ENSTA BRETAGNE

Java object-oriented programming

Raúl Mazo
Professor
raul.mazo@ensta.fr

Objectifs

1. Acquérir les concepts fondamentaux du langage Java
2. Acquérir les concepts fondamentaux de la POO
3. Apprendre à les implémenter en Java
4. Apprendre dès le départ le bon usage des concepts étudiés.

Support et matériels de cours disponibles sur l'EPI:

https://moodle.ensta-bretagne.fr/course/view.php?id=1252

Convention de Codage :

http://www.oracle.com/technetwork/java/codeconventions-150003.pdf

Specs Java d'Oracle :

http://docs.oracle.com/javase/specs/jls/se8/html/

Programme du cours

1. **Introduction et Notions de base du langage (2h cours + 2h TP)**: propriétés de Java, Java vs Python, clases, attributs, types primitifs, tableaux, méthodes, input scanner, lecture/écriture de fichiers, structures de contrôle, instructions itératives et de sélection, méthode incrémentale de conception, au-delà des bases pour écrire un bon code Java.
2. **Notions de base de la POO (2h cours + 2h TP)**: objets, classes, attributs, méthodes (constructeurs, getters/setters, logique métier), instanciation, surcharge, protection, variables et méthodes de classe vs variable et méthodes d'instance
3. **Notions avancées de la POO (2h cours + 2h TP)**: héritage, polymorphisme, typage statique (opérateur *cast*), résolution de variables, résolution de méthodes, redéfinition de méthodes, classes et méthodes abstraites, interfaces.
4. **Gestion des erreurs (2h cours + 2h TP)**: importance de la qualité des logiciels, *Java errors*, Exceptions, try-with-resources, éviter les try-catch?, bugs, JUnit et les test unitaires
5. **Interface homme-machine (2h cours + 2h TP)**: installation et introduction à JavaFx, anatomie d'une application JavaFx, éléments graphiques, définir le comportement, layout, transformations, Charts, HTML content, Media, transitions animées, pièges à éviter.
6. **Présentation d'avancement** des projets (**4h tutorat**)
7. **Présentation des projets** (**6h: 20 min par projet + 10 min questions/feedback/note**)

Présentation et organisation des TPs

Programme du cours	Contenu/Compétences	Exercices des TPs
Introduction et Notions de base du langage (2h cours + 2h TP)	Propriétés de Java	No Pain, No Gain ; Un peu de récursivité
	Classes	No Pain, No Gain ; Un peu de récursivité
	Attributs	Un peu de récursivité
	Types primitifs	No Pain, No Gain ; Un peu de récursivité ; Méthodes de tri
	Tableaux	Un peu de récursivité ; Méthodes de tri
	Méthodes	Un peu de récursivité ; Méthodes de tri
	Input Scanner	Température par ville, Guess the number
	Lecture/écriture de fichiers	Température par ville
	Structures de contrôle	Méthodes de tri ; Température par ville, Guess the number
	Instructions itératives et de sélection	Méthodes de tri ; Température par ville, Guess the number
	Méthode incrémentale de conception	Un peu de récursivité ; Méthodes de tri
Notions de base de la POO (2h cours + 2h TP)	Objets	Dépôt de voitures d'occasion ; Compte bancaire ; Taxes ; Dossiers des étudiants ; Cercles
	Classes	Dépôt de voitures d'occasion ; Compte bancaire ; Taxes ; Dossiers des étudiants ; Cercles
	Attributs	Dépôt de voitures d'occasion ; Compte bancaire ; Taxes ; Dossiers des étudiants ; Cercles
	Méthodes	Dépôt de voitures d'occasion ; Compte bancaire ; Taxes ; Dossiers des étudiants ; Cercles
	(Constructeurs, getters/setters, logique métier)	Dépôt de voitures d'occasion ; Compte bancaire ; Taxes ; Dossiers des étudiants ; Cercles
	Instanciation	Dépôt de voitures d'occasion ; Taxes ; Dossiers des étudiants ; Cercles
	Surcharge	Voir tous les exercices de la section Notions avancées de la POO
	Protection	Dépôt de voitures d'occasion ; Compte bancaire ; Taxes ; Dossiers des étudiants ; Cercles
	Variables et méthodes de classe vs variable et méthodes d'instance	Compte bancaire ; Dossiers des étudiants ; Cercles
Notions avancées de la POO (2h cours + 2h TP)	Héritage	Bibliothèque ; Polygones ; Simulation d'un mini-écosystème de cigales et fourmis ; Expressions arithmétiques ; Hiérarchie des comptes créditeurs
	Polymorphisme	Bibliothèque ; Polygones ; Simulation d'un mini-écosystème de cigales et fourmis ; Expressions arithmétiques ; Hiérarchie des comptes créditeurs
	Typage	Bibliothèque ; Polygones ; Simulation d'un mini-écosystème de cigales et fourmis ; Expressions arithmétiques
	Statique (opérateur cast)	Bibliothèque ; Polygones
	Résolution de variables	Polygones ; Simulation d'un mini-écosystème de cigales et fourmis ; Expressions arithmétiques
	Résolution de méthodes	Bibliothèque ; Polygones ; Simulation d'un mini-écosystème de cigales et fourmis ; Expressions arithmétiques ; Développer une hiérarchie des comptes créditeurs
	Redéfinition de méthodes	Bibliothèque ; Polygones ; Simulation d'un mini-écosystème de cigales et fourmis ; Expressions arithmétiques ; Hiérarchie des comptes créditeurs
	Classes et méthodes abstraites	Bibliothèque ; Expressions arithmétiques ; Développer une hiérarchie des comptes créditeurs
	Interfaces	Expressions arithmétiques ; Développer une hiérarchie des comptes créditeurs
Gestion des erreurs (2h cours + 2h TP)	Exceptions	Exceptions ; Lecture d'un fichier ; Les numéros (© Chua Hock-Chuan) ; Factorials
	try-with-resources	Exceptions ; Lecture d'un fichier ; Factorials
	JUnit et les test unitaires	Les numéros (© Chua Hock-Chuan) ; Compte Bancaire
Interface homme-machine (2h cours + 2h TP)	Eléments graphiques	Interface Graphique Simple ; Simple Transitions ; Voitures de course ; Menus ; SliderDemo
	Définir le comportement	Interface Graphique Simple ; Simple Transitions ; Voitures de course ; Menus ; SliderDemo
	Layout	Simple Transitions ; Voitures de course ; Menus ; SliderDemo
	Transformations	Simple Transitions ; Voitures de course ; SliderDemo
	Charts, HTML content, Media	----
	Transitions animées	Simple Transitions ; Voitures de course

