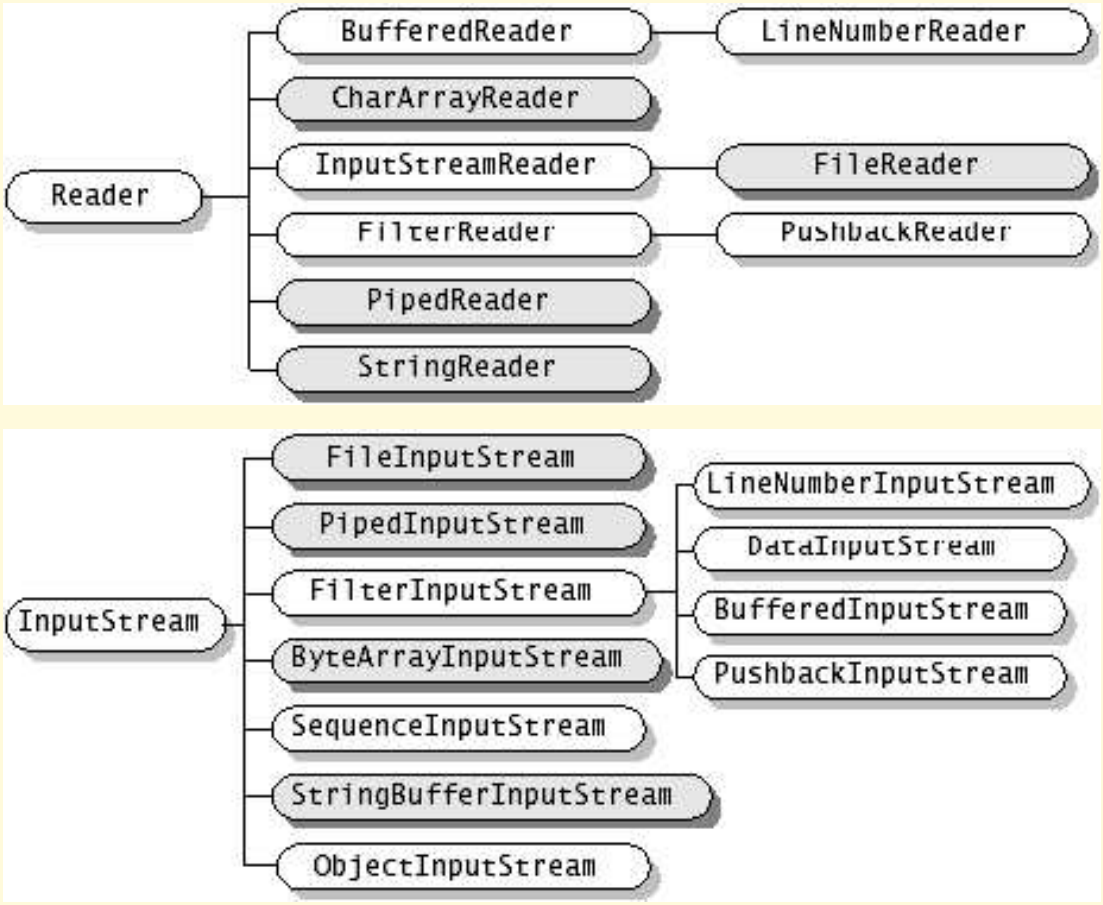


source : http :// java . sun.com/docs/books/tutorial/essential/ io / overview.html

Les entrées-sorties

- Package java.io
  - Large palette de **streams**
  - Répartis en deux grandes familles
    - **Binaire** : manipulant des **bytes**
      Classes abtraites InputStream/OutputStream et ses dérivées
    - **Texte** : manipulant des **caractères** (2 bytes)
      Classes abtraites Reader/Writer et ses dérivées

Les entrées-sorties



source : http :// java . sun.com/docs/books/tutorial/essential/ io / overview.html

Les entrées-sorties

Reader

- public int read()
- public int read(char [] caractereBuffer)
- public abstract int read(char [], int offset , int length)
- public long skip(long nombreCaractere)
- public void mark(int nombreCaractere)
- public void reset()
- public boolean markSupported()
- public boolean ready()
- public abstract void close()

