

Le Langage Java

1ère année

M. Bastreggi J. Beleho P. Bettens M. Codutti
A. Hallal C. Leruste D. Nabet N. Pettiaux A. Rousseau

Haute École de Bruxelles — École Supérieure d'Informatique

Année académique 2011 / 2012

Liste des leçons — Quadrimestre I

- | | |
|------------------------------------|-------------------------------|
| 1 Objectifs, moyens et évaluations | 13 Les boucles (survol) |
| 2 Programme et langage | 14 Écrire du code robuste |
| 3 Et Java dans tout ça ? | 15 Les types et les littéraux |
| 4 Développer en Java | 16 Les tableaux (survol) |
| 5 Algorithmes séquentiels (survol) | 17 La documentation Java |
| 6 L'erreur est humaine | 18 Tester le code |
| 7 Alternatives (survol) | 19 Variables locales |
| 8 La notion de grammaire | 20 Les expressions |
| 9 La grammaire lexicale de Java | 21 Les assignations |
| 10 Écrire du code lisible | 22 Instructions |
| 11 Code modulaire (survol) | 23 Les tableaux |
| 12 Organiser le code | |

Liste des leçons — Quadrimestre II

Les slides pour le second quadrimestre seront disponibles en janvier. Au programme :

- ▶ L'orienté objet
- ▶ Les listes
- ▶ Les exceptions
- ▶ Les entrées-sorties
- ▶ Les conversions
- ▶ Les énumérations

Leçon 1

Objectifs, moyens et évaluations

- Objectifs
- Moyens
- Évaluations

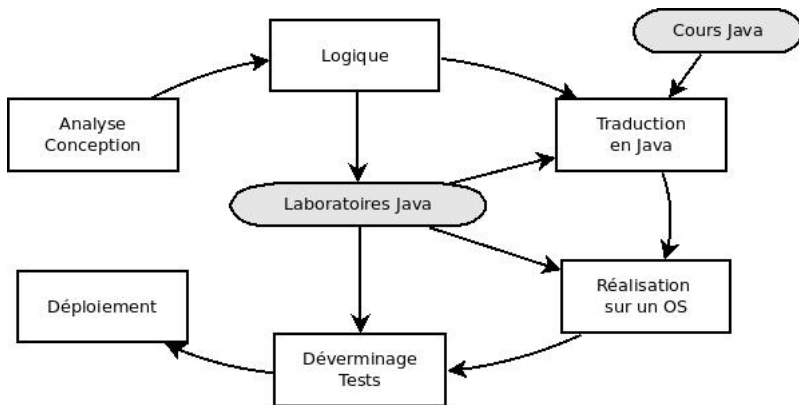
Objectif du cours

S'initier à la programmation au travers de l'apprentissage

- ▶ d'un langage (Java)
- ▶ des bons comportements
 - Programmes lisibles, robustes, documentés et testés
 - Capacités de **déverminage** (debugging)
 - Autonomie dans le travail (recherche dans la documentation)

Liens avec les autres cours

Étape dans le développement d'un *Système d'Information*



Objectif secondaire

Familiarisation avec le **système d'exploitation**
(*Operating System*) **Linux**

- ▶ Pas vu au cours
- ▶ Mais utilisé lors des **laboratoires**
- ▶ Être capable de l'utiliser dans les tâches courantes de programmation

Les supports d'apprentissage

Pas de syllabus pour ce cours

- ▶ De nombreux livres dans le commerce
- ▶ Les transparents sont disponibles

Importance d'un **livre d'accompagnement**

- ▶ Attention : Certains se concentrent sur
 - un aspect bien particulier du langage
 - une API spécifique

Les ressources *en ligne*

De nombreuses ressources sur **poÉSI**

- ▶ la **plateforme d'apprentissage** de l'ÉSI
- ▶ elearning.esi.heb.be
- ▶ On y trouve : notes de cours, références, ...

Voir aussi le **forum**

- ▶ fora.namok.be
- ▶ Consulté par des professeurs et des élèves
- ▶ Permet
 - de se tenir au courant des dernières nouvelles
 - d'aider ou de se faire aider en cas de problème

Le problème de l'anglais

Une connaissance de l'**anglais technique** est primordiale

- ▶ Messages d'erreurs
- ▶ Documentation

Nous essayerons d'introduire chaque nouveau terme dans les deux langues.

L'évaluation

Deux **cotes différentes**

- ▶ Une pour la partie **cours**
- ▶ Une autre pour la partie **laboratoire**



On peut donc réussir une partie et pas l'autre.

L'évaluation du cours

Uniquement un **oral** en fin d'année

Qu'est-ce qu'on attend de vous ?

- ▶ **Comprendre** les concepts abordés au cours
(**pas** de «**par coeur**»)
- ▶ Pouvoir les expliquer clairement
- ▶ Pouvoir les illustrer au travers d'exemples courts
- ▶ De la **rigueur** : pas d'«à peu près»
- ▶ Du **détail** : ne pas s'arrêter à l'«usage courant»

L'évaluation des laboratoires

Évaluations continues pendant les laboratoires

- ▶ Courtes interrogations en **début de séance**
- ▶ Interrogations de **synthèse**
- ▶ Un **projet**

Un examen en fin d'année permet de **remettre en jeu** la cote d'année.

Qu'est-ce qu'on attend de vous ?

- ▶ De la **rigueur**
- ▶ De l'**autonomie**

Leçon 2

Programme et langage

- Concepts
- Historique
- Traduction

Définitions

Pouvez-vous définir les concepts suivants :

- ▶ Un **programme** ?
- ▶ **Programmer** ?
- ▶ Un **langage de programmation** ?
- ▶ Différence entre **langue** et *langage* ?

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)

Cassons un mythe

« Elle est bête cette machine,
elle s'est encore plantée ! »

Que penser de l'*intelligence* d'un ordinateur ?

Cassons un mythe

« Elle est bête cette machine,
elle s'est encore plantée ! »

Que penser de l'*intelligence* d'un ordinateur ?

Réponse de E. Dijkstra :

« Se demander si un ordinateur peut penser,
c'est aussi intéressant que se demander si un
sous-marin peut nager. »

Un programme

La seule chose dont est capable un ordinateur est de réaliser extrêmement rapidement des instructions élémentaires

Toute tâche qu'on veut lui confier doit donc être préalablement décrite comme une suite séquentielle d'instructions (un programme)

Un programme

Ex : Taille de quelques programmes

- ▶ Firefox : 2 millions de lignes de code
- ▶ Windows XP : 40 millions de lignes de code
- ▶ Mac OS X : 86 millions de lignes de code
- ▶ Distribution Debian 4 : 283 millions de lignes de code

(source : http://en.wikipedia.org/wiki/Source_lines_of_code)

Historique des langages

Langage machine : ensemble de 0 et de 1

Assemblage : abstraction des instructions

Haut niveau (60') : abstraction des expressions
(ex : COBOL, FORTRAN)

Structuré (70') : abstraction des structures de contrôle
(ex : Pascal, C)

Orienté objet (90') : abstraction des données
(ex : Eiffel, Java, C++)

Historique des langages

On a aussi les langages :

- ▶ logiques : Prolog, ...
- ▶ fonctionnels : Lisp, Scheme, ...
- ▶ orientés aspects : AspectJ, ...

Une classe de langages est adaptée à une classe de problèmes ... et ces problèmes évoluent dans le temps ...

Les langages orientés objets

Restés longtemps dans les laboratoires

- ▶ Problèmes d'efficacité
- ▶ Indigence des IDE
- ▶ Rupture avec les méthodes d'analyses

Ont explosé il y a une dizaine d'années
(crise du logiciel)

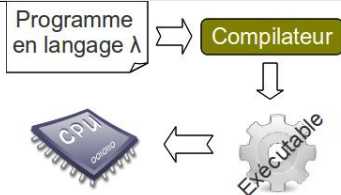
- ▶ Lien avec UML
- ▶ Écriture dans les termes du problème
- ▶ Réutilisabilité

Le problème de la traduction

Un ordinateur ne comprend que le langage machine

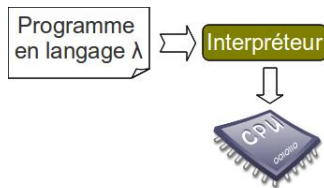
- ▶ Et un autre langage (comme Java) ?
- ▶ Nécessité d'une **traduction**

Compilation



*traduit d'une traite
avant l'exécution*

Interprétation



*traduit morceau par morceau
au moment de l'exécution*

Compilation vs interprétation

Quelle est la meilleure technique au niveau de :


- ▶ La facilité de distribution ?
- ▶ La rapidité ?
- ▶ La facilité de développement ?

Leçon 3

Et Java dans tout ça ?

- Historique
- Les éditions de Java
- Pourquoi Java ?

Historique de Java

- 92 SUN crée oak (systèmes embarqués).
Auteur : James Gosling
- 94 Adapté à Internet grâce aux **applets**.
Devient Java
- 96 Première version stable et gratuite de JDK
- 98 Sortie de Java 2
- 05 Version 1.5 de Java 2
- 09 Oracle rachète Sun (et donc Java)
- 11 Version 1.7 (Java 7, en GPL) 

Les éditions de Java

Java est disponible en 3 **éditions**

- ▶ Même langage
- ▶ Mais taille de la bibliothèque différente
- ▶ Adapté à des situations différentes

Java SE (édition standard)

- ▶ Applications monopostes classiques
- ▶ Les applications Android sont en Java

Les éditions de Java

Java ME (édition mobile - plus léger)

- ▶ Applications **embarquées** : téléphones, appareils électroniques, ...
- ▶ Omniprésent

Java EE (édition entreprise - plus complet)

- ▶ Applications distribuées : client-serveur, web
- ▶ Très présent : riche, robuste et portable
- ▶ Concurrents : .NET (Microsoft), PHP

Pourquoi Java ?

Réelles **qualités pédagogiques**

- ▶ Syntaxe claire et précise
- ▶ Typage fort
- ▶ Détection précoce des erreurs
- ▶ Concepts modernes de programmation
- ▶ Économie d'échelle

A trouvé sa place dans le milieu professionnel

Java et les autres langages

Vous verrez d'autres langages

- ▶ Assembleur en première
- ▶ C et C++ en deuxième
- ▶ Cobol en 1ère, 2ème et 3ème (Gestion)

Vous approfondirez Java

- ▶ Les ateliers logiciels, applications distribuées

Leçon 4

Développer en Java

- La machine virtuelle
- Les outils de développement

Le problème de la portabilité

Il est difficile de développer un programme *multi-OS*

- ▶ Parties de code différentes d'un OS à l'autre
- ▶ \implies ne tourne que sur un OS

Intenable pour Java (**applets**)

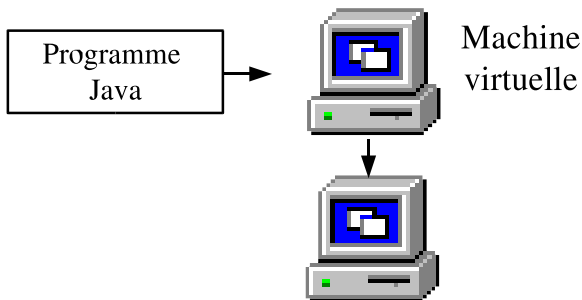
- ▶ Nécessité de développer un code **portable**

Réponse de SUN : la machine **virtuelle** (JVM)

- ▶ Programmes Java développés pour la JVM
- ▶ Comment les faire tourner sur une machine réelle?

La machine virtuelle

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



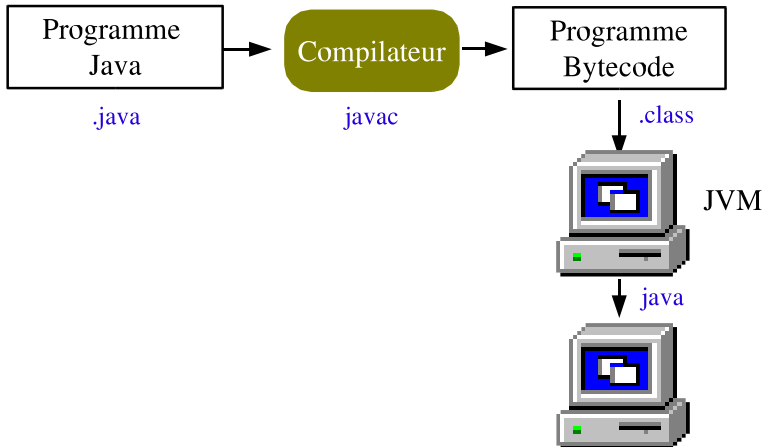
La machine virtuelle

Java serait lent à interpréter (langage de haut niveau)

- ▶ Introduction d'un niveau intermédiaire : le **Bytecode**
 - Proche d'un langage d'assemblage
 - Plus rapide à interpréter
 - C'est en fait le langage de la JVM
- ▶ Java est d'abord compilé en Bytecode

⇒ **Approche mixte** compilation/interprétation

La machine virtuelle



Exemple : premier programme

Prenons un exemple (*fichier `Hello.java`*)

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

Compilons-le `> javac Hello.java` (extrait d'une console)

On obtient la version compilée (`Hello.class`)

Exécutons-la sur la machine virtuelle

```
> java Hello
Bonjour !
```

Les outils de développement

JRE : *Java Runtime Environment*

- ▶ Ce qui est nécessaire à l'exécution
- ▶ Accompagne les navigateurs Web par ex.

JDK SE : *Java Development Kit* (standard edition)

- ▶ Permet de développer en Java
 - `javac` : compilateur Java vers bytecode
 - `java` : la machine virtuelle Java
 - `javadoc` : production automatique de documentation
 - ...
- ▶ Gratuit, fourni par ~~SUN~~ Oracle

Les outils de développement

Les *Environnement de Développement Intégré* :
Netbeans, Eclipse, ...

- ▶ Gratuits
- ▶ Avantages
 - Éditeur, compilateur, débogueur et aide intégrés dans un même outil
 - Génération automatisée de code
- ▶ Inconvénient : on maîtrise moins tout le processus (quand ça ne marche pas, on ne comprend pas pourquoi !)

