- initiation à la programmation
- apprentissage de bons comportements
- implémentation sur un OS (*operating system*)

1. Aborder des thèmes comme : lisibilité, robustesse, documentation, tests
2. Capacités de déverminage
3. Autonomie dans le travail
4. On peut aussi aborder le choix du langage. Pourquoi Java ?

Supports et ressources

Rien

- pas de syllabus ;
- pas de livre ;

Quoique

- les slides sur GitHub ;
- des liens, des documents… sur poÉSI ;
- un forum de discussion, fora.

1. Importance d'un livre d'accompagnement MAIS
2. Attention aux livres qui se concentrent sur un point spécifique du langage ou sur une API spécifique
3. Connaissance primordiale de l'anglais technique

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)

1. `http://en.wikipedia.org/wiki/Source_lines_of_code` donne des exemples de taille de programmes
2. On peut faire référence à Code Studio

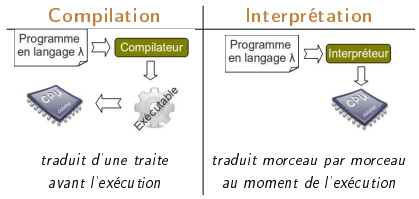
Classes de langages : machine, assembleur, haut niveau, structuré, orienté objet, fonctionnels, orientés aspects...

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

1. On peut aborder ici l'historique des langages : machine, assembleur, haut niveau, structuré, orienté objet, fonctionnels, orientés aspects...

Le problème de la traduction

Un ordinateur ne comprend que le langage machine.
Nécessité d'une **traduction**



traduit d'une traite avant l'exécution

traduit morceau par morceau au moment de l'exécution

1. Faire une comparaison des avantages/inconvénients
2. Facilité de distribution, rapidité, facilité de développement

- Java est-il installé sur ma machine ?
- Puis-je commencer à écrire un programme Java ?
- Qu'ai-je pris comme notes ?

S'il reste du temps, introduire les termes : JRE, JDK, ME, SE, EE. Qui seront introduits dans la séance 2. Mais c'est toujours bon de se répéter un peu.

Java serait lent à interpréter (langage haut niveau). Donc, introduction d'un niveau intermédiaire, le Bytecode, qui est proche d'un langage d'assemblage, plus rapide à interpréter. C'est en fait le langage de la JVM

Les éditions de Java

- Java SE (édition standard)
- Java ME (édition mobile - plus léger)
- Java EE (édition entreprise - plus complet)

Où trouver javac et java ?
JRE (*Java Runtime Environment*)
JDK (*Java Development Kit*)

Montrer le site d'oracle.

Préciser également que l'on peut l'installer via son gestionnaire de paquet pour ceux qui ont un linux. *openjdk-8*

L'image représente le cadre dans lequel on joue. Le cadre pour le programme de base.

Crédit photo

- Apprendre à reconnaitre **rapidement** les erreurs fréquentes
- **Déboguer** son code ; la méthode de l'homme pauvre et/ou le débogueur
- Mettre des tests en œuvre

Pour ceux qui veulent. . . un bel article

`http://namok.be/blog/?post/2012/03/23/debuguer-son-code-sans-utiliser-ide`

Crédit photo

Découper le code

Expliquer pourquoi découper le code. Algo a une volonté d'expliquer que ce découpage soit **naturel**. Évitons donc d'écrire des règles.
L'idée c'est : réutilisabilité, scinder la difficulté, déverminer, lisibilité, répartition du travail.

On aime qu'une méthode porte un nom explicite, résolve un problème précis et clairement défini, qu'elle soit documentée, etc.

Le slide suivant est là pour donner l'envie de venir à la séance suivante où l'on parle de la javadoc.

La documentation, une touche de couleur qui ne « sert à rien » . . . mais qui change tout !

Crédit photo
Crédit photo

- 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)

Qui est intéressé par quoi ?

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

Un **package** donne un nom complet à une classe

- mon.paquet.MaClasse,
- be.he2b.esi.java1.MaClasse,
- java.util.Scanner,
- org.apache.struts2.components.Anchor

1. regroupement de classes liées
2. unicité des noms de classe
3. identifieurs séparés par des .
4. tout en minuscules
5. adresse internet inversée (unicité)

Créer ses packages

Qu'est-ce qui va changer en pratique ?