Présentation et organisation du projet

Simulateur de Planification et de Gestion d'Actifs Mobiles (SPIGA)

Indications pour bien réussir vos projets

1. Recommandations et consignes pour la **formation des groupes et l'organisation du travail (4 étudiants par groupe)**
2. Recommandations et consignes pour la **réalisation du projet**
3. Recommandations pour la **partie rapport et rendu du travail (deadline: dimanche 4 janvier 2026 minuit)**
4. Consignes pour la **présentation et démonstration du projet (en janvier)**
5. **Checklist**

Evaluation
Note projet = Note du rendu du projet * coefficient de maitrise entre 0 et 1 de soutenance individuelle (4 étudiants)

Le coefficient de maitrise est calculé par rapport à la qualité des réponses individuelles aux questions (souvent des questions sur le contenu du cours, la POO, appliquées au projet).

À la porte d'entrée d'une université en Afrique du Sud, le message suivant a été posté pour la contemplation:

"**Détruire un pays ne nécessite pas l'utilisation de bombes atomiques ou l'utilisation de missiles à longue portée. Il suffit d'abaisser la qualité de l'éducation et de permettre la tricherie dans les examens/projets par les étudiants.**

Les patients meurent aux mains de ces médecins.

Les bâtiments s'effondrent aux mains de ces ingénieurs.

L'argent est perdu aux mains de ces économistes et comptables.

L'humanité meurt aux mains de ces savants religieux.

La justice est perdue entre les mains de ces juges ...

L'effondrement de l'éducation est l'effondrement de la nation".

Course organisation and general rules

1. The students will integrate the studied concepts into a **practical project**
2. The **practical project must be sent to the professors (one day before the presentation) and presented to the class during the last session** (scheduled for the exam). That project will be notated (100% of the grade) with regard to its quality and completeness.
3. **If you come to the lectures, you come to work!** (attendance will be taken for internal control purposes). Some professors' remarks to the students of the second year (2A): *"La programmation OO n'est pas comprise, en encore moins un réflexe (structuration du code, héritage, polymorphisme, interfaces, …)" et*
   "les étudiants ont du mal à mobiliser certaines fonctions/methodes élémentaires : lecture, écriture de fichier, extraction ligne à ligne, split, gestion des erreurs…"
4. **Difficulties with your computer, tools** (performance, configuration, bugs, licenses, …) **and working team** (work schedules, division of responsibilities, lack of trust, lack of commitment, and communication …) **are the responsibility of the student, NOT the professor**.
5. **Grades are the responsibility of the student, NOT the professor (grades and comments will be communicated at the end of each presentation)**. At the end of the semester, please avoid sending any mail of this kind:
   - "please put in an extra assignment"/"Puis-je faire qqch pour récupérer mes points perdus ?"
   - "I'm going to lose my scholarship"/"Je comptais beaucoup sur cette note"
   - "my average is going to be very low"/"Pourquoi ai-je obtenu une note aussi basse ?"

Quelques généralités et quelques éléments de comparaison avec Python

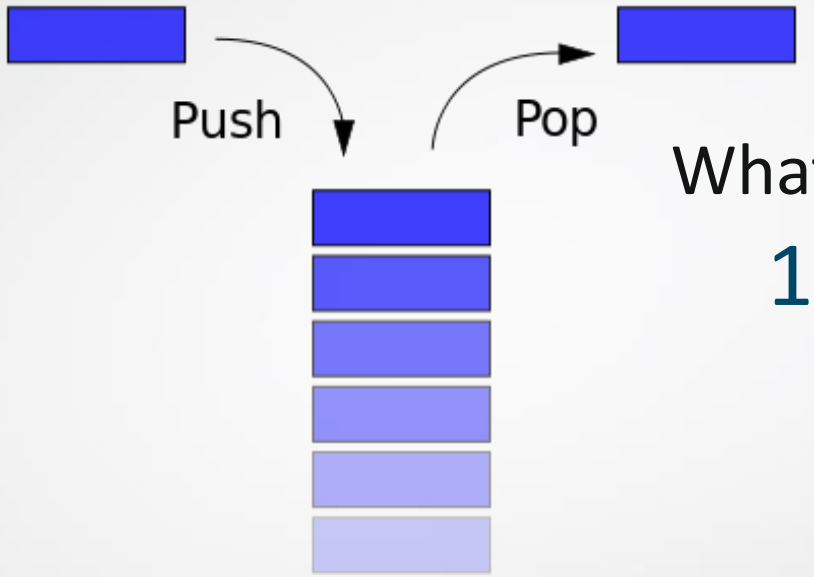
Why OOP and why Java?

```
int stack[100];
int top;

void init(){
   top = -1;
}

void push(int val){
   if (!isFull()){
      stack[++top] = val;
    }
    else error("stack is full");
 }

...
```