Les outils de développement

Autre possibilité : les techniques brutes

- ▶ Un **éditeur avec coloration syntaxique**
- ▶ Gestion manuelle des noms et emplacements des fichiers
- ▶ Compilation et exécution en ligne de commande

*Approche choisie à l'école pour vous faire
comprendre ce qu'il y a derrière*

Leçon 5 — Algorithmes séquentiels (survol)

Nous voyons comment traduire les algorithmes séquentiels que vous écrivez au cours de Logique

- Structure générale d'un programme
- Variables et assignation
- Lire au clavier
- Constantes
- Conventions
- Commentaires

Structure générale du programme

```
public class NomClasse {  
    // Mettre ici les modules (on dit méthode en Java)  
}
```

- ▶ Le nom commence par une majuscule
- ▶ Doit se trouver dans le fichier **NomClasse.java**

Attention ! En Java : minuscule \neq majuscule

Exemple : on ne peut pas écrire

```
Public CLASS NomClasse {  
  
}
```

La méthode principale

```
public class NomClasse {  
    public static void main(String[] args) {  
        // Code de la méthode ici  
    }  
}
```

- ▶ **main** est le nom de la méthode principale
- ▶ C'est par là que commence le programme
- ▶ À écrire tel quel, on verra pourquoi

Les variables

Les types disponibles

En Logique	En Java
Entier	int
Réel	double
Chaine	String
Caractère	char
Booléen	boolean

Exemple de déclaration

```
int nb1;
```

Les calculs

L'assignation se fait via le symbole =

```
nb1 = 1;
```

Les calculs : on dispose de tous les opérateurs *classiques*

+	plus
-	moins
*	fois
/	au moins 1 réel \implies division réelle
	2 entiers \implies division entière (DIV en Logique)
%	reste (MOD en Logique)

Exemple

```
public class Moyenne {  
    public static void main(String[] args) {  
  
        int nombre1;  
        int nombre2;  
        int moyenne;  
  
        nombre1 = 34345;  
        nombre2 = -3213213;  
        moyenne = (nombre1 + nombre2) / 2;  
        System.out.println (moyenne);  
    }  
}
```

Exemple

```
public class Moyenne {  
    public static void main(String[] args) {  
  
        int nombre1 = 34345;  
        int nombre2 = -321321;  
        double moyenne;  
  
        // division réelle car un des 2 opérandes est réel  
        moyenne = (nombre1 + nombre2) / 2.0;  
        System.out.println ("La moyenne est " + moyenne);  
    }  
}
```

Lire au clavier

Moins direct que l'affichage à l'écran

- ▶ Applications modernes (graphiques)
- ▶ Lectures dans des champs de saisie
- ▶ Parfois utile : test ou apprentissage

Exemple

```
import java.util.Scanner;  
// ...  
Scanner clavier = new Scanner(System.in);  
// ...  
nombre1 = clavier.nextInt();
```


Lire au clavier - Exemple

```
import java.util.Scanner;

public class Test {
    public static void main(String[] args) {
        Scanner clavier = new Scanner(System.in);
        double nombre1;
        double nombre2;
        double moyenne;

        nombre1 = clavier.nextDouble();
        nombre2 = clavier.nextDouble();
        moyenne = (nombre1 + nombre2) / 2.0;
        System.out.println (moyenne);
    }
}
```

Lire au clavier

Pour lire...	on écrit...
un entier	<code>nextInt()</code>
un réel	<code>nextDouble()</code>
un booléen	<code>nextBoolean()</code>
un mot	<code>next()</code>
une ligne	<code>nextLine()</code>
un caractère	<code>next().charAt(0)</code>

Constante locale

Clause **final** \Rightarrow constante

Valeur donnée

- ▶ Soit à la déclaration
- ▶ Soit par assignation ultérieure

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

Pourquoi une constante au lieu d'un littéral ?

Conventions de noms

Pour une variable :

- ▶ Tout mettre en **minuscules**
- ▶ Sauf les débuts de *noms composés* en majuscule

Pour une constante :

- ▶ Tout mettre en **majuscules**
- ▶ Utiliser `_` pour séparer les mots

Dans tous les cas : **être explicite**
(sauf abréviations courantes)

Conventions de noms

Exemples

```
String nom;  
int annéeEtude;  
int nbEtudiants;  
boolean partieFinie ;  
final double PI;  
final int TAUX_TVA;
```

Le commentaire

```
// Commentaire sur une ligne  
/* Commentaire sur  
   plusieurs lignes */
```

- ▶ Destiné aux humains
- ▶ Sans effet sur le programme
- ▶ Néanmoins crucial

Leçon 6 — L'erreur est humaine

*Identifier, comprendre et corriger
les erreurs d'un programme*

La réalité de la programmation

Nous avons vu le processus **idéal**

- ▶ On édite / compile / exécute ...et tout « *va bien* »

Cas **extrêmement rare**

- ▶ Le programmeur est humain → faillible
- ▶ Vous avez sûrement déjà tous été confrontés à un logiciel qui « *se plante* »

Quels types d'erreurs rencontre-t-on ?

La réalité de la programmation

Nous avons vu le processus **idéal**

- ▶ On édite / compile / exécute ... et tout « *va bien* »

Cas **extrêmement rare**

- ▶ Le programmeur est humain → faillible
- ▶ Vous avez sûrement déjà tous été confrontés à un logiciel qui « *se plante* »

Quels types d'erreurs rencontre-t-on ?

- ▶ Erreurs de **compilation**
- ▶ Erreurs d'**exécution**
 - Le programme s'arrête
 - Le programme a un mauvais comportement

Les erreurs de compilation

Le compilateur ne comprend pas ce qu'on a écrit

- ▶ **Impossible** pour lui de le **traduire** en *bytecode*
- ▶ Messages d'erreurs pour nous aider à corriger

Exemples : versions modifiées de « *Hello World* »

```
public Class Hello {  
    public static void main(String[] args) {  
        System.out.println ("Bonjour!");  
    }  
}
```

```
> javac Hello.java  
Hello.java:1: class, interface, or enum expected  
public Class Hello {  
    ^
```

Les erreurs de compilation

```
public class Hello {  
    public static void main(string[] args) {  
        System.out.println ("Bonjour_!");  
    }  
}
```

```
> javac Hello.java  
Hello.java:2: cannot find symbol  
symbol   : class string  
location: class Hello  
    public static void main(string[] args) {  
                             ^
```

Les bonnes pratiques

Une grosse partie de la programmation consiste à **détecter / corriger** les problèmes

- ▶ Plus tôt on détecte une erreur, plus vite elle est corrigée (le rapport peut être de 1 à 10)

Aptitude à travailler au laboratoire

- ▶ C'est un **savoir-faire** plus qu'un savoir
- ▶ Suivre quelques **pratiques** qui ont fait leur preuve

Les bonnes pratiques

Application aux erreurs de compilation

- ▶ **Compiler souvent**
(pas uniquement « à la fin »)
 - Moins d'erreurs à corriger à la fois
 - Messages du compilateur plus clairs
 - Ne bâtir que sur du solide
- ▶ **Lire / comprendre** les messages
 - Idéalement, on sait déjà quel est le problème avant de retourner dans l'édition
 - Astuce : si beaucoup d'erreurs, commencer par les premières
- ▶ **Retenir / reconnaître** les erreurs fréquentes

Les erreurs d'exécution

La JVM ne **sait pas exécuter** une instruction

- ▶ S'**arrête** avec un message expliquant
 - L'instruction qui pose problème
 - La **nature du problème**
- ▶ Comme pour la compilation : **lire** / **comprendre** / **retenir** / **reconnaître** les erreurs
- ▶ Afficher des « traces » ou utiliser un « débogueur » peut aider à trouver le problème

Les erreurs d'exécution

Exemple : Une petite division

```
public class Test {  
    public static void main(String[] args) {  
        int numérateur = 1;  
        int dénominateur = 0;  
        System.out.println ( numérateur / dénominateur );  
    }  
}
```

```
> javac Test.java  
> java Test  
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at Test.main(Test.java:5)
```

Les erreurs de calcul

La réponse / le comportement n'est pas le bon

- ▶ Erreur dans notre **logique** ou notre **traduction**
- ▶ Erreurs les plus **pernicieuses**
 - Réflexe à croire que la réponse est bonne
- ▶ **Tester** son programme
 - Vérifier les réponses
 - Penser aux cas **limites**
 - Nous verrons des techniques adaptées (JUnit pour les tests unitaires)

Leçon 7 — Alternatives (survol)

Nous voyons comment traduire les algorithmes contenant des alternatives que vous écrivez en logique

Instructions de choix

Le **Si**

```
if ( condition ) {  
    instructions  
}
```

Le **Si-sinon**

```
if ( condition ) {  
    instructions  
} else {  
    instructions  
}
```

Exemple

```
import java.util.Scanner;
public class Test {
    public static void main(String[] args) {
        Scanner clavier = new Scanner(System.in);
        int nombre1;

        nombre1 = clavier.nextInt();
        if (nombre1 < 0) {
            System.out.println (nombre1 + " est négatif");
        }
    }
}
```

Exemple

```
import java.util.Scanner;
public class Test {
    public static void main(String[] args) {
        Scanner clavier = new Scanner(System.in);
        int nombre1;

        nombre1 = clavier.nextInt ();
        System.out.println (nombre1 + " est un nombre ");
        if (nombre1 < 0) {
            System.out.println (" négatif ");
        } else {
            System.out.println (" positif ");
        }
    }
}
```

Exercice

Comment traduire cet algorithme?

```
MODULE Test
  nombre1: Entier
  LIRE nombre1
  SI nombre1 > 0 ALORS
    ECRIRE nombre1, "est positif"
  SINON
    SI nombre1 = 0 ALORS
      ECRIRE nombre1, "est nul"
    SINON
      ECRIRE nombre1, "est négatif"
    FIN SI
  FIN SI
FIN MODULE
```

Expressions booléennes

Pour les tests, on peut utiliser :

- ▶ Des comparateurs : `<`, `>`, `<=`, `>=`, `==`, `!=`
- ▶ Des opérateurs booléens : `&&` (et), `||` (ou), `!` (non)

Attention ! Bien distinguer `=` et `==`

Exemple

```
import java.util.Scanner;
public class Exemple {
    public static void main(String[] args) {
        Scanner clavier = new Scanner(System.in);
        int nombre1;

        nombre1 = clavier.nextInt();
        if ((nombre1 % 2) == 0) {
            System.out.println ("Le nombre est pair");
        } else {
            System.out.println ("Le nombre est impair");
        }
    }
}
```

Exemple

```
import java.util.Scanner;
public class Exemple {
    public static void main(String[] args) {
        Scanner clavier = new Scanner(System.in);
        int âge;

        âge = clavier.nextInt();
        if ( âge<21 || âge>=60 ) {
            System.out.println (" Tarif réduit ");
        }
    }
}
```


Le « selon-que »

Première forme

```
switch(numéroJour) {  
    case 1 : intituléJour ="Lundi"; break;  
    case 2 : intituléJour ="Mardi"; break;  
    case 3 : intituléJour ="Mercredi"; break;  
    case 4 : intituléJour ="Jeudi"; break;  
    case 5 : intituléJour ="Vendredi"; break;  
    case 6 : intituléJour ="Samedi"; break;  
    case 7 : intituléJour ="Dimanche"; break;  
    default : intituléJour ="Inconnu"; break;  
}
```

- Notez le **break**
- Possible avec : entiers, caractères et chaînes

Le « selon-que »

Deuxième forme : la logique suivante

```
Selon que  
  nb > 0 : Ecrire " positif "  
  nb = 0 : Ecrire "nul"  
  autrement : Ecrire "négatif"  
Fin selon que
```

s'écrit en Java

```
if (nb>0) {  
    System.out.println (" positif ");  
} else if (nb==0) {  
    System.out.println ("nul");  
} else {  
    System.out.println (" négatif");  
}
```

Leçon 8

La notion de grammaire

- La notion de grammaire
- Comment fonctionne une grammaire ?
- Notations de la grammaire Java

Le besoin d'une référence

Un langage de programmation doit pouvoir

- ▶ **être décrit** auprès des programmeurs
- ▶ faire l'objet d'une **compilation rigoureuse**

Le besoin d'une référence

Un langage de programmation doit pouvoir

- ▶ **être décrit** auprès des programmeurs
- ▶ faire l'objet d'une **compilation rigoureuse**

Il faut un **document de référence**

- ▶ Pour Java : **The Java Language Specification**
 - Contient beaucoup de texte (en **Anglais**)
 - Ce qui est parfois **incomplet** / **ambigu**

Le besoin d'une référence

Un langage de programmation doit pouvoir

- ▶ **être décrit** auprès des programmeurs
- ▶ faire l'objet d'une **compilation rigoureuse**

Il faut un **document de référence**

- ▶ Pour Java : **The Java Language Specification**
 - Contient beaucoup de texte (en **Anglais**)
 - Ce qui est parfois **incomplet** / **ambigu**

Nécessité d'utiliser un **formalisme** précis

- ▶ En Logique : LDA
- ▶ En Analyse : UML
- ▶ En Langage : une **grammaire**

La notion de grammaire

Une grammaire est une description **finie** de l'**infinité** des programmes corrects

programme \equiv phrase \supset mots \supset caractères

- ▶ Chaque **mot** (on dit **token** en informatique) doit être légal
 - déterminé par une **grammaire lexicale**
- ▶ **Séquence de mots** doit être légale
 - déterminé par une **grammaire syntaxique**
- ▶ La **sémantique** ne peut pas être décrite aussi rigoureusement

La notion de grammaire

Parallèle avec une **langue naturelle**
(dictionnaire, grammaire)

Exemples

- ▶ «*El tahc tse rion*»
- ▶ «*Le est chat noir*»
- ▶ «*Le chat est noir*»
- ▶ «*Le parapluie mange l'ascenseur*»

Comment fonctionne une grammaire ?

Une grammaire est composée de

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

Est **valide** ce qui peut être **produit** par la grammaire

Comment fonctionne une grammaire ?

Pour la grammaire **syntactique**

- ▶ les symboles terminaux sont les mots
- ▶ les règles construisent la phrase

Pour la grammaire **lexicale**

- ▶ les symboles terminaux sont les caractères
- ▶ les règles construisent le mot

Pourquoi ne pas utiliser un dictionnaire pour les mots ?

Exemple : le langage MU

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

Exemple : le langage MU

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

start:

debut MU fin

debut:

debut I

I

fin:

I fin

I

- La règle de départ : start

Exemple : le langage MU

Quelques **phrases correctes** (qu'on peut *produire*)

- ▶ IMUI
- ▶ IIMUI
- ▶ IMUIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
- ▶ ...à l'infini

Exemple : le langage MU

Quelques **phrases correctes** (qu'on peut *produire*)

- [illegible]

Quelques **phrases incorrectes** (impossibles à *produire*)

- ▶ MU
- ▶ MUI
- ▶ MUIMU
- ▶ ...à l'infini

Exemple

Voici un autre exemple

- ▶ Le symbole terminal : A
- ▶ Les 2 règles de production

liste:

element

element liste

element:

A

- ▶ La règle de départ : liste

Quel est le langage produit par cette grammaire?