Les entrées-sorties

Writer

- public void write(char [] caracteres)
- public abstract write(char [] caracteres,
  int offset , int length)
- public void write(int codeCaractere)
- public void write(String)
- public void write(String string , int offset , int length)
- public abstract void flush()
- public abstract void close()

Les entrées-sorties

- InputStream : idem Reader
  - sauf available () qui remplace ready()
- OutputStream : idem Writer
  - moins write(String) et write(String, int , int)

Les entrées-sorties

Différentes implémentations de base disponibles

- InputStream
  - FileInputStream
  - PipedInputStream
  - ByteArrayInputStream
  - StringBufferInputStream
- OutputStream
  - FileOutputStream
  - PipedOutputStream
  - ByteArrayOutputStream

Les entrées-sorties

- FileInputStream
  - public FileInputStream(String fileName)
  - protected void finalize ()
- FileOutputStream
  - public FileOutputStream(String fileName)
  - protected void finalize ()

Les entrées-sorties

- ByteArrayInputStream
  - public ByteArrayInputStream(byte[] bytes)
- ByteArrayOutputStream
  - public ByteArrayOutputStream(int length)
  - public ByteArrayOutputStream()
  - public synchronized void reset()
  - public synchronized byte[] toByteArray()
  - public String toString()

Les entrées-sorties

- StringBufferInputStream : idée similaire à TestByteArrayInputStream
- StringBufferOutputStream : n'existe pas
- PipedInputStream : lié aux pipes (cf. cours de système)
- PipedOutputStream : idem

Les entrées-sorties

Enfin passons en revue les streams plus sophistiqués

- InputStream
  - FilterInputStream
    - LineNumberInputStream
    - DataInputStream
    - BufferedInputStream
    - PushBackInputStream
  - ObjectInputStream

Les entrées-sorties

- OutputStream
  - FilterOutputStream
    - DataOutputStream
    - BufferedOutputStream
    - PrintStream
  - ObjectOutputStream

Les entrées-sorties

- FilterInputStream et FilterOutputStream
  - Le inputStream décoré est stocké dans un attribut
  - Toutes les méthodes sont déléguées par défaut vers cet inputStream
  - protected FilterInputStream(InputStream inputStream)
- BufferedInputStream et BufferedOutputStream
  - Utilise une zone tampon pour la lecture/écriture
  - Optimisisation des opérations (rapidité)
  - Permet de revenir en arrière

Les entrées-sorties

- DataInputStream
  - Permet des lectures de types primitifs (en binaire)
  - public final boolean readBoolean()
  - public final char readChar()
  - public final double readDouble()
  - public final int readInt()
  - ...
- DataOutputStream : symétrique pour l'écriture

Les entrées-sorties

Reader/Writer

- Très similaire à la famille précédente mais concernant les caractères
- InputStreamReader établit la possibilité de créer un Reader stream à partir d'un InputStream stream
  - public InputStreamReader(InputStream inputStream)
- OutputStreamReader établit la possibilité de créer un Writer stream à partir d'un OutputStream stream
  - public OutputStreamReader(OutputStream ouputStream)

Les entrées-sorties

La classe Lire

- Voyons comment tout cela a été utilisé pour la classe Lire que vous utilisez

Les exceptions

- Code d'une méthode : traitement *normal* des données
  - paramètres ok
  - environnement n'ayant aucune défaillance
- Voeux pieux et . . .périlleux
  - On ne sait pas qui va l'utiliser ni comment
    - Bonne documentation ne suffit pas (est-on sûr qu'elle va être lue ?)
  - On ne contrôle pas l'environnement

Les exceptions

Exemple

- Une méthode dont un paramètre est l'âge d'une personne
- La valeur du paramètre doit être valide (càd une valeur entière comprise entre 0 et 120 , bornes comprises).
- Le typage d'un paramètre (**int**) permet d'imposer une partie de la contrainte
- Pour le reste ($0 \leq$ âge ≤ 120) seul un test explicite au début du code permet de le vérifier