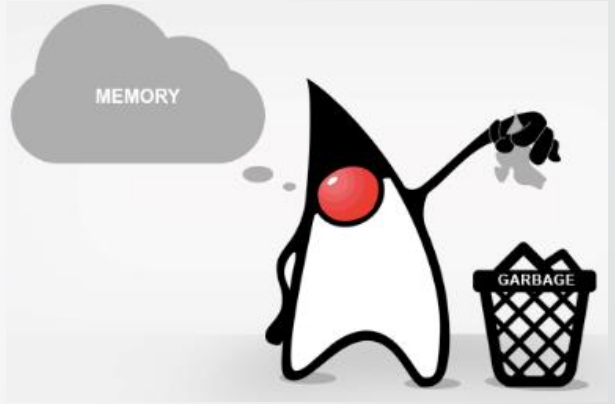
What are the problems?

1. Not generic
   - What happens if more stacks are needed?
   - What happens if a stack of a different type is needed?
2. Fixed size
3. No effective information hiding mechanism

But who uses Java?

Propriétés de Java

- **Langage OO d'usage général :** à base de classes (pas de variables globales, pas de pointeurs explicites).
- **Gestion automatique de la mémoire dynamique** : utilise un « garbage-collector » pour récupérer la mémoire inutilisée (pas de « libération » d'objet explicite).
- **Portable** : Utilisation d'une machine virtuelle (JVM) et d'un émulateur pour produire de exécutables indépendant de la machine.
- **Support du multi-tâches** (grâce aux threads)
- **Adapté à la programmation Internet**.
- **Assez sûr**: fortement typé, pas de pointeurs « fous », vérifications au chargement des classes et durant l'exécution (débordements).
- Java est **fourni avec une bibliothèque très riche**.



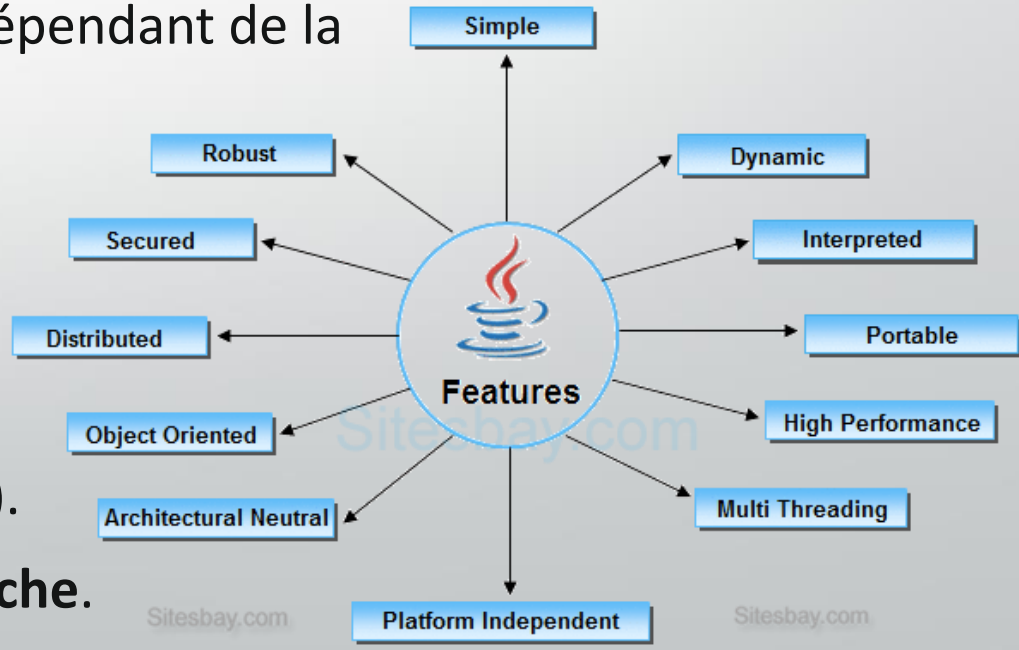
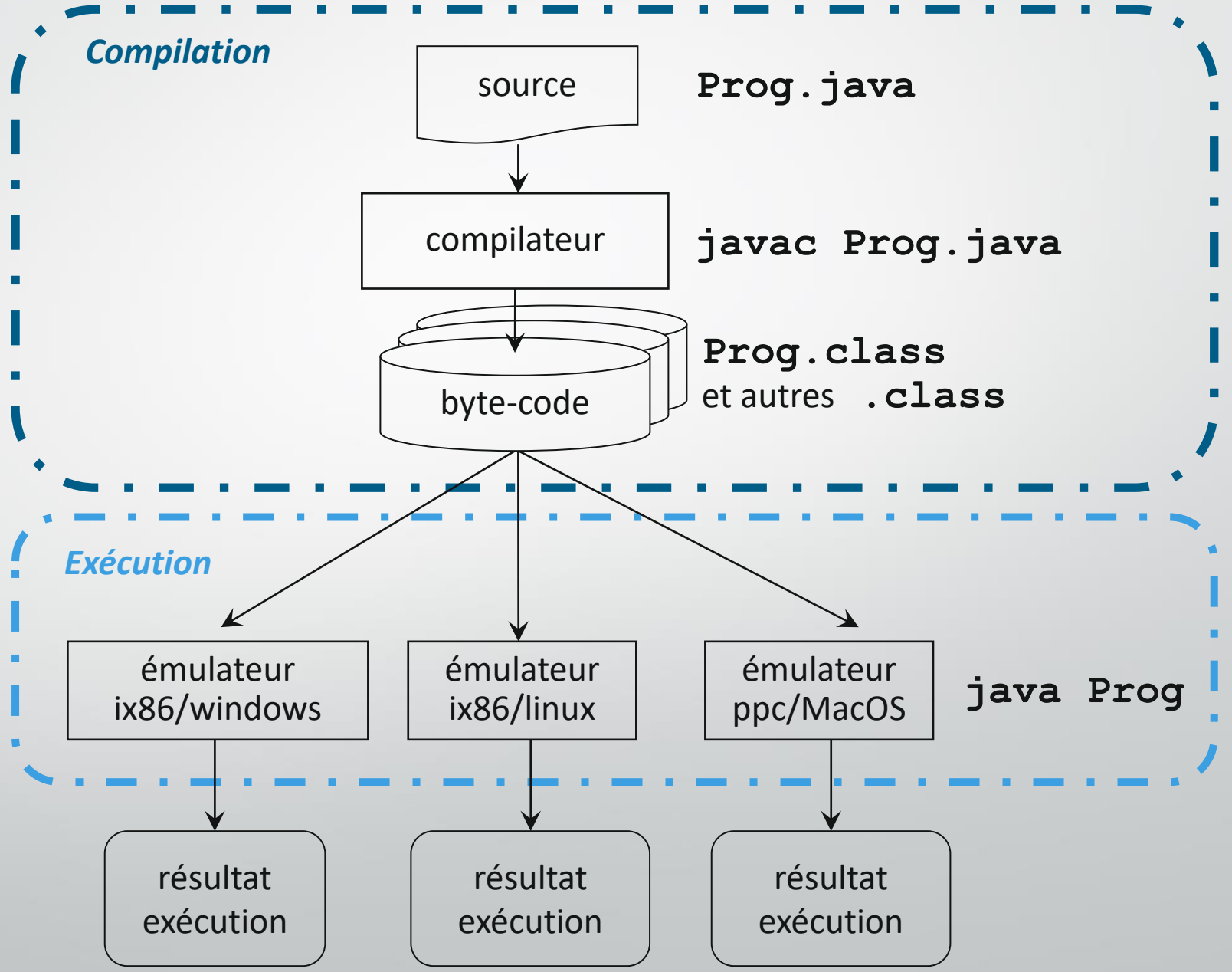