Notations de la grammaire Java

Une règle de la grammaire syntaxique java :

ReturnStatement :

return Expression_(opt) ;

- Imaginez des exemples d'instruction « return » valides

La **grammaire complète** de Java peut être consultée dans le livre « *The Java Language Specification* »

Leçon 9

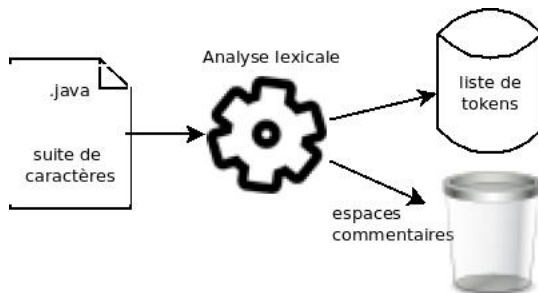
La grammaire lexicale de Java

- Analyse lexicale
- Les espaces
- Les commentaires
- Les tokens

Analyse lexicale de Java

Première phase d'analyse d'un programme

- ▶ Examine la séquence des caractères d'entrée
- ▶ Supprime caractères d'espacement et commentaires
- ▶ Identifie les *tokens* (mots) du langage



Analyse lexicale de Java

L'analyse lexicale se base sur une grammaire lexicale

- ▶ Les symboles terminaux sont l'ensemble des caractères Unicode

Java a fait le choix de l'UTF-16 (Unicode)

- ▶ ASCII : 7 bits
- ▶ ASCII étendu : 8 bits
- ▶ EBCDIC : 8 bits (IBM)
- ▶ UTF-16 : 2 bytes, 16 bits
(les 128 premiers caractères \equiv ASCII)

Les séquences Unicode

Java permet une représentation ASCII de tout caractère UTF-16 (via un *caractère d'échappement*)

- ▶ **Ex** : le caractère **p** peut aussi s'écrire `\u0070`
- ▶ Une traduction préalable est réalisée :

```
\u0070ublic class MaClasse
```

devient

```
public class MaClasse
```

Les caractères d'entrée

UnicodeInputCharacter :

UnicodeEscape

RawInputCharacter

UnicodeEscape :

\UnicodeMarker HexDigit HexDigit HexDigit HexDigit

UnicodeMarker :

u

UnicodeMarker u

RawInputCharacter :

any Unicode character

HexDigit : one of

0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

Les caractères d'espacement

Les «espaces» n'ont pas de sens en Java

- ▶ Peuvent être utilisés librement entre les mots
- ▶ Mais ne peuvent pas couper un mot
- ▶ Exception : 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

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

Le commentaire

- Sur 1 ligne

```
// Commentaire sur 1 ligne
```

- Sur plusieurs lignes

```
/* Exemple de commentaire  
sur plusieurs lignes */
```

```
/** Pour un commentaire Javadoc */
```

- Ne peuvent pas être imbriqués
- N'importe où **entre** les mots

Les tokens (mots du langage)

Les caractères qui restent vont former les *tokens* (mots, symboles terminaux)

- 5 sortes de tokens

Token : one of
Identifier Literal Keyword Separator Operator

Les tokens (mots du langage)

Liste des *keyword* (**mots-clés** ou réservés)

Keyword : one of

```
abstract boolean break byte case catch  
char class const continue default do  
double else extends final finally float  
for goto 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 identifiants

Identifier :

IdentifierChars

but not a *Keyword* or *BooleanLiteral* or *NullLiteral*

IdentifierChars :

JavaLetter

IdentifierChars JavaLetterOrDigit

JavaLetter :

any Unicode character that is a Java letter (_ et \$ sont compris)

JavaLetterOrDigit :

any Unicode character that is a Java letter-or-digit

(cf. la grammaire complète pour les détails)

Les tokens (mots du langage)

Liste des séparateurs et opérateurs

Separator : one of

() { } [] ; , .

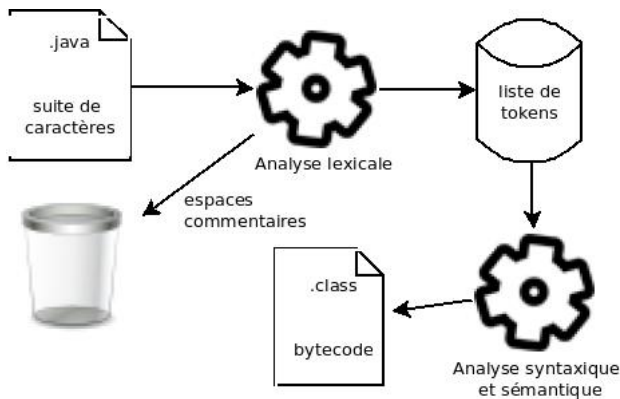
Operator : one of

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

Comment l'analyseur lexical va-t-il reconnaître ——— ?

Récapitulatif des phases de compilation

Une fois les tokens identifiés, le compilateur peut passer à l'analyse syntaxique et sémantique



Leçon 10 — Écrire du code lisible

Nous montrons pourquoi et comment écrire du code lisible

Et pourtant ça tourne !

On a vu que le compilateur ne se préoccupe pas de la *mise en page*

- Tous les *whitespace* sont éliminés lors de l'analyse lexicale (*whitespace* \equiv *espace*, *retour à la ligne*, ...)

\implies ceci est **équivalent** au *Hello World*

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

Et pourtant ça tourne !

C'est correct pour le compilateur mais à **proscrire**

- ▶ Un code est **souvent lu**
 - Lorsqu'il est écrit / mis au point
 - Correction de bug
 - Évolution du code (les besoins changent)
- ▶ Et souvent par des **personnes différentes**

⇒ **La lisibilité est essentielle**

Un code lisible

Règle 1 : **Indenter** correctement son code

- ▶ Pour un aperçu global de la structure du code
- ▶ Pour repérer rapidement la fin d'un *bloc*

Règle 2 : Bien choisir le **nom des variables**

- ▶ Ce bout de code est syntaxiquement correct
- ▶ Mais que fait-il ?

```
int u=clavier.nextInt(),n=clavier.nextInt(),  
t=clavier.nextInt();  
double p=u*n*(1+t/100.0);  
System.out.println(p);
```


Un code lisible

Nous lui préférons celui-ci plus lisible :

```
double àPayer;  
int prixUnitaire = clavier.nextInt();  
int nombreArticles = clavier.nextInt();  
int tauxTva = clavier.nextInt();  
  
àPayer = prixUnitaire * nombreArticles * (1 + tauxTva/100.0);  
  
System.out.println (àPayer);
```

Un code lisible

Règle 3 : Décomposer les expressions trop longues

- ▶ Une variable intermédiaire peut accroître la lisibilité : le nom donne un sens à l'expression
- ▶ La perte de place mémoire est négligeable (voire nulle car un compilateur peut optimiser)
- ▶ Trop décomposer nuit parfois à la lisibilité (limite floue)
⇒ importance de l'**expérience**

Un code lisible

Exemple : reprenons l'exemple du calcul du prix à payer

```
int prixUnitaireHTva = clavier.nextInt ();  
int nombreArticles = clavier.nextInt ();  
int tauxTva = clavier.nextInt ();  
int prixUnitaireTTC = prixUnitaireHTva * (1 + tauxTva/100.0);  
double àPayer = prixUnitaireTTC * nombreArticles;
```

Exemple : que préférer de

```
int hypo = sqrt( a*a + b*b );
```

```
int aCarré = a*a;  
int bCarré = b*b;  
int hypo = sqrt( aCarré + bCarré );
```

Un code lisible

Règle 4 : Utiliser des **constantes**

- ▶ Rend le code plus lisible
- ▶ Facilite aussi son évolution

```
final double TAUX_TVA = 0.21;  
double taxe;  
double prix = clavier.nextInt();  
taxe = prix * TAUX_TVA;  
System.out.println (taxe);
```

Un code lisible

Anti-règle : Surcharger de commentaires

- ▶ Vient souvent au secours d'un code non lisible
- ▶ Idéalement on utilisera les commentaires
 - En début de programme, de module
 - Pour expliquer **ce qu'il fait**
 - Mais surtout **pas comment** il le fait
 - cf. *javadoc*

Il y a d'autres règles ? Oui !

- ▶ Pour Java, document reprenant les conventions à respecter : <http://www.oracle.com/technetwork/java/codeconv-138413.html>

La refactorisation

En réalité : On n'écrit pas un code lisible du premier coup !

Refactoriser = changer le code en vue d'améliorer sa lisibilité / son évolutivité sans changer ce qu'il fait

- ▶ **Exemples**

- donner un nom plus explicite à une variable
- mieux indenter le code

- ▶ **Contre-exemples**

- ajouter une fonctionnalité
- réparer un bug

La refactorisation

Mais refactoriser c'est toucher à du code qui fonctionne !
Ce n'est pas dangereux ?

Oui ! \implies importance

- ▶ des tests de **non-régression** (cf. JUnit)
- ▶ d'un système de **gestion des versions**

Pour aller plus loin : Martin Fowler,
« *Refactoring : Improving the design of existing code* »

Leçon 11 — Code modulaire (survol)

Nous voyons comment traduire les modules de votre cours de Logique

- Découper le code
- Appel
- Définition
- Paramètres
- Conclusion

Découper du code

Pourquoi ?

- ▶ Pour le réutiliser
- ▶ Pour scinder la difficulté
- ▶ Pour faciliter le déverminage
- ▶ Pour accroître la lisibilité
- ▶ Pour diviser le travail

Découper du code

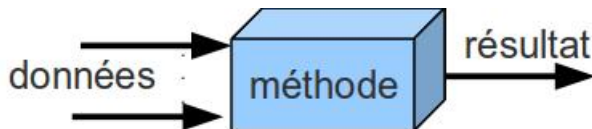
Comment ?

- ▶ \exists un **nom** qui décrit tout ce qu'il fait
- ▶ Il résout un **sous-problème** bien précis
- ▶ Il est **fortement documenté**
- ▶ Il est le plus **général possible**
- ▶ Il tient sur une page

En Java on dit **méthode** et pas **module**

Appel d'une méthode

Une méthode est une **boîte noire**



Pour l'utiliser, on doit savoir :

- ▶ Son nom
- ▶ Quoi lui donner
- ▶ Ce qu'elle retourne
- ▶ Mais **pas comment** elle fait

Appel d'une méthode

À partir du code d'une **autre classe**

- ▶ `nomClasse.nomMéthode(...)`
- ▶ **Exemples :**

```
double racine = Math.sqrt(4.0);  
double aléatoire = Math.random();  
int nb = -10;  
int absolu = Math.abs(nb);
```

Si on est dans la même classe

- ▶ On indique directement le nom de la méthode
(*des exemples suivront*)

Définition d'une méthode

```
public static typeRetour nomMéthode ( <<paramètre(s) éventuel(s)>> ) {  
    // code de la méthode  
    return <<résultat>>; // de type : typeRetour  
}
```

Exemple : la moyenne de 2 réels

```
public static double moyenne ( double nb1, double nb2 ) {  
    double moyenne = (nb1 + nb2) / 2.0;  
    return moyenne;  
}
```

- Appel possible (si dans la même classe)

```
double cote = moyenne(12.5, 17.5);
```

Définition d'une méthode

Exemple : la valeur absolue

```
public static int absolu ( int nb ) {  
    int abs = nb;  
    if (nb<0) {  
        abs = -nb;  
    }  
    return abs;  
}
```

► Exemples d'appels

```
int résultat = absolu(4);  
int écart = -10;  
int écartAbsolu = absolu(écart);
```

Définition d'une méthode

Si pas de valeur de retour :

- ▶ On indique **void**
- ▶ Pas de **return**

Exemple :

```
public static void présenter (String nomPgm) {  
    System.out.println ("Programme_ "+nomPgm);  
}
```

- ▶ Exemple d'appel

```
présenter ("moyenne_de_2_nombres");
```

Définition d'une méthode

On trouve aussi des méthodes sans paramètre

Exemple :

```
public static int lireEntier () {  
    Scanner clavier = new Scanner(System.in);  
    int nb;  
    System.out.println ("Entrez_un_nombre_entier!");  
    nb = clavier.nextInt ();  
    return nb;  
}
```

► Exemple d'appel

```
int nb = lireEntier ();
```


Commentaire d'une méthode

Il est essentiel de commenter chaque méthode

- ▶ Pour savoir pourquoi et comment l'utiliser
- ▶ Pour la comprendre et ainsi pouvoir la corriger et/ou la modifier

Exemple : la valeur absolue

```
/**  
 * Calcul de la valeur absolue.  
 * @param nb le nombre dont on veut la valeur absolue.  
 * @return la valeur absolue de <code>nb</code>  
 */  
public static int absolu ( int nb ) {  
    ...  
}
```

Commentaire d'une méthode

La documentation suit une notation précise

- ▶ Permet une production automatique de documents d'aide
- ▶ En respectant un style uniforme pour s'y retrouver facilement

absolu

```
public static int absolu(int nb)
```

Calcul de la valeur absolue.

Parameters:

`nb` - le nombre dont on veut la valeur absolue.

Returns:

la valeur absolue de `nb`

Plus de détails dans la leçon dédiée à la documentation du code

Un exemple complet

```
package be.heb.esi.lg1.cours;
import java.util.Scanner;

public class MaxEntiers {

    /**
     * Donne le maximum de 2 nombres.
     * @param nb1 le premier nombre.
     * @param nb2 le deuxième nombre.
     * @return la valeur la plus grande entre 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

```
/**
 * Lit un nombre entier.
 * Le nombre est lu sur l'entrée standard (le clavier).
 * @return le nombre entier lu.
 */
public static int lireEntier () {
    Scanner clavier = new Scanner(System.in);
    System.out.println("Entrez un nombre entier!");
    return clavier.nextInt();
}

/**
 * Affiche le maximum de 2 nombres entrés au clavier.
 * @param args pas utilisé.
 */
public static void main ( String [] args ) {
    int max; // Le max des nombres lus
    int nb1, nb2; // Chacun des nombres lus
    nb1 = lireEntier ();
    nb2 = lireEntier ();
    max = max(nb1,nb2);
    System.out.println("max=" + max);
}
```

Passage de paramètres

En logique 3 passages de paramètres :

- ▶ en entrée, en sortie, en entrée-sortie

En Java, uniquement **par valeur**

- ▶ = la valeur est copiée dans le paramètre
- ▶ \simeq paramètre en entrée

Conclusion

Insistons : Une **méthode**

- ▶ **fait une et une seule chose**
- ▶ possède un **nom explicite**
- ▶ est **fortement documentée**

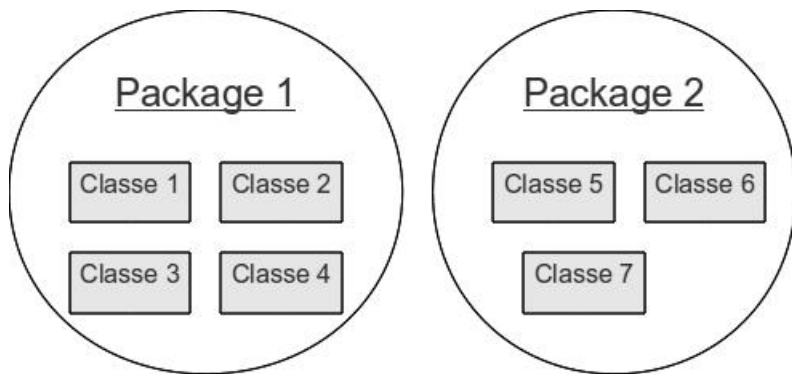
Leçon 12 — Organiser le code

Dans une application réelle, la taille des programmes impose une organisation rigoureuse

Le groupement en package

L'**API** (Application Programming Interface) désigne la bibliothèque standard Java

- ▶ Elle contient des **milliers** de classes
- ▶ Elles sont regroupées en **package**



La notion de package

Un **package**

- ▶ Regroupe les classes liées
- ▶ Permet l'unicité des noms de classe
 - nom complet / qualifié : `monPackage.MaClasse`

Nom d'un package

- ▶ identifiants séparés par des points .
- ▶ tout en minuscules
- ▶ adresse internet inversée (unicité)
- ▶ ex : `be.heb.esi.java1`, `org.apache.struts.action`

Utilisation

Pour utiliser une classe

- mettre le nom **qualifié** (complet)

```
java.util.Calendar now = java.util.Calendar.getInstance();
```

- ou utiliser **import** qui crée un raccourci

```
import java.util.Calendar;  
public Test {  
    ...  
    Calendar now = Calendar.getInstance();  
    ...  
}
```

- Cas particulier : le package **java.lang** est importé implicitement
 - Exemple : on peut tout de suite écrire

```
double racine = Math.sqrt(1.21);
```

Utilisation

Comment savoir comment utiliser les classes et méthodes ? En lisant la **javadoc**

The screenshot shows the Java Platform Standard Ed. 7 Javadoc page for the `java.util.Calendar` class. The left sidebar lists packages and classes, with `Calendar` selected. The main content area displays the class name, implemented interfaces (`Serializable`, `Cloneable`, `Comparable<Calendar>`), and direct known subclasses (`GregorianCalendar`). The class is defined as a public abstract class that provides methods for converting between a specific instant in time and a set of calendar fields. It also provides additional fields and methods for implementing a concrete calendar system. The page includes a section for getting and setting calendar field values.