Les exceptions

- Vérification assez lourde mais nécessaire pour garantir un respect effectif des contraintes
- Que faire si l'âge passé est invalide (150 par ex) ?
  - on dynamite toute l'application avec un abort (souvent inadmissible)
  - valeurs/traitements par défaut + log (l'utilisateur va-t-il s'en rendre compte ?)
  - valeur de retour indiquant le problème (une seule valeur de retour !)
  - le **mécanisme des exceptions**

Les exceptions

- Le mécanisme des exceptions
  - Possibilité d'**interrompre** à tout moment l'exécution d'un morceau de code en difficulté
  - **Rend la main** au code appelant
  - A un **endroit prévu** pour gérer ce type de défaillance
- Quand utiliser ce mécanisme ?
  - le code en difficulté ne sait pas comment résoudre le problème
  - pas pour une fin de fichier par exemple

Les exceptions

- Repose sur un ensemble d'objets **Throwable**
- Peuvent être projetés dans le code appelant
- Au sein d'une instruction prévue pour le recevoir
- Interrompant le code en cours d'exécution
- Projection réalisée au moyen de l'instruction **throw**
- Récupération au moyen de **try and catch**
- Similaire et parallèle au mécanisme du **return**

Les exceptions

- Cette classe est standard : java.lang.Throwable
- Sous-classe
  - Exception : regroupe toutes les exceptions **contrôlées** par le compilateur
    - Le compilateur va s'assurer que toute méthode levant une exception controlée le déclare
    - De même, tout appel de méthode susceptible de lever une exception controlée se fera dans le cadre d'une instruction **try and catch**

Les exceptions

- Autres sous-classes
  - RunTimeException
    (ex : ArrayIndexOutOfBoundsException)
    - ne doivent pas être déclarées dans la clause throw de la méthode
  - Error : exceptions liées au dysfonctionnement de la Virtual Machine
    - Déconseillé d'essayer de les traiter
    - Ne doivent pas être déclarées dans la clause throw de la méthode

Les exceptions

- Lancement d'une exception : au moyen de l'instruction **throw**
- Exemple

```
public class B {
   public void leveB() throws MyException {
         throw new MyException("leveB");
   }
}
```

- On doit spécifier les exceptions potentiellement lancées (clause **throws**)
- Si dans un constructeur, l'objet n'est pas construit

Les exceptions

- Récupération de l'exception
- Tout appel à une méthode susceptible de lever une Exception devra être incluse dans une instruction **try and catch**
- Partie **try** : recueille le code susceptible de lever une exception
- Suivie d'un nombre indéterminé de **catch**
- Chaque catch contient le code prévu pour réagir aux différentes exceptions

Les exceptions

- Exemple

```
public class B {
   public void foo(){
      try{
         B b= new B();
      } catch (MyException e){
          System.out.println("hum! some problems");
      }
}
```

Les exceptions

- Si plusieurs catch, on exécute le premier qui capture la bonne exception
- Exemple

```
public class B {
   public void foo(){
      try{
         B b= new B();
      } catch (Exception e) {
          System.out.println("hum! some problems");
      } catch (MyException e) {
          System.out.println("Jamais exécuté");
      }
}
```

Les exceptions

- La clause **finally** est exécutée dans tous les cas
- Exemple

```
public class B {
   public void foo(){
      try{
         B b= new B();
      } catch (MyException e){
          System.out.println("hum! some problems");
      } finally {
          System.out.println("fin de foo");
      }
}
```

Les exceptions

- Si exception potentielle capturée par aucun catch, la méthode englobante doit déclarer cette exception comme pouvant être lancée par elle
- Exemple

```
public class B {
   public void foo() throws MyException {
      try{
         B b= new B();
      }finally{
          System.out.println("fin de foo");
      }
}
```

Les exceptions

- On peut créer ses propres types d'exceptions
- Exemple

```
public class MonException extends Exception {
}
```

Crédits

Ce document a été produit avec les outils suivants

- **LaTeX** comme système d'édition
- La classe **Prosper** pour les transparents
- Les packages **pst-grad**, **listings**, **fancyvrb** et **atbeginend**
- **Linux** comme système d'exploitation
- **Kile** comme environnement de développement
- **ps2pdf** pour la conversion en **PDF**
- **acroread** pour la visualisation