Schéma de compilation et d'exécution

Le source Java (Prog.java) est compilé et donne lieu à un **byte-code** (Prog.**class**) **pour la machine virtuelle de Java (JVM)**.

Ce byte-code ne peut être exécuté directement et nécessite un **émulateur** : un programme qui **lit le byte-code, le décode et l'exécute**.

Le **byte-code** peut être exécuté sur **n'importe quelle machine** pourvu qu'un **émulateur** soit présent.



Java vs. Python

Pour éviter tout malentendu :

Il n'y a pas de bons et de mauvais langages,

… il n'y a que de bons et de mauvais programmeurs.

Utiliser l'un ou l'autre dépend de l'utilisation (apprentissage, contexte, système critique ou pas, du but du logiciel, …)

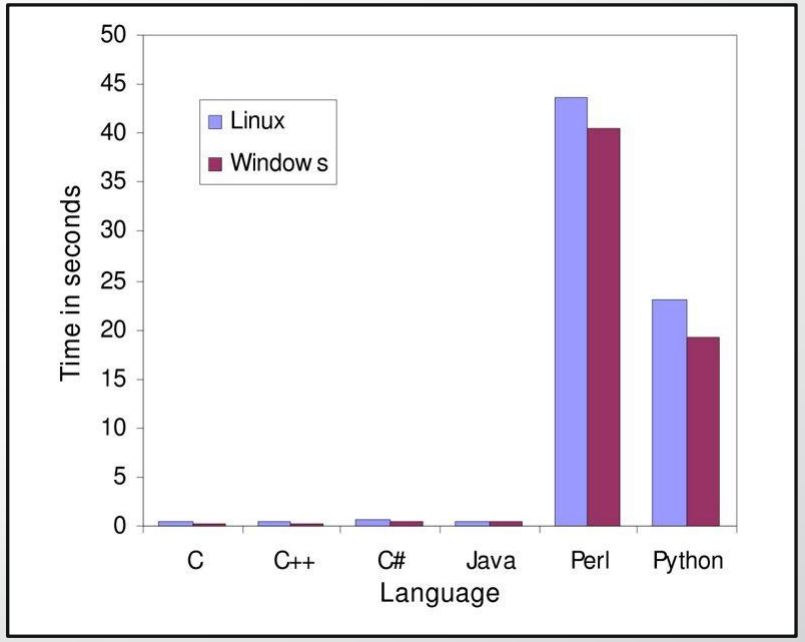
De l'artisanat

De l'ingénierie

Java vs. Python

In terms of **speed** Java is faster. Whenever in projects speed matters the Java is best. Python is slower because python is interpreted and the type of the data is determined at run time.