(<http://download.oracle.com/javase/7/docs/api/>)

Utilisation

On peut y lire le nom du package

java.util

Class Calendar

et la description de la méthode

getInstance

```
public static Calendar getInstance()
```

Gets a calendar using the default time zone and locale. The `Calendar` returned is based on the current time in the default time zone with the default locale.

Returns:

a `Calendar`.

On verra comment produire une *javadoc* similaire pour son code

Créer ses packages

Pour placer une classe dans un package, la commande est **package** nom_package;

- ▶ Doit être la **première instruction** du fichier
- ▶ Exemple :

```
package be.heb.esi.java1;  
public class Test {  
    // Nom complet : be.heb.esi.java1.Test  
}
```

Créer ses packages

Qu'est-ce qui va changer en pratique ?

- ▶ La compilation ne change pas :

```
javac NomClasse.java
```

- ▶ L'exécution change :

```
java nomPackage.NomClasse
```

- ▶ Contraintes sur l'endroit où placer le **bytecode**
 - Lié à la notion de **CLASSPATH**
 - Sera détaillé au laboratoire

Réutiliser du code

Si on doit coder quelque chose, c'est **peut-être déjà fait**

- ▶ \implies autant le réutiliser
 - Gain de temps
 - Probablement mieux écrit
- ▶ Importance de **connaître l'API**
(en tout cas les classes principales)

Leçon 13 — Les boucles (survol)

Nous voyons comment traduire les algorithmes contenant des boucles que vous écrivez au cours de Logique

Instructions répétitives

Le **Tant que** :

```
while ( condition ) {  
    instructions  
}
```

Exemple :

```
int puissance = 1;  
while ( puissance < 1000 ) {  
    System.out.println ( puissance );  
    puissance = 2 * puissance;  
}
```

Exemple

```

import java.util.Scanner;
public class Exemple {
    /**
     * Affiche la somme d'entiers positifs entrés au clavier .
     * S'arrête dès qu'une valeur nulle ou négative est donnée.
     * @param args non utilisé
     */
    public static void main(String[] args) {
        Scanner clavier = new Scanner(System.in);
        int nb;
        int somme = 0;
        nb = clavier.nextInt();
        while ( nb > 0 ) {
            somme = somme + nb;
            nb = clavier.nextInt();
        }
        System.out.println (somme);
    }
}

```

Instructions répétitives

Le **Pour** :

```
for ( int i=début; i<=fin; i=i+pas ) {  
    instructions  
}
```

Exemple :

```
for ( int i=1; i<=10; i=i+1 ) {  
    System.out.println ( i );  
}
```

Exemple

```
public class Exemple {  
    /**  
     * Affiche la somme des nombres pairs entre 2 et 100.  
     * @param args non utilisé  
     */  
    public static void main(String[] args) {  
        int somme;  
  
        somme = 0;  
        for ( int i=2; i<=100; i=i+2 ) {  
            somme = somme + i;  
        }  
        System.out.println (somme);  
    }  
}
```

Exemple

```
public class Exemple {  
    /**  
     * Affiche un compte à rebours à partir de 10.  
     * @param args non utilisé  
     */  
    public static void main(String[] args) {  
        for ( int i=10; i>=1; i=i-1 ) {  
            System.out. println ( i );  
        }  
        System.out. println ( "Partez !" );  
    }  
}
```

Instructions répétitives

`i++` est un raccourci pour `i=i+1`

Exemple :

```
for ( int i=1; i<=n; i++ ) {  
    System.out.println ( i );  
}
```

Étude de cas

Objectif : lecture d'une donnée entière positive

► **Étape 1** : lire un entier

```
/**
 * Lit un entier au clavier .
 * Les valeurs non entières sont passées.
 * @return l'entier lu .
 */
public static int lireEntier () {
    Scanner clavier = new Scanner(System.in);
    int nb;
    // Tant que ce n'est pas un entier au clavier
    while ( ! clavier . hasNextInt() ) {
        clavier . next (); // le lire , le passer
    }
    nb = clavier . nextInt ();
    return nb;
}
```

Étude de cas

► Étape 2 : lire un entier positif

```
/**
 * Lit un entier au clavier .
 * Les valeurs non entières , nulles ou négatives sont passées .
 * @return l'entier lu .
 */
public static int lirePositif () {
    int nb;
    nb = lireEntier ();
    while (nb<=0) {
        nb = lireEntier ();
    }
    return nb;
}
```


Leçon 14

Écrire du code robuste

- Motivation
- Gérer les erreurs
- Confiner les problèmes

Motivation

Un programme ne tourne pas dans un monde idéal

Il doit pouvoir **résister aux défaillances** de l'environnement

- ▶ On tente d'ouvrir un fichier qui n'existe pas
- ▶ L'utilisateur entre des données incorrectes
- ▶ ...

La gestion des erreurs

Exemple :

```
import java.util.Scanner;
public class Affiche {
    /**
     * Affiche un nombre entier lu au clavier .
     * @param args non utilisé
     */
    public static void main(String[] args) {
        Scanner clavier = new Scanner(System.in);
        int nb;

        nb = clavier.nextInt();
        System.out.println(nb);
    }
}
```

- ▶ À priori tout va bien !

La gestion des erreurs

Et si l'utilisateur entre une lettre ?

```
> javac Affiche.java
> java Affiche
a
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:857)
    at java.util.Scanner.next(Scanner.java:1478)
    at java.util.Scanner.nextInt(Scanner.java:2108)
    at java.util.Scanner.nextInt(Scanner.java:2067)
    at Affiche.main(Affiche.java:7)
```

Nom de l'exception

Pile d'appels (par où il est passé)

- ▶ Une **exception** est générée
- ▶ Le programme s'**arrête brutalement** et affiche un message d'erreur

La gestion des erreurs

2 inconvénients majeurs

- ▶ Le message d'erreur est très utile pour le développeur mais pas pour l'utilisateur
- ▶ L'arrêt du programme est rarement le comportement souhaité

On aimerait pouvoir **gérer** le problème

- ▶ Au mieux, le régler
- ▶ Au pire, afficher un message plus clair pour l'utilisateur

La gestion des erreurs

Possible grâce à l'instruction **try catch**

- ▶ **try** : contient les instructions qui **peuvent mal se passer**
- ▶ **catch** : contient le code qui est **en charge** de gérer le problème

Quand un problème se présente dans le **try**

- ▶ Le code du **try** est interrompu
- ▶ Le code du **catch** est exécuté
- ▶ Ensuite, on continue après le **try-catch**

La gestion des erreurs

Exemple : on affiche un message plus clair

```
package be.heb.esi.lgjl ;
import java.util.Scanner;
public class Affiche {
    /**
     * Affiche l'entier lu au clavier ou un message si ce n'est pas un entier.
     */
    public static void main(String[] args) {
        Scanner clavier = new Scanner(System.in);
        int nb;
        try {
            nb = clavier.nextInt();
            System.out.println(nb);
        }
        catch(Exception e) {
            System.out.println("Ce n'est pas un entier!");
        }
    }
}
```

```
> java be.heb.esi.lgjl.Affiche
deux
Ce n'est pas un entier!
```

Confiner les problèmes - Motivation

Imaginons la situation suivante :

- ▶ Un programme demande un entier à l'utilisateur
- ▶ Il doit être positif
- ▶ L'utilisateur entre un nombre négatif
- ▶ Le programme ne le vérifie pas tout de suite

Confiner les problèmes - Motivation

On aura un problème :

- ▶ Un plantage
- ▶ Un résultat erroné
- ▶ Un effet indésiré (perte de données, ...)

Mais le problème va survenir :

- ▶ Plus tard dans le temps
- ▶ Plus loin dans le code

⇒ **Difficile** à comprendre et **corriger**

Confiner les problèmes

Besoin de **confiner** les problèmes

- ▶ Un problème est **détecté rapidement** avant qu'il ne se propage dans le reste du code

Cas pratique : vérifier les **paramètres**

- ▶ Si une contrainte est associée à un paramètre
 - Le vérifier en début de méthode
 - Que faire si pas valide ?

Confiner les problèmes

Nous avons appris à attraper une exception.
On peut aussi en créer une

```
/**
 * Calcule la racine carrée d'un nombre.
 * @param nb le nombre dont on veut la racine carrée.
 * @return la racine carrée de <code>nb</code>.
 * @throws IllegalArgumentException si <code>nb</code> est négatif.
 */
public static double racineCarrée(double nb) {
    if (nb < 0) {
        throw new IllegalArgumentException("nb doit être positif !");
    }
    // Traitement normal. On est sûr que le paramètre est OK.
}
```

- ▶ On dit qu'on **lance** une exception (ici de type `IllegalArgumentException`)
- ▶ Pourra être attrapée via un **catch**

Confiner les problèmes

Exemple

```
try {  
    System.out.println ( racineCarrée( val ) );  
} catch (Exception ex) {  
    System.out.println ( "Calcul impossible !" );  
}
```

On peut aussi préciser qu'on n'attrape **que** les `IllegalArgumentException`

```
try {  
    System.out.println ( racineCarrée( val ) );  
} catch (IllegalArgumentException ex) {  
    System.out.println ( "Calcul impossible !" );  
}
```

Leçon 15

Les types et les littéraux

- Un langage typé
- Les types entiers
- Les types flottants
- Les booléens
- La chaîne de caractères

Les types

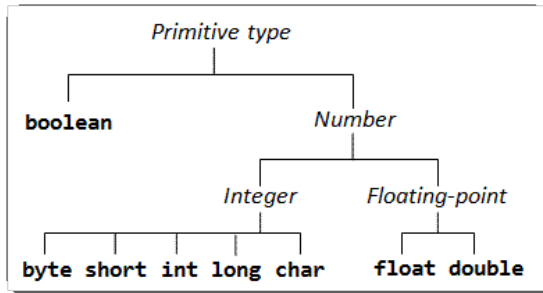
Toute donnée a un type

- ▶ Cohérence sémantique
- ▶ Allocation mémoire adaptée

Quels types ?

- ▶ **primitifs prédéfinis** :
 - entier, réel, booléen (logique)
- ▶ **références prédéfinis** :
 - tableaux, String, ...
- ▶ **références définis par le programmeur**

Les types primitifs



source : http://www3.ntu.edu.sg/home/ehchua/programming/java/J2_Basics.html

PrimitiveType : one of
NumericType boolean

NumericType : one of
IntegerType *FloatingPointType*

IntegralType : one of
byte short int long char

FloatingPointType : one of
float double

Les types numériques entiers

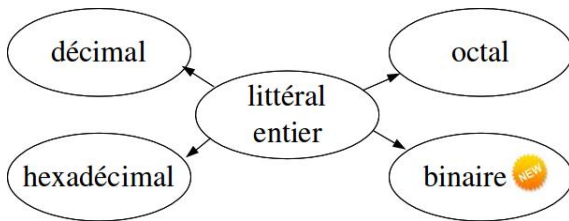
byte, **short**, **int** et **long** (**char** sera vu à part) :

- ▶ Nombres signés (en complément à 2)
- ▶ Codés sur 8-bit, 16-bit, 32-bit et 64-bit
- ▶ Comprennent donc les valeurs
 - -128 à 127
 - -32768 à 32767
 - -2147483648 à 2147483647
 - -9223372036854775808 à 9223372036854775807
- ▶ **Modélisation** de la notion mathématique d'entier
 - Capacité limitée
 - permet de représenter un intervalle fini de \mathbb{Z}
 - *out of range* possible

Les littéraux entiers

Littéral : représentation d'une valeur

Pour les entiers, différents formats sont possibles



IntegerLiteral : one of

DecimalIntegerLiteral

HexIntegerLiteral

OctalIntegerLiteral

BinaryIntegerLiteral


Les littéraux entiers

Un **DecimalNumeral** (numérique décimal)

DecimalIntegerLiteral :

*DecimalNumeral IntegerTypeSuffix*_(opt)

IntegerTypeSuffix : one of 1 L

- ▶ *DecimalNumeral* : suite de chiffres
- ▶ `_` permis pour la lisibilité 
- ▶ Le suffixe (`1` ou `L`) : distingue un **int** d'un **long**
- ▶ Pas de **byte** ou **short** ? Écrire un **int**
- ▶ Pas de **littéral négatif** ? Utiliser le `-`
- ▶ Exemples corrects : `0` `1275` `1_421` `23L`
- ▶ Exemples incorrects : `12.3` `1 000` `1,000` `1.0`

Les littéraux entiers

Octal

- ▶ précédé de 0
- ▶ chiffres de 0 à 7

Héxadécimal

- ▶ précédé de 0x ou 0X
- ▶ chiffres + a,b,c,d,e,f (minuscules/majuscules)

Binaire

- ▶ précédé de 0b ou 0B
- ▶ chiffres 0 et 1

Les littéraux entiers

Exemple : Quelques littéraux corrects pour la valeur 100 de type **int**

- ▶ 100
- ▶ 1_0_0
- ▶ 0144
- ▶ 01_44
- ▶ 0x64
- ▶ 0b110_0100

Le type numérique caractère

char

- ▶ Caractère Unicode
- ▶ Entier non signé sur 16 bits
- ▶ Assimilé à un entier (on peut faire des calculs!)
- ▶ Plusieurs notations

CharacterLiteral :

- ’ *SingleCharacter* ’
 - ’ *EscapeSequence* ’
-

- ▶ La plus simple : le **caractère entre *single quote***

SingleCharacter :

any character but not ’ or \ or *LineTerminator*

Les littéraux caractères

Notations spéciales avec les séquences d'échappement (*Escape Sequence*)

- Pour les caractères non représentables simplement

<code>\n</code>	line feed	<code>\t</code>	tabulation	<code>\'</code>	'
<code>\r</code>	carriage return	<code>\b</code>	backspace	<code>\"</code>	"
				<code>\\</code>	\

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

- Pour utiliser le code Unicode

- Exemples :

- `'\u0F40'` pour le KA tibétain
- `'\u17E0'` pour le chiffre 0 Khmer

Les types à virgule flottante

float, double

- ▶ Respectent la norme IEEE754
- ▶ Codés sur (respectivement) 32-bit, et 64-bit
- ▶ On utilisera plus souvent le type **double**
- ▶ **Modélisation** de la notion mathématique
 - Capacité limitée (*out of range* possible)
 - Précision limitée
 - → **imprécision** 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

Notation assez souple

FloatingPointLiteral :

Digits . *Digits*_(opt) *ExponentPart*_(opt) *FloatTypeSuffix*_(opt)
.*Digits* *ExponentPart*_(opt) *FloatTypeSuffix*_(opt)
Digits *ExponentPart* *FloatTypeSuffix*_(opt)
Digits *ExponentPart*_(opt) *FloatTypeSuffix*

ExponentPart :

e *signedInteger*
E *signedInteger*

FloatTypeSuffix : one of

f F d D

Les littéraux à virgule flottante

partie entière	.	partie décimale	E	exposant	suffixe
----------------	---	-----------------	---	----------	---------

- ▶ 4 parties optionnelles (mais pas ensemble)
 - Cela doit rester sensé
 - On ne peut pas le confondre avec un entier
- ▶ En l'absence de suffixe : un double
- ▶ Exemples : `1.2E3`, `1.F`, `.1`, `1e-2d`, `1f`
- ▶ Contre-exemples : `1`, `.E1`, `E1`

Le type booléen

boolean

- ▶ Appelé aussi **logique**
- ▶ 2 valeurs : **true** (vrai) et **false** (faux)

La chaîne de caractères

String

StringLiteral :

" *StringCharacters*_(opt) "

StringCharacters :

StringCharacter

StringCharacters *StringCharacter*

StringCharacter :

InputCharacter but not " or \

EscapeSequence

InputCharacter :

UnicodeInputCharacter but not CR or LF

► Exemples de **String** :

■ "Bonjour_"

■ "'Un_peu_de_tout'_:_\\"n\\\""

► Attention : **char** ≠ **String**

Leçon 16

Les tableaux (survol)

- Présentation
- Type / Déclaration / Création
- Accès
- Taille
- Tableau et méthode
- Plusieurs dimensions
- Erreurs fréquentes

Avertissement

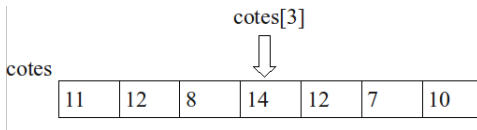
Nous présentons ici une **vue** très **simplifiée**
des tableaux en Java afin de **coller**
à votre cours de **logique**.

