Introduction programmation Java

Cours 1

Stéphane Airiau

Université Paris-Dauphine

Organisation du cours

Contrôle des connaissances

- petit projet 30%
- examen final : 70%

Les slides et les notes de cours seront postés à cette page

`http://www.lamsade.dauphine.fr/~airiau/Teaching/L3-Java/`

Programmation orientée Objet en Java

Objets et Classes

Un **objet** se définit par ses états (on peut aussi parler de ses caractéristiques) et son comportement.

exemple d'un *objet* voiture

Etats	Comportements
marque	accélerer
modèle	passer rapport supérieur
cylindrée	passer rapport inférieur
quantité d'essence	tourner volant
niveau d'huile	ouvrir porte
pression des pneus	fermer porte
nombre de tours	freiner

Une **classe** est un *plan* ou un *moule* pour fabriquer des objets.

- les *états* d'un objet vont être représentés par des *variables*
- les *comportements* d'un objet seront representés par des *méthodes*.

Un **objet** est une **instance** d'une classe.

Autrement dit

Une **classe** est un *type abstrait* caractérisé par des propriétés (attributs et méthodes) communes à un ensemble d'objets et permettant de créer des objets ayant ces propriétés.

Un **objet** ou une **instance de classe** possède un comportement et un état qui ne peut être modifié que par les actions du comportement.

On peut créer une classe `Personnage` car tous les personnages partagent les mêmes caractéristiques. Lorsqu'on veut créer un personnage, on instancie la classe `Personnage`.

IMPORTANT par **convention**,

- le nom d'une classe commence toujours par une **majuscule**.
- ce qui **n'**est **pas** le nom d'une classe commence par une minuscule

Des classes déjà existentes

`Java` possède une large librairie de classes. La librairie se compose de différents `packages` et sous-`packages`.

`http://docs.oracle.com/javase/8/docs/api/overview-summary.html`

(dans un prochain cours nous reviendrons sur le concept de package)

Par exemple, le `package java.lang` contient des classes de base du langage `Java`. On y trouve aussi une classe pour manipuler les chaînes de caractères appelée `String`.

Attributs et variables de classe

- **variables d'instance** ou **attributs** :
  ces variables définissent les caractéristiques de l'objet.
  - initialisation optionnelle.

  accès : `<nom objet>.<nom attribut>`

  ex : `monVelo.couleur`
- **variables de classe** : ces variables sont communes à *toutes* les instances de la classe,
  - déclaration avec le mot clé `static`
  - initialisation obligatoire

  accès : `<nom de classe>.<nom variable de classe>`

  ex : `Float.MAX_VALUE`

exemple : classe `Float` pour encapsuler un nombre flottant `float`.

- variables de classes : `MAX_VALUE`, `MAX_EXPONENT`, `NaN`, etc.

Méthodes d'instance et méthodes de classe

- **méthode d'instance** : ces méthodes permettent de *modifier* ou d'*accéder* à l'*état* de l'objet.
- **méthode de classe** : ces méthodes *ne* modifient *pas* l'état interne d'un objet.

exemple : la classe `Float`

- méthode d'instance `String toString()`
  ⇨ retourne une représentation en chaîne de caractères de l'*objet courant*
- méthode de classe `static String toString(Float f)`
  ⇨ retourne une représentation en chaîne de caractères du float passé en paramètre

```
1 Float f;
2 ...
3 System.out.println(f.toString());
4 System.out.println(Float.toString(3.1419));
```

Encapsulation

Les comportements et les états d'un objets peuvent être

- *connus de tous* ⇨ `public`
  toute classe peut
  - exécuter la méthode publique
  - modifier ou accéder à un attribut publique
- *cachés* ⇨ `private` la méthode ou l'attribut ne peut être accédé que depuis l'*intérieur* de la classe
  - ⇨ cacher un méchanisme interne
    (on pourra changer une implémentation sans que cela ait un impact sur la partie publique).
  - ⇨ protection

Constructeurs

Une classe est un *plan* ou un *moule* pour fabriquer un objet, ce qu'on appelle *instancier un objet*.
Les méthodes pour instancier un objet sont appelées des **constructeurs**.
Un constructeur :

- porte le nom de la classe
- n'a pas de type de retour.