- La compilation ne change pas :

```
javac NomClasse.java
```

- L'exécution change :

```
java nom.paquet.NomClasse
```

Cela a une incidence sur l'endroit où placer le **bytecode**

1. En profiter pour insister sur la différence entre des appels de méthodes dans la même classe, dans des classes différentes de même package, de packages différents.

Un *littéral*, c'est la représentation d'une valeur

Ne pas avoir peur d'insister car une erreur circule dans un résumé d'étudiants. Ils ont tendance à croire qu'un littéral c'est type.

On peut relire Unicode et tutti quanti

1. \n (linefield), \r (carriage return), \t (tabulation), \b (backspace), \', \", \\
2. Par exemple : '\n', '\\', '\"
3. Pour utiliser le code Unicode
4. Par exemple : '\u0F40' pour le KA tibétain

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

1. Capacité limitée (out of range possible)
2. Précision limitée (cfr. 10^{-30} et $(1 + 10^{-30}) - 1$)

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

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
  $\longrightarrow$ le tout est correct
- Pas forcément suffisant
  (ex : ne teste pas la performance)
  $\longrightarrow$ d'autres types de tests existent

- 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**

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

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 $\Longrightarrow$ Importance de l'expérience

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

Je vous présente, JUnit, l'architecte et exécutant des tests, celui qui mettra en place toutes les pièces du puzzle !

Où se trouve la classe org.junit.runner.JUnitcore ?

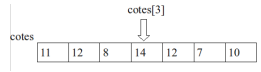
Première icône, installation.
Deuxième icône, où trouver JUnit ?

Reparler du CLASSPATH, de l'endroit où placer le jar . . . et de dire ce qu'est un jar.

« Il faut vraiment tout tester ? »

- Compromis entre le temps que l'on consacre aux tests et la probabilité de trouver une erreur
- Trop simple $\longrightarrow$ 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)

Nécessité de manipuler **plusieurs variables similaires** auxquelles on accède par un **indice**



Pourquoi pas plusieurs variables ?

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

Déclaration

Exemples

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

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

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

Tableau et méthode

Exemple

```
public static void méthodeFausse(double[] réels) {
    double[] réelsDePassage = {4.2, -7, Math.PI};
    réels = réelsDePassage; // INUTILE
}
```

Quel que soit l'appel, le tableau que l'on passe en paramètre ne sera pas modifié

Différencier passage de paramètre par valeur et en entrée-sortie. Comparer la situation en algorithmique et en java

Règles **imposées** par la grammaire

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

Conventions **supplémentaires**

- Utilisées dans le monde entier
  - Eviter $ et _
  - Commence par une minuscule
  - Plusieurs mots accollés ⇒ 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

Opérateurs entiers

$+ -$ $+ - * / \%$

Opérandes pouvant intervenir
- un littéral
- une variable
- une expression

Distinguer les opérateurs binaires et les unaires

Donner des exemples (beaucoup) d'expressions entières

i+i*2

(i+i)*2 ou i+(i*2) ?

Notions de priorité des opérateurs

En gros : « comme en math »

Les chaines de caractères

Opérateur

+

- Un seul opérateur pour la **concaténation** de deux chaines.
- Conversion si un des 2 opérandes n'est pas une chaine.

Donner des exemples.

Donner des exemples où il y a conversion et où la priorité à de l'importance

Opérateurs booléens

! && ||

Tables de vérité

(ET)	true	false
true	true	false
false	false	false

(OU)	true	false
true	true	true
false	true	false

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

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

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

Cas des opérateurs **binaires**

Si opérandes de types différents

$\Longrightarrow$ on *élargit* le moins large

On élargit au minimum vers **int**

Car les opérateurs n'existent pas en dessous

- conversion dans le type que possède déjà l'expression
- par facilité (si on n'est pas sûr)
- parfois nécessaire pour aider le compilateur (cf. OO)
- permise aussi dans les autres contextes car simplifie les règles

Crédit photo