Nous aurons l'occasion d'être plus précis
lors d'une prochaine leçon.

Présentation

Nécessité de manipuler **plusieurs variables similaires**

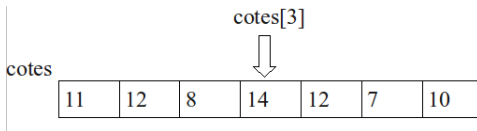
- ▶ **Ex** : plusieurs cotes, plusieurs températures
- ▶ Accès à un des éléments via un **indice** (*position*)



Présentation

Nécessité de manipuler **plusieurs variables similaires**

- ▶ **Ex** : plusieurs cotes, plusieurs températures
- ▶ Accès à un des éléments via un **indice** (*position*)



Pourquoi pas plusieurs variables ?

- ▶ Écriture **compacte** et qui s'**adapte** à la taille
- ▶ **Ex** : En logique, si `tab` est un tableau de N entiers

```
pour i de 1 à N faire  
    écrire tab[i]  
fin pour
```

Type et déclaration

Éléments de type $T \Rightarrow$ type du tableau = $T[]$

Exemples

- ▶ **int** [] est le type *tableau d'entiers*
- ▶ **String** [] est le type *tableau de chaines de caractères*

Type et déclaration

Éléments de type $T \Rightarrow$ type du tableau = $T[]$

Exemples

- ▶ `int []` est le type *tableau d'entiers*
- ▶ `String []` est le type *tableau de chaines de caractères*

En Java : uniquement des tableaux **dynamiques**

- ▶ La taille ne fait pas partie du type
- ▶ Déclaration et création sont séparées

Type et déclaration

Éléments de type $T \Rightarrow$ type du tableau = $T[]$

Exemples

- ▶ **int** [] est le type *tableau d'entiers*
- ▶ **String** [] est le type *tableau de chaines de caractères*

En Java : uniquement des tableaux **dynamiques**

- ▶ La taille ne fait pas partie du type
- ▶ Déclaration et création sont séparées

La déclaration suit la syntaxe habituelle

Exemples

```
int [] cotes;  
String [] noms;
```

Création

Il faut encore **créer** le tableau (via le mot clé **new**)

Exemple :

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

t

--	--	--

 (espace pour 3 entiers)

Création

Il faut encore **créer** le tableau (via le mot clé **new**)

Exemple :

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

t

On peut aussi combiner déclaration/création

Exemple :

► **int[] t = new int[3];**

Création

On peut aussi créer le tableau en **donnant ses valeurs**

- ▶ Dans ce cas, on ne spécifie pas la taille (elle est déduite)
- ▶ **Exemple :**

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

entiers

4	5	6
---	---	---

Accès aux éléments

Définition : taille = Nombre d'éléments

En Java on ne choisit pas l'indice de départ : toujours 0

- ▶ —> Indices de **0 à taille du tableau - 1**
- ▶ Exemple :

```
int[] entiers = {7,14,0};  
int entier = entiers[2]; // entier vaut 0  
entiers[1] = 85;  
entier = entiers[entier+1]; //entier vaut 85
```

Accès aux éléments

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 < 10; i++) {
            tableau[i] = i;
        }
    }
}
```

Taille

En Java : tout tableau connaît sa **taille**

- ▶ Via `nomTableau.length`
- ▶ **Exemple :**

```
int[] entiers = {4,5,6};  
int taille = entiers.length;  
System.out.println ( taille ); // écrit 3
```

- ▶ Permet d'écrire un code qui s'adapte mieux aux changements

Taille

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]);
        }
    }
}
```

Taille

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]);
        }
    }
}
```

Tableau et méthode

Un tableau peut être un paramètre d'une méthode.

Exemple : Afficher un tableau

```
public static void afficher ( int[] tab ) {  
    for(int i = 0; i<tab.length; i++) {  
        System.out. println (tab[i]);  
    }  
}
```

► L'appel pourrait être

```
int[] cotes = {12,8,10,14,9};  
afficher ( cotes );
```

Tableau et méthode

En Java : passage de paramètres par **valeur**

- ▶ normalement \equiv paramètre en entrée
- ▶ pour un tableau \equiv paramètre en **entrée-sortie**

(lié à la représentation mémoire que nous verrons plus tard)

Exemple : Remplir un tableau

```
public static void remplir( int[] tab, int val ) {  
    for(int i = 0; i < tab.length; i++) {  
        tab[i] = val;  
    }  
}
```

- ▶ L'appel pourrait être

```
int[] cotes = new int[16]; // Ne pas oublier de le créer  
remplir( cotes, 20 );
```

Tableau et méthode

Un tableau peut être une valeur de retour

Exemple : Créer un tableau avec valeur

```
public static int[] créer( int  taille , int  val ) {  
    int[] tab = new int[ taille ];  
    for(int i = 0; i < taille ; i++) {  
        tab[i] = val;  
    }  
    return tab;  
}
```

► L'appel pourrait être

```
int[] cotes = créer(16, 20);
```

Tableaux à plusieurs dimensions

Pour indiquer plusieurs dimensions, on multiplie les `[]`

Exemple :

```
int [][] tab = new int [3][4];
for( int i=0; i<3; i++) {
    for( int j=0; j<4; j++) {
        tab[i][j] = i*j*(i+j);
    }
}
```

Tableaux à plusieurs dimensions

On peut demander le nb de colonnes comme le nb de lignes

Exemple : afficher un tableau reçu en paramètre

```
public static void afficher (int [][] tab) {  
    for( int i=0; i<tab.length; i++) {  
        for( int j=0; j<tab[0].length; j++) {  
            System.out.print (tab[i][j] + " ");  
        }  
        System.out.println ();  
    }  
}
```

Erreurs fréquentes

Lorsque vous manipulerez des tableaux vous pourrez tomber sur ces exceptions

- ▶ `NullPointerException` : si vous essayez d'accéder à un élément d'un tableau qui n'a pas été créé (le tableau vaut **null** dans ce cas)
- ▶ `ArrayIndexOutOfBoundsException` : si vous donnez un indice qui n'existe pas (ex : `tab[10]` quand il n'y a que 10 éléments dans le tableau)

Leçon 17

La documentation Java

- Motivation
- La Javadoc
- Les tags
- Le code HTML
- Produire la documentation
- Pour une «bonne» documentation

Motivation

Documenter son code

- ▶ Pour qui ?
 - Le programmeur qui va **utiliser** le code
 - Le programmeur qui va **maintenir** le code (peut-être vous)
- ▶ Quel type de documentation ?
 - **Ce que fait** la méthode/classe
 - **Comment** elle le fait (peut être réduit au minimum si code lisible)

Motivation

Qui est intéressé par quoi ?

- ▶ Le programmeur-**utilisateur**
 - intéressé uniquement par le **quoi**
- ▶ Le programmeur-**mainteneur**
 - intéressé par le **quoi** et le **comment**

Motivation

Qui est intéressé par quoi ?

- ▶ Le programmeur-**utilisateur**
 - intéressé uniquement par le **quoi**
- ▶ Le programmeur-**mainteneur**
 - intéressé par le **quoi** et le **comment**

Où mettre la documentation ?

- ▶ Avec le code
 - Plus facile pour le maintenir
 - Plus de chance de garder la synchronisation avec le code
- ▶ Mais le programmeur-utilisateur n'a pas à voir le code pour l'utiliser

Motivation

→ Utilisation du **litterate programming**

- ▶ la documentation accompagne le code
- ▶ un outil extrait cette documentation pour en faire un document facile à lire
- ▶ de plus, toute la documentation suit la même structure, le même style
 - plus facile à lire

Javadoc

En Java, l'outil est javadoc

- ▶ Commentaire javadoc identifié par `/** ... */`

```
/**  
    Calcule et retourne le maximum de 2 nombres.  
*/
```

- ▶ documentation produite au format HTML
- ▶ On commente essentiellement
 - la classe : rôle et fonctionnement
 - les méthodes publiques : ce que ça fait, paramètres et résultats
- ▶ Se met **juste au dessus** de ce qui est commenté

Les tags

Utilisation de **tags** pour identifier certains éléments

Les plus courants :

- ▶ **@param** : décrit les paramètres
- ▶ **@return** : décrit ce qui est retourné
- ▶ **@throws** : spécifie les exceptions lancées
- ▶ **@author** : note sur l'auteur

Les tags

Exemple

```
/**  
 * Donne la racine carrée d'un nombre.  
 * @param nb le nombre dont on veut la racine carée.  
 * @return la racine carée du nombre.  
 * @throws IllegalArgumentException si le nombre est négatif.  
 */  
public static double sqrt( double nb ) {...}
```

- ▶ Les types sont déduits de la signature et ajoutés à la documentation
- ▶ La première phrase (terminée par un `.`) sert de résumé

Les tags

Résumé

Modifier and Type	Method and Description
static double	<code>sqrt</code> (double nb) Donne la racine carée d'un nombre.

Détail

sqrt

```
public static double sqrt(double nb)
```

Donne la racine carée d'un nombre.

Parameters:

`nb` - le nombre dont on veut la racine carée.

Returns:

la racine carée du nombre.

Throws:

`java.lang.IllegalArgumentException` - si le nombre est négatif.

Remarquez tout ce qui est ajouté!

Le code HTML

Peut contenir des balises HTML

Exemple :

```
/**
 * Indique si l'année est bissextile . Pour rappel :
 * <ul>
 *   <li>Une année qui n'est pas divisible par 4 n'est pas bissextile
 *     (ex: 2009)</li>
 *   <li>Une année qui est divisible par 4</li>
 * </ul>
 *   <li>est en général bissextile (ex: 2008)</li>
 *   <li>sauf si c'est un multiple de 100 mais pas de 400 (ex: 1900, 2100)</li>
 *   <li>les multiples de 400 sont donc bien bissextiles (ex: 2000, 2400)</li>
 * </ul>
 * </ul>
 * Plus formellement, <code>a</code> est bissextile si et seulement si <br/>
 * <code>a MOD 400 = 0 OU (a MOD 4 = 0 ET a MOD 100 != 0)</code>
 * @param année l'année dont on se demande si elle est bissextile
 * @return vrai si l'année est bissextile
 */
```

Le code HTML

Ce qui donne

estBissextile

```
public static boolean estBissextile(int année)
```

Indique si l'année est bissextile. Pour rappel :

- Une année qui n'est pas divisible par 4 n'est pas bissextile (ex: 2009)
- Une année qui est divisible par 4
 - est en général bissextile (ex: 2008)
 - sauf si c'est un multiple de 100 mais pas de 400 (ex: 1900, 2100)
 - les multiples de 400 sont donc bien bissextiles (ex: 2000, 2400)

Plus formellement, *a* est bissextile si et seulement si

$a \text{ MOD } 400 = 0 \text{ OU } (a \text{ MOD } 4 = 0 \text{ ET } a \text{ MOD } 100 \neq 0)$

Parameters:

année - l'année dont on se demande si elle est bissextile

Returns:

vrai si l'année est bissextile

Production de la documentation

On utilise la commande `javadoc`

- ▶ Création de la documentation d'une classe
`javadoc Temps.java`
- ▶ Possibilité de spécifier des sources multiples
`javadoc *.java`
- ▶ Créer la documentation dans un dossier spécifique
`javadoc -d doc *.java`
- ▶ Il y a beaucoup d'autres options ...
(cf. la documentation de `javadoc`)

Une bonne documentation

Une bonne *javadoc* décrit le **quoi** mais jamais le **comment**

- ▶ —→ Ne jamais parler de ce qui est privé
- ▶ Mauvais exemples :
 - *On utilise un for pour parcourir le tableau.*
 - *Pour aller plus vite, on stocke le prix hors tva dans une variable temporaire.*

Une bonne documentation

Ne pas écrire ce que javadoc écrit lui-même :

- ▶ Mauvais exemples :
 - *nb - un entier qui ...*
 - *La méthode sqrt ...*
 - *Cette méthode ne retourne rien.*
- ▶ Pour en savoir plus :

<http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>

Leçon 18

Tester le code

- Présentation
- Les tests
- Plan de tests
- JUnit
- Conclusion

Présentation

Tous les programmes réels contiennent des **bugs** (erreurs, défauts)

- ▶ Parfois même **beaucoup**
- ▶ **Inacceptable**
 - **Inconfort** pour l'utilisateur
 - **Perte** de temps, d'argent, de données, de matériel
 - Voire **danger** pour la vie humaine

Présentation

Rappel des types d'erreurs

- ▶ À la **compilation**
- ▶ À l'**exécution**, le programme **s'arrête**
- ▶ À l'**exécution**, le programme fournit une **mauvaise réponse**

On pourrait aussi parler d'autres défauts

- ▶ Trop **lent**
- ▶ Trop gourmand en **mémoire**

Présentation

« J'ai fait tourner le programme ;
il fonctionne très bien ! »

Insuffisant ! Il compile et tourne dans les cas les plus courants. Mais :

- ▶ Cas particuliers
- ▶ Comportement face à une défaillance de l'environnement
- ▶ Comportement face à une utilisation non conforme

Présentation

Pour produire un logiciel **sans bug** il faut

- ▶ Suivre une **méthodologie** éprouvée
 - Pour produire une première version avec peu de bugs (cf. *Analyse*)
- ▶ **Tester, tester** et ... **tester** encore !
 - Pour détecter ceux qui restent
 - Besoin d'outils pour nous aider
 - Le plus facile/rapide possible

Les tests

Plusieurs sortes de tests existent : unitaires, d'intégration, fonctionnels, non-régression, ...

- ▶ Nous ne pouvons pas tout aborder ici
- ▶ Nous nous intéressons aux tests **unitaires**

Pour en savoir plus :

[http://fr.wikipedia.org/wiki/Test_\(informatique\)](http://fr.wikipedia.org/wiki/Test_(informatique))

Les tests

Tests **unitaires** : test de **chaque méthode**

- ▶ Fait-elle ce qu'elle est censée faire ?
- ▶ C'est défini par la *spécification* (documentation)
- ▶ Idée : Si chaque méthode est correcte
→ le tout est correct
- ▶ Pas forcément suffisant
(ex : ne teste pas la performance)
→ d'autres types de tests existent

Plan de tests

Tester une méthode ne s'improvise pas

- ▶ Besoin d'un **plan** reprenant les tests à effectuer
 - Quelles **valeurs de paramètres** ?
 - Quel est le **résultat attendu** ?
- ▶ Préparé pendant que l'on code (ou même avant et éventuellement par une autre personne)
- ▶ Permet de s'assurer que l'on teste **tous les cas**

Plan de tests

On ne peut pas tester toutes les valeurs possibles

- ▶ Choisir des valeurs **représentatives**
 - Cas général / particuliers
 - Valeurs limites
- ▶ Il faut imaginer les cas qui pourraient mettre en évidence un défaut de la méthode

Plan de tests

On s'inspire des erreurs les plus fréquentes en programmation

- ▶ On commence/arrête trop tôt/tard une boucle
- ▶ On initialise mal une variable
- ▶ Dans un test, on se trompe entre $<$ et \leq
- ▶ Dans un test, on se trompe entre *ET* et *OU*
- ▶ ...

C'est un savoir-faire \implies Importance de l'expérience

Plan de tests - Exemple

Exemple : Soit la méthode

public static int max(**int**[] tab) ...

qui calcule la valeur maximale d'un tableau

- ▶ À quoi penser en plus du cas général ?
 - Le maximum est la première/dernière valeur
 - Le tableau ne contient qu'une seule valeur
 - Le tableau ne contient que des nombres négatifs

Plan de tests - Exemple

Plan de tests de la méthode *max*

#	tab	résultat	ce qui est testé
1	[1, 3, 0, 2]	3	cas général
2	[1, -3, -4, -2]	1	maximum au début
3	[1, 3, 4, 11]	11	maximum à la fin
4	[-1, -3, -4, -2]	-1	que des négatifs
5	[1]	1	tableau de taille 1

Plan de tests

Quand tester ? Le plus **souvent** possible

- ▶ Erreur plus facile à identifier/corriger
- ▶ Idéalement après chaque méthode écrite

Que tester ? **Tout**

- ▶ Le nouveau code peut mettre en évidence un problème dans le code ancien (**régression**)

JUnit

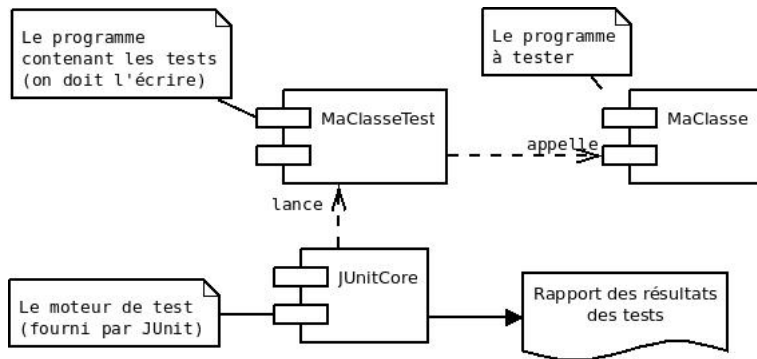
Comment tester ?

- ▶ Pas à la main : intenable
- ▶ Besoin d'un outil **automatisé**

JUnit : outil pour automatiser les tests unitaires

- ▶ Le programmeur fournit les tests,
- ▶ JUnit **exécute** tous les tests et
- ▶ établit un **rapport** détaillant les problèmes

JUnit



JUnit - En pratique

La classe de test contient **une méthode** de test **par cas**

- ▶ Autonome (ne reçoit rien ne retourne rien)
- ▶ Contient des **affirmations**
 - Appel de la méthode à tester
 - **Comparaison** entre le résultat **attendu** et le résultat **obtenu**

JUnit - En pratique

Exemple