"**Compiled languages are faster than interpreted languages** because the code is prepared for execution and does not require an intermediate, real-time step to process the code before execution" (Bell, 2017)



Speed comparison of the global alignment algorithm implemented in C, C++, C#, Java, Perl and Python. Two DNA sequences of 3216 bp and 3217 bp were used. (Fourment & Gillings, 2008)

Java vs. Python

Compilé en byte-code

Typage :

- Statique : chaque variable connait son type à la compilation
- Une variable doit être déclarée
- (Erreur si type non défini)

- Avantage : permet une analyse à la compilation
- Désavantage : syntaxe plus contraignante

Conversion de type :

```
String str = Integer.toString(i);
```

// **commentaires** … ou /* … */

Interprété

Typage :

- Dynamique : chaque variable connait son type à l'exécution
- Le type d'une variable n'est pas à déclarer

- Avantage : "semble" plus facile à programmer
- Désavantage : analyse difficile car l'inférence de type est un problème difficile.

Conversion de type :

```
Int (x [ ,base])
```

commentaires … ou """ … """

Notions de base du langage Java

Structure générale

Syntaxe proche de celle du **C, C++**; change par rapport à **Python** car le typage est différent et en Java il y a plus de règles à respecter.

Un fichier source Java est **constitué de *classes*** (souvent 1 seule).

Une **classe comprend la définition de variables** (appelées *attributs*) **et de fonctions** (appelées *méthodes*).

À différence de Python, on ne peut pas exécuter n'importe quel fichier. En Java, une application doit contenir une méthode **`main`** définie **à l'intérieur d'une classe portant le même nom que le fichier source** (ATT: majuscules et minuscules sont différenciés en Java et doivent donc l'être dans les fichiers sources).

Déclaration de la fonction main:

```
public static void main(String[] args) {
    … corps du main …
}
```

Définition de classes et d'attributs

Une classe se définit, en Java, comme suit :

```
<protection> class <nom classe> {
   … définition d'attributs …
     … définition de méthodes …
}
```

Par convention **<nom classe>** commence par une majuscule.

La **<protection>** est facultative et sera étudiée plus tard. Pour l'instant nous utiliserons juste la protection **public** pour la classe où est définie le **main**.

Remarque: **une classe publique doit porter le même nom que le fichier source dans lequel elle est définie** (il ne peut donc y avoir au plus qu'une classe publique dans un même fichier source).

Les attributs peuvent être vus comme des variables globales pour une classe. Ils sont visibles par toutes les méthodes de la classe.

A différence de Python où tous les attributs sont publiques, en Java ils peuvent être aussi privés.

Les attributs ont la possibilité d'initialisation lors de la déclaration :

```
<type> <nom variable> [ = <expr init> ] ;
```

Un **<type>** étant

soit un ***type primitif***, ex:

```
int age;
float coeff = 1.2;
```

soit un ***type Classe***, ex:

```
String nom;
```

Types primitifs

Type	*bits*	*Plage de valeurs*	*Description*
`boolean`	1	**`true`** ou **`false`**	valeur logique
`char`	16	0 à 65535	Entier non signé sur 2 octets = Unicode
`byte`	8	-128 à +127	entier sur 1 octet
`short`	16	-32768 à +32767	entier sur 2 octets
`int`	32	- 2 147 483 648 à + 2 147 483 647	entier sur 4 octets
`long`	64	- 9 223 372 036 854 775 808 à + 9 223 372 036 854 775 807	entier sur 8 octets
`float`	32	± 1.40239846e-45 à ± 3.40282347e+38	réel sur 4 octets (simple précision)
`double`	64	± 4.9406564584124654 4e-324 à ± 1.7976931348623157 0e+308	réel sur 8 octets (double précision)

Types primitifs

Les **opérations sur les types primitifs sont plus performantes que sur les objets**.

Les types: **`byte`**, **`short`**, **`int`** et **`long`** sont des entiers signés.

Les types primitifs sont toujours passés par valeur aux méthodes. **Ils ne sont donc pas modifiables par la méthode appelée**.

In addition to the eight primitive data types listed previously, the Java programming language also provides special support for character strings via the java.lang.String class. Enclosing your character string within double quotes will automatically create a new String object; for example, String s = "this is a string";

String objects, as well as the primitives, are *immutable*, which means that once created, their values cannot be changed. The String class is not technically a primitive data type, but considering the special support given to it by the language, you'll probably tend to think of it as such.

```
String name = "Alan Turing";
name.toUpperCase();
System.out.println(name);
```

The previous code displays "Alan Turing", because the value of *name*, which refers to the original String object, never changes!

