



## GBIN6U03

### Introduction au langage java

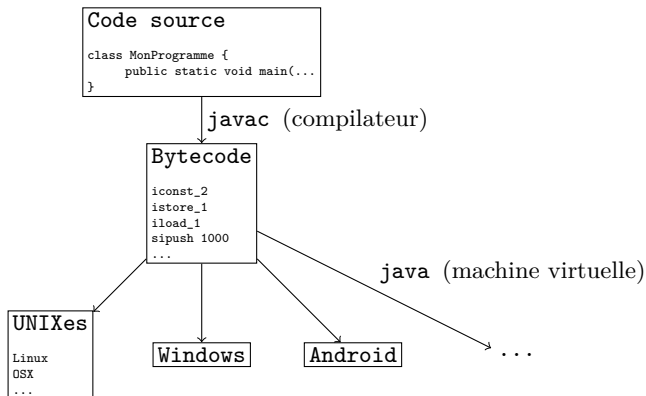
Présentation très incomplète du langage mettant l'accent sur les notions importantes au sein de l'UE via l'utilisation d'exemples

Elle s'adresse à un public connaissant le langage C

Certaines notions sont présentées par analogie avec ce langage

# Origines

Créé par James Gosling and Bill Joy dans les années 90  
Evolution d'Oak (systèmes embarqués) pour l'internet et le web



# Caractéristiques

## Moderne

- ▶ orienté objet
- ▶ exceptions
- ▶ généricité
- ▶ réflexivité
- ▶ threads

## Simple (trop ?)

- ▶ pas de pointeurs (ni arithmétique, ni adresses, *garbage collector*)
- ▶ pas d'héritage multiple
- ▶ pas de surcharge d'opérateurs
- ▶ un seul schéma d'allocation mémoire

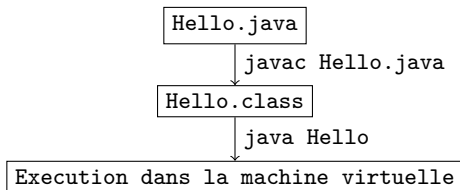
## Équilibré

- ▶ "efficace" (compilation *just-in-time* adaptative)
- ▶ "robuste et sûr" (machine virtuelle et politiques de sécurité)
- ▶ "portable" (bytecode interprété, indépendant de l'architecture)

# Hello world

Fichier Hello.java contenant

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



# Notion de classe

Classe : modèle permettant de créer des objets, regroupe des définitions

- ▶ d'attributs : données stockées dans les objets de la classe
- ▶ de méthodes : opérations applicables sur les objets de la classe

En java, une classe

- ▶ est associée à un type de même nom désignant une référence ( $\approx$  pointeur) à un objet de cette classe
- ▶ peut être instanciée en un objet à l'aide de l'opérateur `new` qui renvoie une référence à l'objet ainsi créé

Chaque objet

- ▶ contient ses propres attributs selon le modèle indiqué dans sa classe
- ▶ est manipulé par une référence à l'aide de laquelle on peut appeler les méthodes de la classe

Remarque : un objet n'a pas besoin d'être détruit (*garbage collector*)

# Par rapport aux langages non orienté objet

On retrouve des notions comparables

- ▶ les attributs correspondent aux différents champs d'une définition de type structurée
- ▶ les méthodes sont similaires à des fonctions ayant un argument implicite, leur objet

De petites différences justifient les termes différents

- ▶ les attributs ne sont pas des variables, ils sont forcément attachés à un objet
- ▶ les méthode ont un argument implicite, l'objet à partir duquel elles sont appelées
- ▶ les références, à la différence des pointeurs, n'ont pas d'arithmétique ni de valeur absolue manipulable

# Un exemple de pile en C (parenthèse code)

```
typedef struct donnees_pile *pile;

// Constructeur
pile creer_pile_vide();

// Opérateurs
void empiler(pile p, int element);
int depiler(pile p);
int est_pile_vide(pile p);

// Destructeur
void detruire_pile(pile p);
```

# Un exemple de pile en Java (fin de parenthèse)

```
class Maillon {
    int element;
    Maillon suivant;

    Maillon(int e, Maillon s) { ...
}

class Pile {
    Maillon sommet;

    // Constructeur (vide par défaut)

    // Operateurs
    void empiler(int element) { ...
    int depiler() { ...
    boolean est_pile_vide() { ...

    // Pas de destructeur
}
```



# A propos des objets

## Construction, new

- ▶ alloue la mémoire nécessaire au stockage de l'objet (attributs, ...)
- ▶ appelle, sur l'objet, avec ses arguments, un constructeur de la classe
- ▶ est la seule manière de créer un objet (pas d'allocation statique)

## Objet sur lequel une méthode est appelée

- ▶ implicite lors de l'utilisation du nom de ses attributs
- ▶ peut être explicité avec `this`

## Exemple

```
class Maillon {  
    int element;  
    Maillon suivant;  
  
    Maillon(int element, Maillon suivant) {  
        this.element = element;  
        this.suivant = suivant;  
    }  
}
```