```
@Test
public void max_cas1() {
    int[] tab = {1,3,0,2};
    assertEquals( 3, max(tab) );
}
```

- ▶ Reconnu comme un test unitaire grâce à l'**annotation** `@Test`
- ▶ Pas de **static**
- ▶ `assertEquals` vérifie que les 2 valeurs sont identiques
- ▶ Il y a aussi `assertTrue(val)`, `assertFalse(val)`, ...

JUnit - En pratique

Comment **lancer** les tests ?

```
java org.junit.runner.JUnitCore MaClasseTest
```

Résultat ?

- ▶ Nb de (méthodes de) tests effectués / réussis
- ▶ Détail sur les tests ratés
 - Nom du test
 - Résultat obtenu comparé au résultat attendu

JUnit - Exemple

Exemple : soit le code suivant à tester

```
package be.heb.esi.java1 ;

public class Outil {
    public static int max(int[] tab) {
        int max = 0;
        for(int i=0; i<tab.length; i++) {
            if (tab[i]>max) {
                max = tab[i];
            }
        }
        return max;
    }
}
```

JUnit - Exemple

La classe de test donnerait

```
package be.heb.esi.java1;  
import org.junit .Test; // Pour que @Test soit connu  
import static org.junit .Assert .*; // Pour assertEquals  
  
public class OutilTest {  
    @Test public void max_cas1() {  
        int [] tab = {1,3,0,2};  
        assertEquals( 3, Outil.max(tab) );  
    }  
  
    @Test public void max_cas4() {  
        int [] tab = {-1,-3,-4,-2};  
        assertEquals( -1, Outil.max(tab) );  
    }  
  
    // + plus les cas 2, 3 et 5  
}
```

JUnit - Exemple

On peut compiler la classe de test et la donner au moteur de test

```
javac OutilTest.java
java org.junit.runner.JUnitCore be.heb.esi.java1.OutilTest
JUnit version 4.5
..E
Time: 0,02
There was 1 failure :
1) max_cas4(OutilTest)
java.lang.AssertionError: expected:<-1> but was:<0>
[...]
```

FAILURES!!!

Tests run: 2, Failures : 1

Une idée du problème ?

JUnit - Tester les exceptions

Imaginons une méthode `sqrt` pour calculer la racine carrée d'un entier

- ▶ Hypothèse : elle doit lancer une exception en cas de paramètre négatif
- ▶ Le code pourrait ressembler à ceci

```
public static int sqrt(int val) {  
    if (val < 0) {  
        throw new IllegalArgumentException(  
            "Pas de racine carrée pour un entier négatif");  
    }  
    // suite : calcul de la racine carrée  
}
```

JUnit - Tester les exceptions

À priori c'est correctement écrit mais il faut le tester !

```
@Test(expected=IllegalArgumentException.class)
public void sqrt_cas_négatif() {
    sqrt(-1);
}
```

- ▶ Le test est réussi si la méthode lance l'**exception** indiquée

Conclusion

« Il faut vraiment tout tester ? »

- ▶ Compromis entre le temps que l'on consacre aux tests et la probabilité de trouver une erreur
- ▶ Trop simple → perte de temps
- ▶ Attention ! On commet vite une erreur même dans du code simple
- ▶ De plus, les tests unitaires aident à la «refactorisation» (cf. la leçon sur la lisibilité)

Conclusion

« *Any program feature without an automated test simply doesn't exist.* »
Kent Beck in *Extreme Programming Explained*.

Pour aller plus loin :

- ▶ Le site de JUnit : <http://www.junit.org>
- ▶ La Javadoc : <http://junit.sourceforge.net/javadoc>
- ▶ Télécharger JUnit :
<http://sourceforge.net/projects/junit/files/junit>

Leçon 19

Variables locales

- Présentation
- Allocation mémoire
- Déclaration
- Conventions sur les noms
- Valeur initiale
- Le concept de «portée»
- Constantes

Présentation

Désignation générique d'un **emplacement** de la **mémoire vive**

- ▶ **Possède un type**
- ▶ Ne peut contenir que des valeurs de ce type
- ▶ Allocation différente si type **primitif** ou **référence**

Allocation mémoire

Pour un type **primitif**

- ▶ Indique la zone mémoire (sur la pile/*stack*) où se trouve la **valeur**

Nom variable



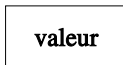
valeur

Allocation mémoire

Pour un type **primitif**

- Indique la zone mémoire (sur la pile/*stack*) où se trouve la **valeur**

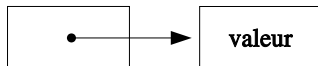
Nom variable



Pour un type **référence** (ex : `String`, tableau)

- La zone mémoire contient l'**adresse** de la zone mémoire (sur le tas/*heap*) contenant la valeur (indirection)

Nom variable



Déclaration

Déclaration locale à un **bloc**

Block :

{ BlockStatements_(opt) }

BlockStatements :

BlockStatement

BlockStatements BlockStatement

BlockStatement :

LocalVariableDeclarationStatement

Statement

Peut-on mélanger déclarations et instructions ?

Déclaration

(règles légèrement simplifiées)

LocalVariableDeclarationStatement :

`final(opt) Type VariableDeclarators ;`

VariableDeclarators :

VariableDeclarator

VariableDeclarators , VariableDeclarator

VariableDeclarator :

Identifiant

Identifiant = Expression

► **Exemples :**

- `int i;`
- `String nom, prénom;`
- `boolean ok=true, fini;`
- `char lettre , chiffre ='1';`

Nom d'une variable

Quel nom peut-on choisir ? Différence entre

- ▶ Ce qui est permis par Java
- ▶ Les conventions supplémentaires

Règles **imposées** par la grammaire

- ▶ Longueur illimitée
- ▶ Composé de *lettres*, de *chiffres*, \$ et _
(internationalisation)
- ▶ Ne commence pas par un chiffre
- ▶ \neq *keyword* ou *literal*
- ▶ Ex valides : `nom`, `Nom`, `Nom23`, `Unpeu2touT`
- ▶ Ex invalides : `2main`, `le total`, `for`, `true`, `12`

Nom d'une variable

Conventions **supplémentaires**

- ▶ Utilisées dans le monde entier
 - Eviter \$ et _
 - Commence par une minuscule
 - Plusieurs mots accolés \Rightarrow les suivants commencent par une majuscule (*mixedcase*)
 - Noms explicites (sauf abréviations courantes)
 - Articles omis
- ▶ Autres recommandations de ~~Sun~~ Oracle
 - Déclarer en début de bloc
 - Une déclaration par ligne

Valeur initiale

On peut indiquer une **valeur initiale**
(par **défaut**, **aucune** valeur n'est assignée)

- ▶ 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
```

- ▶ Exemple

```
int poidsKilo; // Un poids en kilos  
int poidsGramme = 1000*poidsKilo; // erreur à la COMPILE
```


Scope (portée) d'une variable locale

Définition : Le **scope** (**portée**) d'une variable indique la portion du programme où elle *existe*

Le *scope* d'une variable locale est

- ▶ le *block* de sa déclaration (entre `{}`)
- ▶ dès sa propre initialisation

Scope (portée) d'une variable locale

Exemples : bon ou pas ?

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

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

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

Constante

Clause **final** \Rightarrow constante

- Valeur donnée
 - Soit à la déclaration
 - Soit par assignation ultérieure

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

- Pourquoi une constante au lieu d'un littéral ?

Constante

Convention de nom différente

- ▶ Tout mettre en **majuscules**
- ▶ Utiliser `_` pour séparer les mots

Exemples

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

Leçon 20

Les expressions

- Les expressions entières
- Les expressions flottantes
- Les expressions caractères
- Les chaines de caractères
- Les expressions booléennes
- Les expressions relationnelles
- Les expressions conditionnelles
- Un mot sur les conversions

Définitions

Expression : calcul faisant intervenir une ou plusieurs valeur(s) pour une opération déterminée

Exemple : $1+2$

- ▶ 1 et 2 sont les opérandes
- ▶ + est l'opérateur
- ▶ l'expression est de type **int**
- ▶ la valeur de l'expression est 3

Les expressions entières

Opérateurs

- ▶ unaires : $+$ et $-$
- ▶ binaires : $+$, $-$, $*$,
 $/$ (division entière) et $\%$ (modulo)

Opérandes pouvant intervenir

- ▶ un **littéral** : $1 + 2$ vaut 3
- ▶ une **variable**
 - si i vaut 3, $i + 2$ vaut 5
 - si i vaut 3, $i + i$ vaut 6
- ▶ une **expression**
 - si i vaut 3, $(i + i) + 2$ vaut 8

Les expressions entières

Uniquement avec des opérandes d'un **même** type entier

- ▶ Type de l'expression = celui de ses opérandes
- ▶ Exemples :
 - $1 + 2$ vaut 3 de type **int**
 - $1L + 2L$ vaut 3 de type **long**
 - $3 / 2$ vaut 1 de type **int**

Priorité et associativité

Problème avec l'expression : $i + i * 2$

- ▶ Le compilateur va-t-il comprendre
 - $(i+i) * 2$?
 - $i + (i*2)$?

La stratégie d'évaluation se base sur

- ▶ la **priorité** d'un opérateur
- ▶ l'**associativité** des opérateurs de même priorité

priorité	opérateur	associativité
grande	$-$, $+$ unaires	\Leftarrow
	$*$, $/$, $\%$	\Rightarrow
faible	$-$, $+$ binaires	\Rightarrow

Tableau des priorités et associativités

Exercices : comment comprendre

- ▶ $3 + 3 * 2 + 1$?
- ▶ $3 + 3 * 2 / - 4 + 5 \% 8$?

Parenthèses pour forcer la stratégie

- ▶ Exemple : $(3 + 3) * (2 + 1) \rightarrow 6 * 3 \rightarrow 18$

Moralité : mieux vaut parenthéser complètement

- ▶ **ordre explicite** de l'évaluation
- ▶ **clarté** pour les autres utilisateurs

Erreurs de calcul

La **division par zéro**

- ▶ Lance une *exception* (*ArithmeticException*)
- ▶ Pour l'instant, **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 flottantes

Uniquement avec les opérandes d'un même type **flottant**

- ▶ Type de l'expression = type des opérandes
- ▶ Opérateurs unaires : $+$ et $-$
- ▶ Opérateurs binaires : $+$, $-$, $*$, $/$
- ▶ Le reste est identique aux entiers

Exemples : Donner la valeur et le type de

- ▶ $1.0 + 2.3$
- ▶ $1.0d - 2.3d$
- ▶ $7.0f / 3.5f$
- ▶ $1. / 3.$

Les expressions caractères

char est un type numérique entier

- ▶ **Aucun opérateur** spécifique
- ▶ On pourrait effectuer des calculs mais non recommandé

Exemple : 'a' + 'b'

Les chaînes de caractères

Un seul opérateur (binaire) disponible :

- ▶ **+** : concaténation de deux chaînes
- ▶ **Ex** : "Ja" + "va" vaut "Java"

Conversion si un des 2 opérandes n'est pas une chaîne.

- ▶ **Ex** : ""+3.5 vaut "3.5"
- ▶ **Ex** : "1"+2+3 vaut "123"
- ▶ **Ex** : 1+"2"+3 vaut "123"
- ▶ **Ex** : 1+2+"3" vaut "33" !

Les expressions booléennes

Opérateurs :

- ▶ unaire : **!** (non)
- ▶ binaires : **&&** (et) et **||** (ou)

priorité	opérateur	associativité
grande	- , + unaires, !	\Leftarrow
	* , / , %	\Rightarrow
	- , + binaires	\Rightarrow
	&&	\Rightarrow
faible	 	\Rightarrow

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**

Exemples

Comment évaluer ?