If you want to change *name* to be uppercase, then you need to assign the return value:

```
String name = "Alan Turing";
name = name.toUpperCase();
System.out.println(name);
```

Types primitifs

Primitive types like int, double, and char cannot be null, and they **do not provide methods**. For example, you can't invoke *equals* on an int:

```
int i = 5;
System.out.println(i.equals(5));  // compiler error
```

But for each primitive type, there is a corresponding **wrapper class** in the Java library. **The wrapper class for int is named Integer**, with a capital **I**:

```
Integer i = Integer.valueOf(5);
System.out.println(i.equals(5));  // displays true
```

Other wrapper classes include Boolean, Character, Double, and Long. They are in the java.lang package, so you can use them without importing them. Like strings, **objects from wrapper classes are immutable, and you have to use the equals method to compare them**:

```
Integer x = Integer.valueOf(123);
Integer y = Integer.valueOf(123);
if (x == y) {                    // false
   System.out.println("x and y are the same object");
}
if (x.equals(y)) {               // true
   System.out.println("x and y have the same value");
}
```

QUIZ QUESTIONS

What are the expected outputs of the following codes:

```
String str = "12345";
int num = Integer.parseInt(str);
```

The result is the integer number 12345 because Wrapper classes also provide methods for converting strings to and from primitive types. For example, `Integer.parseInt` converts a string to an int. In this context, **parse** means “read and translate”.

```
int num = 12345;
String str = Integer.toString(num);
```

The result is the String object "12345".

```
String str = "five";
int num = Integer.parseInt(str);
```

In this case ***`parseInt`*** throws a **`NumberFormatException`**, because the characters in the string "five" are not digits.
This shows that it’s always possible to convert a primitive value to a string, but not the other way around.

Déclaration de tableaux

Un tableau se déclare par

```
<type> [] <nom variable>;
```

Un tableau est en réalité un objet et doit donc s'initialiser grâce à l'opérateur **new** (que nous verrons plus tard) comme suit:

```
int [] t = new int [10];
```

On peut aussi initialiser un tableau:

```
int [] r = {5, 3, 6};
```

Ou encore:

```
t = new int [] {x, 3, 12, y+3};
```

Un tableau à n éléments est indicé de 0 à n-1. On peut connaître le nombre d'éléments d'un tableau par **<nom>.length**

```
int nbElem = t.length;
```

Un tableau est un objet et les objets sont toujours passés par référence. Ils sont donc *modifiables* par la méthode appelée.

Définition des méthodes

La définition des méthodes est proche du C mais différente à celle de Python :

Java

```
<protection> <type> <nom méthode> ( <arg1>, … , <argn> ) {
   … corps de la méthode…
}
```

Python

```
def function_name( args ):
   function code
```

Le **<type>** est le type renvoyé par la méthode ou **void** si elle ne renvoie rien.

Par convention le **<nom méthode>** commence par une minuscule.

<arg_i> sont les déclarations des paramètres formels. On utilise la même syntaxe que pour la déclaration d'attributs. Exemple:

```
int somme(int x, int y) {
   return x + y;
}
```

Un premier programme

HelloWorld.java

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

Pour les affichages on utilise:
`System.out.println("<mesg>");`
L'argument attendu de **`println`** est un **`String`**. Si ce n'est pas le cas il est automatiquement converti en **`String`** (si c'est un objet sa méthode **`toString()`** est appelée).

Windows

```
C:\> javac HelloWorld.java

C:\> dir
././.  09:58 109 HelloWorld.java
././.  10:03 425 HelloWorld.class

C:\> java HelloWorld
Hello World
```

UNIX

```
C:\> javac HelloWorld.java

C:\> ls
HelloWorld.java
HelloWorld.class

C:\> java HelloWorld
Hello World
```

Instruction d'affectation et expressions

On utilise comme en C :

```
<variable> = <expr>
<variable> <op> = <expr>
```

Une expression numérique est constituée de constantes numériques, de variables entières ou réelles et des opérateurs :

```
+  -  *  /  %    ++    --
```

Une expression sur les chaînes peut utiliser l'opérateur **+** pour concaténer 2 chaînes.

Une expression booléenne est constituée de constantes (**`false`**, **`true`**), de variables booléennes et des opérateurs :

```
==   !=   <   <=   >   >=
```
(comparaisons)

`&&` (ET) **`||`** (OU) **`!`** (NON)

Les expressions booléennes apparaissent aussi comme tests dans l'instruction conditionnelle (**`if`**) et les boucles (**`while`**, **`for`**, …).

Instruction conditionnelle

Syntaxe en Java:

```
if (<expr booléenne>)
    <instruction si vrai>
else
    <instruction si faux>
```

En Java, si la partie ALORS (respectivement SINON) nécessite plus d'une instruction on utilisera un bloc **{ }**

Exemple:

```
if (a > b && a > c) {
    m = a;
    u = b + c;
} else
    u = a + b + c;
```

Prog.java

```
public class Prog {
    public static void main(String[] args) {
      int var1 = 5;
      int var2 = 6;
      int var3 = 5;

      if(var1==var3){
        System.out.println("equals");
      }

      int aux=1;

      while(aux<8){
        System.out.println(aux);
        aux++;
      }
    }
}
```

```
C:\> javac Prog.java
C:\> java Prog
```

```
equals
1
2
3
4
5
6
7
```

Instructions itératives

Comme en C on dispose de 2 boucles principales qui répètent un traitement **tant que** la condition est vraie.

```
while (expression) {
    <instruction à itérer>;
}
```

Avec cela, on peut ne jamais passer dans la boucle

```
do {
    <instruction à itérer>;
} while (expression);
```

Avec un do… while, on passe au moins 1 fois dans la boucle

Prog.java

```
public class Prog {
    public static void main(String[] args){
      int n = 10;
      int sum = 0;
      int i = 1;
      while(i<=n){
        sum=sum+i;
        i++;
      }
      System.out.println("The sum is: " +
sum);
    }
}
```

```
The sum is: 55
```

Instructions itératives

La boucle **`for`** est prévue pour gérer un (ou plusieurs) compteur(s).

```
for (<instruction init>, <expression>, <instruction fin iter>) {
    <instruction à itérer>;
}
```