On appelle constructeur **par défaut** le constructeur **sans** arguments :

```
1  public class <nom classe> {
2     // déclaration des variable d'instances et
3     // variables de classe
⋮     ...
5     // constructeur par défaut
6     public <nom classe>(){
7       // corps de la méthode
8     }
9  }
```

Exemple

La *surcharge* permet d'avoir des constructeurs avec des signatures différentes.
Pour une classe `Personnage`, on peut donc écrire :

```
1  public class Personnage {
2     public String nom;
3
4     // constructeur par défaut
5     public Personnage(){
6       nom = "Inconnu";
7     }
8
9     public Personnage(String name){
10      nom = name;
11    }
12 }
```

Création d'un objet

- Déclaration : comme pour les types primitifs :
  `<Nom de la classe> <nom objet>;`
- Création/initialisation à l'aide du mot clé **new** et appel du constructeur :
  new `<Nom de classe>(<liste d'arguments>);.`
- comme pour les types primitifs, on peut déclarer et initialiser plusieurs objets du même type en même temps.

```
1 Personnage asterix = new Personnage("Astérix");
2 Personnage obelix = new Personnage("Obelix"),
3    idefix = new Personnage(”Idéfix”),
4    romain = new Personnage();
```

Egalité

```
1  Personnage asterix = new Personnage("Astérix");
2  Personnage asterixBis = asterix;
3  Personnage asterixTer = new Personnage("Astérix");
4  if (asterix == asterixBis)
5     System.out.println("Bleu");
6  else
7     System.out.println("Rouge");
8  if (asterix == asterixTer)
9     System.out.println("Bleu");
10 else
11    System.out.println("Rouge");
```

Qu'est-ce qui est affiché sur la console ?

Egalité

```
1  Personnage asterix = new Personnage("Astérix");
2  Personnage asterixBis = asterix;
3  Personnage asterixTer = new Personnage("Astérix");
4  if (asterix == asterixBis)
5    System.out.println("Bleu");
6  else
7    System.out.println("Rouge");
8  if (asterix == asterixTer)
9    System.out.println("Bleu");
10 else
11   System.out.println("Rouge");
```

Qu'est-ce qui est affiché sur la console ?

- la variable est une *référence* vers l'objet
  et **non** l'objet lui même
- == désigne l'égalité de la référence :
  deux variables peuvent *désigner* le même objet
- pour tester l'égalité entre les **propriétés** de l'objet,
  on utilise la méthode `boolean equals(Object o)`.