- ▶ **true && false || true**
- ▶ **false || false && ! true**
- ▶ **true || false && true**
- ▶ **false && (true || false)**
- ▶ **!(true || false) && true**

Résumé

Voici les règles qui résument tout ceci

Expression :

ConditionalOrExpression

ConditionalOrExpression :

ConditionalAndExpression

ConditionalOrExpression || *ConditionalAndExpression*

ConditionalAndExpression :

AdditiveExpression

ConditionalAndExpression && *AdditiveExpression*

AdditiveExpression :

MultiplicativeExpression

AdditiveExpression + *MultiplicativeExpression*

AdditiveExpression - *MultiplicativeExpression*

Résumé

MultiplicativeExpression :

UnaryExpression

MultiplicativeExpression * *UnaryExpression*

MultiplicativeExpression / *UnaryExpression*

MultiplicativeExpression % *UnaryExpression*

UnaryExpression :

+ *UnaryExpression*

- *UnaryExpression*

! *UnaryExpression*

Literal

(*Expression*)

Identifieur

- ▶ On y voit la priorité et l'associativité des opérateurs
- ▶ Pas suffisant pour intégrer toutes les contraintes additionnelles (par exemple?)

Les expressions relationnelles

Opérateurs de comparaison et d'égalité

- ▶ Opérandes doivent être de **même type**
- ▶ Le **résultat** est du type **boolean**

Opérateurs de **comparaison** : **<**, **>**, **<=** et **>=**

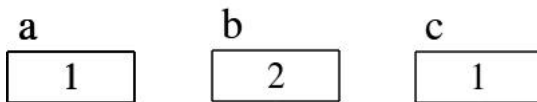
- ▶ Uniquement pour des **types numériques**
- ▶ Exemple : **true < false** n'est pas accepté
- ▶ Exemple : **"Absolu" < "Relatif"** non plus

L'égalité de valeurs

Opérateurs d'**égalité** : `==` et `!=`

- ▶ S'appliquent à **tous les types**
- ▶ Exemple : `true == false` est **false**
- ▶ Sens différent si type primitif ou référence

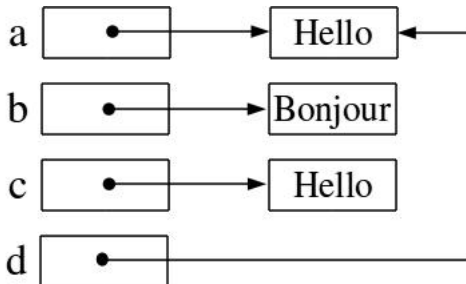
Type primitif : les **valeurs** sont comparées



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

Égalité de valeurs

Type référence : les **références** sont comparées



On a : $a \neq b$, $a \neq c$ mais $a == d$

Particularité du type String

```
{  
  String s1 = "Hello";  
  String s2 = "Hello";  
  String s3 = "Hel";  
  s3 = s3 + "lo";  
  System.out.println (s1==s2); // Vrai  
  System.out.println (s1==s3); // Faux  
}
```

- ▶ Réutilisation de l'espace mais pour les littéraux uniquement
- ▶ Pas si résultat d'un **calcul** ou **lecture** au clavier

Egalité de valeur

Pour les types références, on peut aussi utiliser la méthode `equals`

- ▶ Ne teste pas que les références sont identiques
- ▶ Mais bien que les **valeurs** référencées sont égales

```
{  
    String s1 = "Hello";  
    String s2 = "Hello";  
    String s3 = "Hel";  
    s3 = s3 + "lo";  
    System.out.println (s1.equals(s2)); // Vrai  
    System.out.println (s1.equals(s3)); // Vrai  
}
```

L'expression conditionnelle

Équivalent du **si-sinon** sous forme d'expression

condition ? si valeur vrai : si valeur faux

- ▶ Parfois plus lisible mais ne pas abuser
- ▶ C'est une expression, elle a une valeur

Exemples

- ▶ Quel type a l'expression :
heure < 12 ? "bonjour" : "bonsoir"
- ▶ Que vaudra abs ?

```
int n = -4;  
int abs = n > 0 ? n : -n;
```

Tableau des priorités et associativités

(opérateurs déjà vus)

priorité	opérateur	associativité
grande	-, + unaires, !	←←
	*, /, %	⇒⇒
	-, + binaires	⇒⇒
	<, >, <=, >=	⇒⇒
	==, !=	⇒⇒
	&&	⇒⇒
		⇒⇒
	?:	←←
faible		

Un mot sur les conversions

Peut-on mélanger les types ?

- ▶ Normalement pas
- ▶ Accepté si pas de perte d'information
- ▶ **Conversion** effectuée **automatiquement** par le compilateur
- ▶ Une leçon entière sera consacrée à ce sujet

Un mot sur les conversions

Voyons les situations les plus fréquentes

- ▶ Calcul mélangeant les entiers et les réels
 - Les entiers sont convertis en réels
 - **Ex** : `3.2/2` vaut 1.6 de type **double**
- ▶ Assigner un entier à un réel
 - L'entier est converti en réel
 - **Ex** : `double d = 1; // d vaut 1.0`
- ▶ Assigner un réel à un entier
 - Refusé ... sauf si demandé explicitement (**casting**)
 - **Ex** : `int i = 1.2; // refusé`
 - **Ex** : `int j = (int) 1.6; // j vaut 1`

Leçon 21

Les assignments

(et autres « expressions - instructions »)

- Les assignments
- Post/pré incrémentation/décrémentation
- Appel de méthode
- Recommandation

L'assignation

(version légèrement simplifiée)

Assignment :

LeftHandSide = Expression

LeftHandSide :

Identifier

ArrayAccess

► Exemples

```
bro1 = 1  
bro1[i]=j
```

► Tiens ! Pas de ; à la fin ?

Les expressions instructions

En fait, l'assignation est une **expression**

- ▶ qui peut devenir une **instruction** (*statement*)
- ▶ par ajout d'un ;
- ▶ la valeur est perdue

Une expression qui peut devenir une instruction s'appelle une *statement expression*

Statement :
ExpressionStatement
(...)

ExpressionStatement :
StatementExpression ;

StatementExpression :
Assignment
(...)

L'assignation - expression

Une **assignation** est d'abord une **expression**

- ▶ Son type : le type de la variable
- ▶ Sa valeur : la valeur du *left hand side*
- ▶ Peut donc intervenir comme élément d'une autre expression
- ▶ Priorité faible et associative de *droite à gauche*

L'assignation - expression

Ceci explique pourquoi on peut écrire

```
i = j = k = l = 0;  
i = (j = i+j) + 1;  
f(i=1,j=0);  
while( (i=i-1) != 0 ) {...}  
while( ok=true ) {...} // boucle infinie !
```

Mais pas

```
i = j = k = l = 0 // erreur compilation  
while( ok=true; ) {...} // idem
```

Autres Assignations

Il existe d'autres **opérateurs d'assignation**

Assignment :

LeftHandSide AssignmentOperator Expression

*AssignmentOperator : one of = *= /= %= += -=*

- ▶ `var += expr` équivaut à `var = var + expr`
- ▶ **Ex** : `i+=1` équivaut à `i = i + 1`
- ▶ Que penser de ?

```
i = 2;  
i = i = (i*=2) + 1;  
(i+1) -= 2;
```

Les expressions instructions

Existe-t-il d'autres *expressions instructions* ?

StatementExpression :

Assignment

PreIncrementExpression

PreDecrementExpression

PostIncrementExpression

PostDecrementExpression

MethodInvocation

ClassInstanceCreationExpression (cf. OO)

Post/pré incrémentation/décrémentation

`++` permet d'incrémenter une variable

- ▶ Peut se placer avant ou après la variable
- ▶ `i++`; \equiv `++i`; \equiv `i+=1`; \equiv `i=i+1`;

Il existe tout de même une différence

- ▶ `++i` : `i` est incrémenté **avant** d'être utilisé
- ▶ `i++` : `i` est incrémenté **après** avoir été utilisé
- ▶ Exemples

```
int i = 5;
i = i++;
i = ++i;
i = i++ + ++i;
i = (i++)++;
i = 2++;
```

Post/pré incrémentation/décrémentation

-- fonctionne comme ++ mais décrémente

► Exemples

```
int i = 5;  
i = i--;  
i = i-- - ++i;
```

Comment comprendre ceci ?

```
i = i++ + i;  
i = i- -- i;  
i = i+++1;  
i = i-- - -- i;  
i = i+++++i;
```

► Aide : penser au fonctionnement du compilateur

Tableau des priorités et associativités

priorité			associativité
forte	postfixes unaires	(params), ., expr++, expr--	⇒
	préfixes unaires	++expr, --expr, -, +, !, new	⇐
	multiplicatif	*, /, %	⇒
	additif	-, +	⇒
	relationnels	<, >, <=, >=	⇒
	égalité	==, !=	⇒
	et	&&	⇒
	ou		⇒
faible	condition	:?	⇐
	assignments	=, +=, -=, *=, /=, %=	⇐

Ordre d'évaluation

Comment comprendre ceci ?

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

- ▶ Problème dans beaucoup de langages
- ▶ En Java, **évaluation garantie de gauche à droite**

Aussi 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)
```


Appel de méthode

L'**appel de méthode** est un autre cas d'expression-instruction

- Explique pourquoi ceci est valable

```
a = f(1);  
f(1);  
Math.sqrt(4);
```

- La valeur de retour est **perdue** !

Recommandation

Comment utiliser les assignments (en tous genres) ?

Afin d'assurer une bonne lisibilité du code :

- ▶ **Jamais** comme une **expression**
- ▶ Mais **toujours** comme une **instruction**

Leçon 22

Instructions

- L'instruction vide
- La notion de bloc
- Le « if-else »
- Le « switch »
- Le « while »
- Le « do-while »
- Le « for »
- L'étiquette
- Le « break »
- Le « continue »

Les instructions

Statement :

Block

EmptyStatement

LabeledStatement

BreakStatement

ContinueStatement

ExpressionStatement

IfThenStatement

IfThenElseStatement

SwitchStatement

WhileStatement

DoStatement

ForStatement

ReturnStatement

AssertStatement

SynchronizedStatement

ThrowStatement

TryStatement

L'instruction vide

Cette instruction ne fait rien et se déroule toujours bien

EmptyStatement :

;

Le bloc

Block :

{ BlockStatements_(opt) }

BlockStatements :

BlockStatement

BlockStatement BlockStatements

BlockStatement :

LocalVariableDeclarationStatement

Statement

Généralement on déclare les variables en début de bloc

if, if-else

La syntaxe (légèrement simplifiée) :

IfThenStatement :

if (Expression) Statement

IfThenElseStatement :

if (Expression) Statement else Statement

- ▶ Peut être suivi par une instruction, pas forcément un bloc
- ▶ **Expression** est de type **booléen**

if, if-else

```
package be.heb.esi.java1 ;
import java.util.Scanner;
public class Test {
    public static void main(String[] args) {
        Scanner clavier = new Scanner(System.in);
        int nombre1;

        nombre1 = clavier.nextInt ();
        System.out.println (nombre1 + " est un nombre ");
        if (nombre1 >= 0)
            System.out.println (" positif ");
        else
            System.out.println (" négatif ");
    }
}
```


if, if-else

Mais aussi

```
public static void afficheTableau (int [] tableau) {  
    if (tableau.length > 0) {  
        System.out.println ("Contenu du tableau:");  
        for (int i=0; i<tableau.length; i++)  
            System.out.println ( "en "+i+ ": " +tableau[i]);  
    }  
    else  
        System.out.println ("tableau vide");  
}
```

if-else - dangling else

Qu'en est-il de :

```
if (nombre1 >= 0)
    if (nombre1 == 0)
        System.out.println ("nul");
else
    System.out.println ("?");
```

- Doit-on afficher **négatif** ou **positif** ?

if-else - dangling else

L'interprétation correcte :

```
if (nombre1 >= 0)
    if (nombre1 == 0)
        System.out.println ("nul");
    else
        System.out.println (" positif ");
```

- ▶ Le **else** est rattaché au **if** le plus proche
- ▶ Comment le rattacher au **if** extérieur ?

if, if-else - exemples

Utilisation admise pour les erreurs

```
if (condErreur)
    throw new Exception ("justification ");
instructions
```

- ▶ Valider les paramètres en début de méthode
- ▶ À isoler du code qui suit

if-else - exemples

```
public static void afficheTableau (int [] tableau) {  
    if (tableau==null)  
        throw new IllegalArgumentException("pas de tableau!");  
  
    if (tableau.length > 0) {  
        System.out.println ("Contenu du tableau:");  
        for (int i=0; i<tableau.length; i++) {  
            System.out.println ( "en " + i + ": " + tableau[i]);  
        }  
    }  
    else {  
        System.out.println ("tableau vide");  
    }  
}
```

switch

switch : proche du **selon que**

Exemple :

- En logique

```
SELON QUE (jour) VAUT
0 : ECRIRE "Lundi"
1 : ECRIRE "Mardi"
2 : ECRIRE "Mercredi"
3 : ECRIRE "Jeudi"
4 : ECRIRE "Vendredi"
5 : ECRIRE "Samedi"
6 : ECRIRE "Dimanche"
autres : ECRIRE "Erreur"
FIN SELON
```

switch


► En Java :

```
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;  
    default : System.out.println ("Erreur");  
}
```

► Ici les accolades sont obligatoires

switch

Remarques

- ▶ L'expression à évaluer est de type limité : **char**, **byte**, **short**, **int** ou **String** 
- ▶ Les expressions constantes sont toutes différentes
- ▶ **break** pour terminer le **case**
- ▶ **default** peut être omis
- ▶ S'il apparaît, il doit être unique (on recommande de le mettre à la fin)
- ▶ Blocs admis mais pas recommandés

switch

- 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_de_mois_incorrect");  
}
```

selon que avec conditions

Le switch ne permet pas de traduire le selon-que **avec condition**

```
if (âge < 5)
    System.out.println ("hors_catégorie");
else if (âge < 6) // 5 ans
    System.out.println ("groupe_poussins");
else if (âge < 9) // 6,7,8
    System.out.println ("groupe_benjamins");
else if (âge < 11) // 9,10
    System.out.println ("groupe_pupilles");
else // > 10
    System.out.println ("hors_catégorie");
```

- Remarquez l'indentation dans ce cas

while

`while (Expression) Statement`

- ▶ Instruction quelconque
- ▶ Expression booléenne
- ▶ L'expression booléenne est recalculée dans la boucle
- ▶ **Ex** : recherche du premier élément non nul

```
int pos = 0;  
while (pos < tableau.length && tableau[pos] == 0)  
    pos++;
```

while - exemples

Remarquez l'ordre des opérandes

```
int pos = 0;  
while (tableau[pos] == 0 && pos < tableau.length)  
    pos++;
```

- Provoque une **ArrayIndexOutOfBoundsException** si aucun élément non nul

while - exemples

Une autre écriture

```
int pos = 0;  
boolean nonNulTrouvé = false;  
while (! nonNulTrouvé && pos < tableau.length)  
    if (tableau[pos] != 0)  
        nonNulTrouvé = true;  
    else  
        pos++;
```

while - exemples

Dans certains cas, les accolades améliorent la lisibilité

```
int pos = 0;
boolean nonNulTrouvé = false;
while (! nonNulTrouvé && pos < tableau.length){

    if (tableau[pos] != 0)
        nonNulTrouvé = true;
    else
        pos++;

}
```

while - exemples

Qu'en est-il de :

```
cpt =0;  
while (cpt < 100) ;  
    cpt++;
```

```
cpt =0;  
while (cpt < 100)  
    cpt = cpt++;
```

do-while

En logique

Faire
instructions
Jusqu'à ce que condition

En Java :

do
instruction
while (condition);

Attention ! C'est en fait un **répéter .. TANT QUE**

for

Plus général que le «pour» de Logique

Contrôles de la boucle en tête de boucle pour une meilleure lisibilité

- ▶ l'initialisation
- ▶ le test
- ▶ l'incrémentation (au sens large)

Utilisé en général quand le nombre d'itérations est connu

for

BasicForStatement :

*for (ForInit_(opt) ; Expression_(opt) ; ForUpdate_(opt))
Statement*

EnhancedForStatement :

for (Type Identifier : Expression) Statement

ForInit :

*StatementExpressionList
LocalVariableDeclaration*

ForUpdate :

StatementExpressionList

StatementExpressionList :

*StatementExpression
StatementExpressionList , StatementExpression*

for

Que penser de ceci ?