Elle est équivalente à la boucle **`while`** suivante:

```
<instruction init>;
while (expression) {
    <instruction à itérer>;
    <instruction fin iter>;
}
```

On exécute d'abord `<instruction init>` (peut être vide). Tant que **`<expression>`** est vrai (si vide c'est true) on exécute l'<instruction à itérer> suivie de l'`<instruction fin iter>` (peut être vide). Et on recommence. Exemple de saisie d'un tableau d'entiers:

```
Scanner sc = new Scanner(System.in);
int[] t = new int[10];
for (int i = 0; i < t.length; i++) {
   System.out.print("entrer i[" + i + "]: ");
   t[i] = sc.nextInt();
}
```

Instructions de rupture de boucle

L'instruction **`break`** sort de la boucle dans laquelle elle se trouve. Cela revient donc à sauter à l'instruction suivant la boucle.

L'instruction **`continue`** permet de passer à l'itération suivante. Dans le cas des boucles **`while`** et **`do`**…**`while`** cela revient à sauter au <test> de boucle. Pour la boucle **`for`**, on exécute <expr fin iter> et ensuite on saute au <test>.

Si on veut que **`break`** ou **`continue`** s'applique à une boucle de plus haut niveau on place un label avant la boucle voulue (pour la nommer) et on utilise:

```
int [][] t = new int [5][10];
boolean trouve = false;
sortie: // on place un label
  for (int i = 0; i < t.length; i++) {
    for (int j = 0; j < t[0].length; j++) {
      if (t[i][j] == x) {
      trouve = true;
      break sortie;
      }
    } // fin de boucle interne
  } // fin de boucle externe
```

Instruction de sélection

Cas général:

```
switch(<expr>) {
    case <expr cst>:
      <instructions>
    break;
    …
    case <expr cst>:
      <instructions>
    break;
    default:
      <instructions>
```

Att: en l'absence de **break**, après l'exécution des instructions d'un cas, on exécute les instructions du cas suivant (jusqu'à trouver un **break**).

Prog.java

```
public class Prog {
  public static void main(String[] args) {
      int switch_case = 1;
      switch (switch_case) {
          case 1:
            System.out.println("Case 1");
            // other instructions
            break;
          case 2:
            System.out.println("Case 1");
            // other instructions
            break;
          default:
            System.out.println("Case def");
            // other instructions
            break;
      }
  }
}
```

Input Scanner/Lecture au clavier

The most flexible and common way to read an input from a user is by asking them to type in something and wait till they respond.

A Scanner allows the program to read any data type from a particular input; hence, the System's input (System.in, which is the command line) and it will continue to read whatever the user is typing until they hit "enter" then the program continues to execute.

Calling the method `nextLine()` in that scanner object will return a String that contains everything the user has typed in before they hit "enter".

For example:

```
import java.util.Scanner;
…
System.out.println("Enter your address: ");
Scanner scanner = new Scanner(System.in);
String address = scanner.nextLine();
System.out.println("You live at: " + address);
```

To be able to access the Scanner class, you have to point your program to the java.util library that includes the Scanner class!

Input Scanner/Lecture au clavier

If you want to read a number into an integer variable instead of the entire line:

For example:

```
System.out.println("Enter your grade: ");
Scanner scanner = new Scanner(System.in);
int grade = scanner.nextInt();
if(grade > 15){
   System.out.println("Wow! you did well!");
}else{
   System.out.println("Not bad, but you can do better next time!");
}
```

Scanner has other methods that you can use (see Java doc for more information):

```
float  f = scanner.nextFloat();
double d = scanner.nextDouble();
String s = scanner.next();
Boolean b = scanner.nextBoolean();
```

Les fichiers à accès direct permettent un accès rapide à un enregistrement contenu dans un fichier.

```
public class FirstFile {
 public static void main(String[] args) {
  try {
   RandomAccessFile monFichier = new RandomAccessFile("file.dat", "rw");
   for (int i = 0; i < 10; i++) {
    monFichier.writeInt(i * 100);
   }
   monFichier.close();
  } catch (Exception e) {
   e.printStackTrace();
  }
 }
}
```

La classe **RandomAccessFile** possède deux constructeurs qui attendent en paramètres le **fichier à utiliser** (sous la forme d'un nom de fichier ou d'un objet de type File qui encapsule le fichier) et le **mode d'accès** («r» ou «rw» selon que le mode est lecture seule ou lecture/écriture).

```
public class FirstFile {
 public static void main(String[] args) {
  try {
   RandomAccessFile monFichier = new RandomAccessFile("file.dat", "rw");
   for (int i = 0; i < 10; i++) {
    System.out.println(monFichier.readInt());
   }
   monFichier.close();
  } catch (Exception e) {
   e.printStackTrace();
  }
 }
}
```

La classe RandomAccessFile possède de nombreuses méthodes writeXXX() et readXXX()

Les fichiers à accès direct permettent un accès rapide à un enregistrement contenu dans un fichier

```
public class FirstFile {
public static void main(String[] args) {
   try {
     RandomAccessFile monFichier = new RandomAccessFile("file.dat",
"rw");
     // 5 représente le sixième enregistement puisque le premier
commence à 0
     // 4 est la taille des données puisqu'elles sont des entiers de type
int (codé sur 4 octets)
     monFichier.seek(5*4); //measured in bytes
     System.out.println(monFichier.readInt());
     monFichier.close();
   } catch (Exception e) {
     e.printStackTrace();
   }
  }
}
```

Résultat :

1	500

1	0
2	100
3	200
4	300
5	400
6	500
7	600
8	700
9	800
10	900

Pour **naviguer dans le fichier**, la classe utilise un **pointeur qui indique la position dans le fichier où les opérations de lecture ou de mise à jour doivent être effectuées**.
La méthode **getFilePointer() permet de connaître la position de ce pointeur** et la méthode **seek((enregistrement-1) *taille_données) permet de le déplacer d'un certain nombre de bytes**.