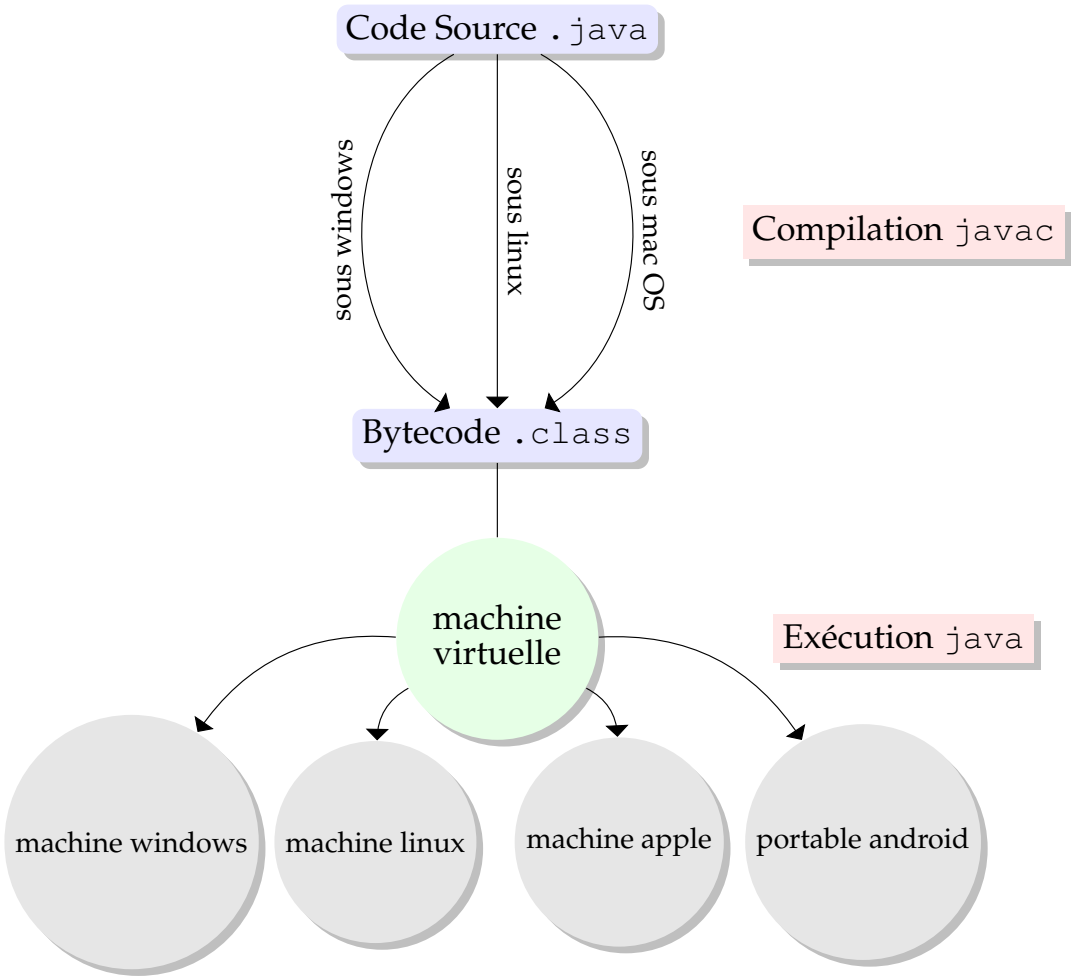
Destruction d'un objet

La destruction des objets est prise en charge par `Java` à l'aide d'un "garbage collector" (GC).

- ⇨ Le GC détruit les objets (i.e. efface la mémoire) qui ne sont référencés par aucun autre objet.
- Les destructions sont *asynchrones* et il n'y a *pas de garanties* que les objets soient détruits.
- Une méthode optionnelle nommée `finalize()` est appelée lorsque l'objet est détruit.
  Elle peut par exemple s'assurer que des fichiers ou des connexions sont bien fermées avant la destruction de l'objet.

Compilation, exécution, machine virtuelle

`Java` n'est pas seulement la description d'un langage et une bibliothèque de classe.
`Java` dispose d'outils pour *générer* et *exécuter* du code.



Compilation

Chaque classe `<MaClasse>` est enregistrée dans un fichier `<MaClasse>.java` : il porte le même nom que la classe et possède l'extension `.java`.

Le développeur doit *compiler* l'ensemble de classes à l'aide d'un programme appelé `javac`.

Le compilateur *traduit* le code écrit par le développeur en un langage plus simple qui pourra être exécuté.

Pour `Java`, le compilateur produit un code dans le langage `bytecode`.

Le résultat de la compilation est un fichie nommé `<MaClasse>.class`

Compilation

Pour simplifier, il y a deux étapes lors de la compilation :

- *analyse syntaxique* : le code est lu, on forme un arbre de syntaxique. on vérifie la syntaxe du code
  (i.e. on vérifie la grammaire du code).
- *analyse sémantique* : l'arbre syntaxique est analysé et traduit en `bytecode`.
  les références à des classes extérieures sont vérifiées (on cherche si la classe existe bien, si elle a besoin d'être compilée, etc).

Exécution

On exécute une méthode spéciale appelée **`main`**.
Cette méthode se trouvera dans *une classe* de votre choix.
Si la méthode `main` se trouve dans une classe `MaClasse`, on lancera l'exécution en lançant l'application `Java` : `~$ java maClasse`
(on peut taper cette commande sous linux ou mac os)

La méthode `main` a une signature fixée *par convention*

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

- `public` : pour être appelée de l'extérieur de la classe
- `static` : pour être appelée sans avoir instancié un objet
- `void` : la méthode ne retourne rien
- `String[] args` : lors du lancement de l'exécution, on peut ajouter du texte et chaque mot sera inséré dans un tableau de `String` qui peut servir à paramétrer l'exécution.
  `~$ java maClasse et je raconte une histoire`
  Le tableau `args` contiendra alors `{"et", "je", "raconte", "une", "histoire"}`

Machine virtuelle

`Java` fournit une **machine virtuelle** : c'est un programme qui lit du code en `bytecode` et interprète ce code dans le langage de la machine pour l'exécuter : Lors de l'exécution `java MaClass`

- on lance un programme appelée `java`
- se programme interprète le code lancé à partir de la méthode `main` de la classe `MaClasse`

Machine Virtuelle

avantages :

- le code est *portable* :
  - ⇨ On peut écrire, compiler et exécuter sur des machines d'architectures différentes. (ordinateur windows, apple, linux, téléphone mobile, caisse enregistreuse, etc).
- la machine virtuelle permet de partager d'une manière sécurisée une machine
- le code est généralement plus compact (pas besoin d'inclure les bibliothèques comme en `C` ou `C++`).
- la machine virtuelle donne l'impression que l'on dispose d'une machine entière (la machine réelle donner du temps processeur à la machine virtuelle).

inconvenients : coût en ressources de la machine virtuelle.