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

for – Ordre d'exécution

- ➊ l'initialisation : *ForInit*
- ➋ l'évaluation du test : *Expression*
- ➌ si le test vaut true alors
 - ➊ le corps : *Statement*
 - ➋ l'incrémentation : *ForUpdate*
 - ➌ retour à l'étape 2
- ➍ sinon, on passe à l'instruction suivant le for

Lien entre for et while

Toute boucle *for* peut s'écrire avec une boucle *while* :

```
for( ForInit ; Expression ; ForUpdate){  
    Statement  
}
```

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

À un détail près : toute variable déclarée dans le «ForInit» n'existe plus en dehors du «for».

for - Exemples

Exemples divers

```
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;  
}
```

foreach

ForStatement :

BasicForStatement

EnhancedForStatement

EnhancedForStatement :

for (Type Identifier : Expression) Statement

- ▶ Écriture simplifiée du **for**
- ▶ Pour parcourir un tableau
(+ d'autres choses. cf. leçon sur les listes)

foreach - exemples

```
double[] tab = new double[10];  
...  
for( double valeur : tab ) {  
    System.out.println ( valeur );  
}
```

Écriture simplifiée pour

```
double[] tab = new double[10];  
...  
for( int i=0; i<tab.length; i++) {  
    double valeur = tab[i];  
    System.out.println ( valeur );  
}
```


foreach - exemples

Attention ! On a accès aux éléments mais pas à l'indice

- ▶ OK pour consulter
- ▶ mais **pas pour modifier**

```
double[] tab = new double[10];  
...  
for( double valeur : tab ) {  
    valeur = 1.0; // aucune erreur mais ...  
                // ne modifie pas le tableau !  
}
```

étiquette

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

LabeledStatement :
Identifieur : Statement

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

break

Pour **arrêter brutalement une instruction**

BreakStatement :

`break Identifieur(opt) ;`

- ▶ Si pas d'étiquette → arrête la première instruction **while/for/do/switch** englobante
- ▶ Si étiquette → arrête brutalement l'instruction étiquetée et passe à la suivante

break - exemples

```
int nb;  
entrée : while(true) {  
    nb = clavier.nextInt();  
    if (nb>0)  
        break entrée;  
    System.out.println ("Mauvais nb. Recommencer.");  
}
```

```
int nb;  
while(true) {  
    nb = clavier.nextInt();  
    if (nb>0) break;  
    System.out.println ("Mauvais nb. Recommencer.");  
}
```

break - exemples

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

continue

Pour une **nouvelle itération d'une boucle**

ContinueStatement :

`continue` *Identifieur*_(opt) ;

- ▶ Si pas d'étiquette → recommence la première instruction répétitive englobante
- ▶ Si étiquette → recommence la boucle étiquetée

continue - exemples

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

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

break-continue

À utiliser avec parcimonie

Exemple tiré d'un code réel

- ▶ Que fait-il ?
- ▶ Comment l'écrire plus proprement ?

break - examples

```
public static int inbuff(char[] meule, char[] aiguille) {  
    int i, j, t=-1, sizem=meule.length, sizea=aiguille.length;  
  
    for (i=0; i<=sizem-sizea; i++) {  
        for (j=0; j<sizea; j++) {  
            if (meule[i+j] != aiguille[j])  
                break;  
            t=j;  
        }  
        if (t==sizea-1) {  
            t=i;  
            break;  
        }  
        else  
            t=-1;  
    }  
    return t;  
}
```

Leçon 23

Les tableaux

- Type
- Déclaration
- Création
- Représentation
- Taille
- Parcours
- Assignment en bloc
- Erreurs et exceptions
- Tableau et méthode

Type

Un tableau contient un nombre **déterminé** de composants (éléments) de **même type**

- ▶ éléments de type T
⇒ type du tableau = $T[]$
- ▶ **Exemples**
 - $\text{int}[]$ est le type *tableau d'entiers*
 - $\text{String}[][]$ est le type *tableau à 2 dimensions de chaines de caractères*
- ▶ La taille ne fait pas partie du type

Déclaration

Déclarations valides

- ▶ **int** [] entiers ;
- ▶ **short** [][] shortss ;

Exemples valides **mais non recommandés** (archaïsme)

- ▶ **int** entiers [];
- ▶ **short** shortss [][];

Création

Voyons (une partie de) la grammaire pour la création d'un tableau :

ArrayCreationExpression :

new TypeName DimExprs Dims_(opt)

new TypeName Dims ArrayInitializer

On voit donc qu'on peut créer un tableau :

- ▶ En donnant **des** tailles
- ▶ En donnant les valeurs

Création

Création en donnant des tailles

ArrayCreationExpression :

*new TypeName DimExprs Dims*_(opt)

DimExprs :

DimExpr

DimExprs DimExpr

DimExpr :

[*Expression*]

Dims :

[]

Dims []

Création

Exemples :

```
int [] t1 = new int[3];  
int [][] t2 = new int[3][2];
```

```
int [] t1;  
int [][] t2;  
int nb = clavier.nextInt();  
t1 = new int[nb];  
t2 = new int[nb][nb];
```

Création

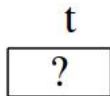
Éléments initialisés à une **valeur par défaut**

- ▶ Numérique : **0**
- ▶ Booléen : **false**
- ▶ Référence : **null** (**référence vers rien**)

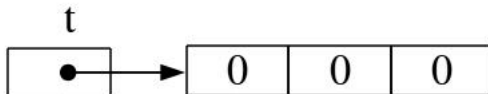
Représentation

Un *Tableau* est un type **référence**

► **Ex** : `int [] t;`



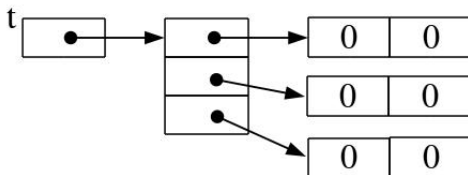
► `t = new int[3];`



Représentation

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

- Un tableau de 3 tableaux de 2 entiers



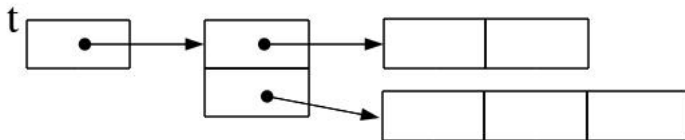
- Représentation interne \neq vision classique

0	0
0	0
0	0

Représentation

Pour les tableaux à plusieurs dimensions

- ▶ Chaque élément d'un tableau à deux dimensions est un tableau indépendant
- ▶ La taille ne fait pas partie du type
- ▶ \Rightarrow Chaque élément peut être d'une **taille différente**

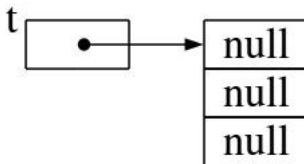


Création

On peut omettre les dernières tailles

- ▶ Le tableau est créé en partie
- ▶ **Exemple :**

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



- ▶ Le reste sera créé plus tard

Création

Création en donnant les valeurs

ArrayCreationExpression :

new TypeName Dims ArrayInitializer

ArrayInitializer :

{ VariableInitializers_(opt) ,_(opt) }

VariableInitializers :

VariableInitializer

VariableInitializers , VariableInitializer

VariableInitializer :

Expression

ArrayInitializer

Création

Exemple de format long

```
int[] t1 = new int[] {4,5,6};  
int[] t2;  
t2 = new int[] {4,5,6};
```

Écriture abrégée (uniquement à la déclaration)

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

Exercice : Donnez la représentation mémoire

- ▶ `int [][] entierss = {{1,2},{3,4,5}};`
- ▶ `int [][] entierss = {{1,2},null};`

Création

Exercice : Les créations suivantes sont-elles correctes ?
Pourquoi ?

- ▶ `int [][] t = new int[2];`
- ▶ `int [][] t = new int[] {1,2};`
- ▶ `int [][] t = new int[] {{1},{2}};`
- ▶ `int [][] t = new int[3][2] {1,2};`
- ▶ `int [][] t = new int[3][2] {null,null};`

Création

Exercice : Lecture d'un vecteur.

Attention ! Ceci n'est pas correct. Pourquoi ?

```
import java.util.Scanner;
public class LectureVecteur {
    public static void main(String[] args) {
        Scanner clavier = new Scanner(System.in);
        int[] vecteur;    // le vecteur à lire
        int  taille;      // taille du vecteur
        taille = clavier.nextInt();
        for (int i=0; i<taille; i=i+1)
            vecteur[i] = clavier.nextInt();
    }
}
```


Taille

La taille doit être un **int**

- ▶ Jusqu'à 2 milliards d'éléments ;-)
- ▶ Peut être **nulle**
 - ex : `int [][] t = new int[0][2];`
 - peut avoir un sens comme cas limite
- ▶ Ne peut **pas** être **négative**
 - sinon une exception est lancée
 - pas vérifié à la compilation même si constante
- ▶ Est une expression générale
 - ex : `int [] t2 = new int[clavier.nextInt ()];`

Taille

Rappel : on connaît la taille via `length`

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) );
```

Taille

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]);
    }
}
```

Taille

Exemple : Parcours 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]);
    }
}
```

foreach

Rappel : Le **foreach** simplifie le parcours d'un vecteur

```
public static void afficher ( int[] tab ) {  
    for( int val : tab ) {  
        System.out.print( val + " " );  
    }  
}
```

On peut l'utiliser pour une matrice

- version qui utilise le parcours d'un vecteur

```
public static void afficher ( int [][] mat ) {  
    for( int[] ligne : mat ) {  
        afficher ( ligne );  
        System.out.println ();  
    }  
}
```

foreach

- ▶ version qui refait tout

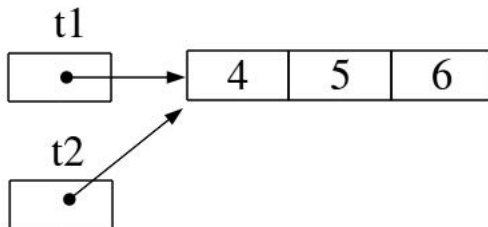
```
public static void afficher ( int [][] mat ) {  
    for( int [] ligne : mat ) {  
        for( int val : ligne ) {  
            System.out.print( val + " " );  
        }  
        System.out.println ();  
    }  
}
```

- ▶ Peut-on utiliser le **foreach** pour un parcours **colonne par colonne** ?

Assignation

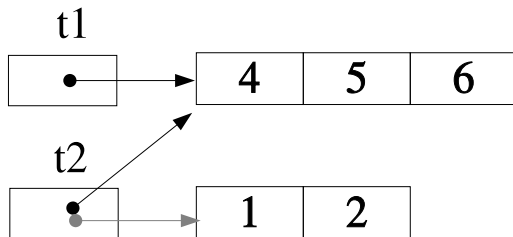
Un tableau est de type référence

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



Assignation

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



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

Accès aux éléments

Pour accéder à un élément on donne tous les indices

Exemple

```
int [][][] t = new int [4][5][3];  
t [0][2][1] = 1; // Faites un schéma mémoire !
```

Cela a du sens de n'en spécifier que certains

Exemple

```
int [] t = { 4,5,6 };  
int [][] t2 = { {1,2,3}, {4,5} };  
t2[1] = t; // Faites un schéma mémoire !
```

Création

Exemple : Autre écriture pour la création d'un tableau.

```
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];
```

Création

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];
    }
}
```

Création

Exemple : Triangle de Pascal

```
import java.util.Scanner;
public class TrianglePascal {
    public static void main(String[] args) {
        Scanner clavier = new Scanner(System.in);
        int taille = clavier.nextInt();
        int [][] pascal; // le triangle de Pascal
        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];
        }
    }
}
```

Erreurs et exceptions

Lors de l'accès à un élément

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

Erreurs et exceptions

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
```

Erreurs et exceptions

Lors d'une assignation

- ▶ 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};
```

Méthode

Passer un tableau en paramètre

= Passer une copie de la référence au tableau

- La méthode agit sur le tableau et pas une copie

Exemple : Un tableau en argument

```
public class TableauEnArgument {  
    public static void iniTableau(int[] tab) {  
        for(int i=0; i<tab.length; i++)  
            tab[i]=0;  
    }  
  
    public static void afficheTab(int[] tab) {  
        for(int i=0; i<tab.length; i++)  
            System.out.print(tab[i]+" ");  
    }  
}
```


Méthode

Exemple : Utilisation

```
public class Test {  
    public static void main(String[] args) {  
        int [] tableau = new int [] {8,4,3,9};  
        System.out.print ("tableau_avant: ");  
        TableauEnArgument.afficheTab(tableau);  
        TableauEnArgument.iniTableau(tableau);  
        System.out.print ("\ntableau_après: ");  
        TableauEnArgument.afficheTab(tableau);  
    }  
}
```

Méthode

Exemple : Un tableau en retour

```
public static int [] tableauEnRetour(int n) {  
    int [] tableau = new int[n];  
    for(int i=0;i<n;i++)  
        tableau[i]=i+1;  
    return tableau;  
}
```

- ▶ Un appel de la méthode **tableauEnRetour** fournira une référence à un tableau
- ▶ Il est possible de modifier les valeurs des éléments

Méthode principale

La méthode `main` reçoit un tableau en argument

```
public static void main(String[] args)
```

- ▶ Il s'agit d'**arguments** fournis au programme
- ▶ **Comment ?** via la ligne de commande

```
java Test mes arguments
```

- ▶ Arguments séparés par un (des) espace(s)

Méthode principale

Exemple :

```
public class Miroir {  
    public static void main(String[] args) {  
        System.out.print (args.length + " :");  
        for(int i=args.length-1; i>=0; i--)  
            System.out.print (args[i] + " ");  
    }  
}
```

```
> java Miroir Un message à l'envers  
4: l'envers à message Un
```

```
> java Miroir "Un message à l'envers"  
1: Un message à l'envers
```

Arrays

La classe **Arrays** offre des outils pratiques

- ▶ **equals** : teste l'égalité de tableaux
- ▶ **fill** : remplit tout un tableau avec une même valeur
- ▶ **toString** : retourne une chaîne reprenant les éléments du tableau
- ▶ **copyOf**, **sort**, ...

Crédits

Ce document a été produit avec les outils suivants

- ▶ La distribution **Ubuntu** du système d'exploitation **Linux**
- ▶ **LaTeX** comme système d'édition
- ▶ La classe **Beamer** pour les transparents
- ▶ Les packages **listings**, **fancyvrb**, ...
- ▶ Les outils **make**, **rubber**, **pdfnup**, ...