La méthode **seek() attend en paramètre un entier long** qui représente la position, dans le fichier, précisée en octets. La première position commence à zéro.

Beyond the basics

Consider the following code samples that all **produce the same output**. Why would you write one instead of the other? What is special about the last two?

```
int[] intArray = new int[3];
intArray[0] = 1;
intArray[1] = 2;
intArray[2] = 3;

for(int i=0; i < intArray.length; i++) {
        System.out.println(intArray[i]);
}
```

Option A:
7 lines

```
var intArray = new ArrayList<Integer>();
intArray.add(1);
intArray.add(2);
intArray.add(3);
intArray.stream().forEachOrdered((i) -> System.out.println(i));
```

Option B:
5 lines

```
IntStream.rangeClosed(1, 3).forEachOrdered((i) -> System.out.println(i));
```

Option C:
1 line

Beyond the basics

Consider options A and B…

When types are unambiguous, like creating a new instance of a class, we can use ***var*** instead of a type declaration

```
int[] intArray = new int[3];
intArray[0] = 1;
intArray[1] = 2;
intArray[2] = 3;

for(int i=0; i < intArray.length; i++) {
        System.out.println(intArray[i]);
}
```

Unsafe element access

Possibly unsafe iteration over array

We can use more abstract types to make it easier and safer to operate over data structure

```
var intArray = new ArrayList<Integer>();
intArray.add(1);
intArray.add(2);
intArray.add(3);
intArray.stream().forEachOrdered((i) -> System.out.println(i));
```

Safer element access w/ obj methods

Doing a precise operation over each element of an abstract representation of the collection

Beyond the basics

Consider options A and B… Focus on the final part of each option, the **for** loop

Iteration over the low-level array

```
int[] intArray = new int[3];
intArray[0] = 1;
intArray[1] = 2;
intArray[2] = 3;

for(int i=0; i < intArray.length; i++) {
        System.out.println(intArray[i]);
}
```

Element access is done with low-level constructs

These constructs require some logic that we cannot screw up (termination of iteration and element access index)

We utilize an abstract ordered collection named ***stream*** to perform operations over the elements of a list-like data-structure.

```
var intArray = new ArrayList<Integer>();
intArray.add(1);
intArray.add(2);
intArray.add(3);
intArray.stream().forEachOrdered((i) -> System.out.println(i));
```

We now use a function (***forEachOrdered***) that abstracts away the details of the iteration and simply does some operation on each individual element that we give to it in the form of a function of a single parameter in this case.

Beyond the basics

So what exactly are we doing in option B?

```
forEachOrdered((i) -> System.out.println(i));
```

The function **forEachOrdered** is a function defined for ***streams*** that takes as an argument some **anonymous function** and ***applies it to every element of the stream in the original order***. This holds even if the computation is done in parallel.

```
forEachOrdered((i) -> System.out.println(i));
```

The **anonymous function** passed as the one and only parameter of **forEachOrdered** will be executed with each of the **elements** that are in the collection. In this case, the variable ***i*** is the element of the collection that will be operated upon. Here **type declarations are not necessary** in the parameter definition. The arrow **"->"** separates the arguments of the function on the left, with the body of the function on the right. The body of the function determines the operations that will be performed on the arguments given. In this case a single operation is performed with a void return type; thus, this function is implicitly void. If the operation had a return type other than void, then the result of the evaluation of either a single instruction, or multi-line block (with opening and closing backets **"{}" and an explicit return statement**) would yield the result of the anonymous function for each element.

Beyond the basics

Now let's see what changes from **B** to **C**.

```
var intArray = new ArrayList<Integer>();
intArray.add(1);
intArray.add(2);
intArray.add(3);
intArray.stream().forEachOrdered((i) -> System.out.println(i));
```

Since we know that we want an ordered collection of three integers, we can use the **built-in** class ***IntStream*** which represents an abstract sequence of integer-valued elements in lieu of explicitly creating a variable that will hold an array and then manually filling the array.

To obtain the stream over which we will operate, instead of transforming the variable of type **ArrayList<Integer>** into a ***stream with the stream() method*** of the **ArrayList<Integer>,** we use the **rangeClosed()** *static method* of the **IntStream** class to generate an **IntStream** instance having precisely the elements comprised in range between the first and second parameter of **rangeClosed()**.

```
IntStream.rangeClosed(1, 3).forEachOrdered((i) -> System.out.println(i));
```

The ***forEachOrdered*** call remains the same on **both**.

Beyond the basics

As Java has evolved, it has incorporated into both its syntax and standard library **many useful features** that are common in other programming languages, even from those coming from other programming paradigms.

Though the core concepts of Java will remain the same, **it is up to you** to **seek out these advances and new features that will make you both more productive and more knowledgeable about the language** in which you program.

It is highly advisable to go beyond the basics of the language and to investigate the changes made in versions 8, 9 and 10 of the language, where you will find all that was just presented and more…