# Classes et système de fichiers

En java, un fichier de nom `Toto.java`

- ▶ doit contenir une classe de nom `Toto`
- ▶ cette classe est visible depuis du code situé dans d'autres fichiers
- ▶ les autres classes ne sont visibles que dans le code de `Toto.java`

En compilant, quand `javac` rencontre un nom de classe, `Tata`

- ▶ il cherche un fichier `Tata.class` contenant son bytecode
- ▶ s'il ne le trouve pas, il cherche un fichier `Tata.java` qu'il compile
- ▶ s'il ne trouve pas ce second fichier il affiche une erreur

≠ `make`

Remarque : la variable d'environnement `CLASSPATH` indique les répertoires où chercher des `.class` et des `.java`

# Tableaux en java

```
class Pascal {
    public static void main(String [] args) {
        // Reference a un tableau
        int [] T;

        // Un tableau est un objet, allocation dynamique
        T = new int[Integer.parseInt(args[0])];

        // Attribut length
        for (int i=0; i<T.length; i++) {
            for(int j=i; j>=0; j--)
                if ((j == 0) || (j==i))
                    T[j] = 1;
            else
                T[j] = T[j] + T[j-1];
        }
        // Methodes utiles dans java.util.Arrays
        System.out.println(
            java.util.Arrays.toString(T));
    }
}
```

# Vérification des bornes d'un tableau

Java vérifie les indices d'accès aux tableaux

- ▶ léger impact sur la performance
- ▶ permet une bien meilleure détection de bugs

Une sortie de l'intervalle d'indices prévu provoque une erreur (exception)

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0  
at Pascal.main(Pascal.java:7)
```

# Exceptions

La gestion des erreurs

- ▶ gonfle le volume d'un code et le rend moins lisible
- ▶ est souvent négligée par les programmeurs
- ▶ est encore plus lourde lorsqu'on propage l'erreur

Un exemple en C :

```
t = malloc(n*sizeof(int));  
if (t == NULL) {  
    // Choix fixe entre traitement et propagation  
    fprintf(stderr, "Erreur d'allocation\n");  
    exit(1);  
}
```

Les exception donnent une solution à ces problèmes

# Attraper des exceptions

```
import java.io.*;
class ExceptionFichier {
    public static void main(String [] args) {
        PrintStream p = null;

        // Code fonctionnel (interrompu en cas d'erreur)
        try {
            File f = new File(args[0]);
            p = new PrintStream(f);
            p.println("Hello world");
        } catch (FileNotFoundException e) {
            System.err.println("Erreur d'ouverture");
        } catch (Exception e) {
            e.printStackTrace(System.err);
        } finally {
            // Nettoyage (toujours exécuté)
            if (p != null) p.close();
        }
    }
}
```

# Java et les exceptions

Les instructions peuvent lever des exceptions dérivées de la classe

- ▶ `Exception`, et doivent être dans un bloc `try/catch` (vérifiées)
- ▶ `RuntimeException`, sous-classe non vérifiée d'`Exception`

Dans un bloc `try / catch / finally`

- ▶ une exception interrompt le bloc `try` qui ne se termine jamais
- ▶ dans ce cas, les blocs `catch` sont consultés dans l'ordre jusqu'à trouver un type correspondant
- ▶ le bloc `finally` est toujours exécuté

Les exceptions se propagent en remontant dans la pile d'appels

- ▶ le programmeur peut attraper une exception au niveau qu'il souhaite
- ▶ permet de laisser remonter une erreur sans écrire de code particulier

# Interface

Contrat passé par une classe

- ▶ ensemble de méthodes que la classe doit définir
- ▶ type de référence associé permettant de manipuler les objets qui l'implémentent

Exemple

```
interface Pile {  
    void empiler(int element);  
    int depiler();  
    boolean est_pile_vide();  
}
```

pas de limite au nombre d'interfaces qu'une classe implémente



# Sur notre exemple de pile

```
class PileListe implements Pile {
    Maillon sommet;

    // Les méthodes implémentant l'interface
    // doivent être publiques
    public void empiler(int element) {
        Maillon m;
        m = new Maillon(element, sommet);
        sommet = m;
    }
    public int depiler() {
        int resultat;
        // Exception si sommet == null (pile vide)
        resultat = sommet.element;
        sommet = sommet.suivant;
        return resultat;
    }
    public boolean est_pile_vide() {
        return sommet == null;
    }
}
```

# Avec une autre implémentation

```
class PileTableau implements Pile {
    int [] elements;
    int sommet;

    PileTableau() {
        elements = new int[10];
        sommet = 0;
    }
    public void empiler(int element) {
        // Exception si sommet >= elements.length
        elements[sommet++] = element;
    }
    public int depiler() {
        // Exception si sommet <= 0
        return elements[--sommet];
    }
    public boolean est_pile_vide() {
        return sommet == 0;
    }
}
```

# Utilisation

```
class EssaiPile {  
    public static void main(String [] args) {  
        Pile p;  
  
        p = new PileListe();  
        // ou alors  
        p = new PileTableau();  
        p.empiler(362);  
        p.empiler(42);  
        System.out.println(p.depiler());  
    }  
}
```