Programmation Orienté Objets en JAVA

Daniele Varacca
Departement d'Informatique
Université Paris Est

2014

UPEC
UNIVERSITÉ PARIS-EST CRÉTEIL VAL DE MARNE
Connaissance - Action

Retour sur les classes

- Une classe contient des champs et des méthodes
- Les champs et les méthodes peuvent être public ou private
- Une classe peux aussi avoir des champs et des méthodes static
- Champs et méthodes non-statiques s'appellent aussi *d'instance*

UPEC
UNIVERSITÉ PARIS-EST CRÉTEIL VAL DE MARNE
Connaissance - Action

Retour sur les classes

- Toute classe a une seule super-classe (sauf Object)
- Si on ne déclare pas une super-classe avec extends alors la super-classe c'est Object
- Toute classe peut implémenter plusieurs interfaces avec le mot clé implements
- Une classe abstraite (mot clé abstract) peux contenir des méthodes abstraites (sans corps)
- Chaque classe a au moins un constructeur

UPEC
UNIVERSITÉ PARIS-EST CRÉTEIL VAL DE MARNE
Connaissance - Action

Chargement des classes

Quand la JVM a besoin d'une classe, elle est *chargée*

- au démarrage, toute une série de classes standard
- quand on crée un objet
- quand on utilise ses champs ou ses méthodes statiques

Quand la classe est chargée, de la mémoire est allouée pour stocker

- champs statiques
- méthodes statiques et d'instance

UPEC
Connaissance - Action
UNIVERSITÉ PARIS-EST CRÉTEIL VAL DE MARNE

Création des objets

- Chaque objet a besoin de mémoire pour stocker les champs d'instance
- Cette mémoire est alloué dynamiquement quand l'objet est créé
- La mémoire pour les champs statiques est allouée quand la classe est chargée
- La mémoire qui contient le code des méthodes est alloué aussi quand la classe est chargée
- Mais les méthodes d'instance ont accès aux champs d'instance alloués dynamiquement

UPEC
Connaissance - Action
UNIVERSITÉ PARIS-EST CRÉTEIL VAL DE MARNE

Construction des objets

Quand un constructeur d'une classe est appelé avec new

- de la mémoire est allouée pour contenir les champs d'instance (références et valeurs primitives) avec des valeurs par défaut (0 pour int, null pour les références, etc etc)
- la super-classe est initialisée
- le code du constructeur de la super-classe est exécuté
- les initialisations déclarés dans la classe sont faites
- le code du constructeur est exécuté

(l'ordre est important au cas où une exception est lancée)

La classe String

La classe String représente une chaîne de caractères.
Quelques méthodes d'instance:

- `String toUpperCase();`
- `String substring(int begin, int end);`
- `String substring(int begin);`
- `char charAt(int index);`

La classe String

Les méthodes doivent renvoyer un résultat, il ne peuvent pas changer l'objet, car les objets de la classe String sont *immutables*

`String toUpperCase()`, renvoie un *nouvel* objet

Concaténation: s1 + s2 c'est aussi un nouvel objet
Problème:

- chaque concaténation crée un nouvel objet
- la création d'objets prends du temps (et de la mémoire)
- plusieurs concaténations c'est pas efficace

UPEC Connaissance - Action | UNIVERSITÉ PARIS-EST CRÉTEIL VAL DE MARNE

La classe StringBuilder

La classe StringBuilder: ses objets sont mutables

```
StringBuilder append(String str);
StringBuilder insert(int offset, String str);
StringBuilder replace(int start, int end, String str);
```

Ces méthodes modifient l'objet this : aucune nouvel objet est créé. L'objet renvoyé est une nouvelle référence à l'objet this

Pourquoi les méthodes renvoient si l'objet est mutable?

sb.append("One␣").append("and␣").append("only!")

UPEC UNIVERSITÉ PARIS-EST CRÉTEIL VAL DE MARNE Connaissance - Action

La classe String

Les expressions constantes: Quand on écrit

```
String s = "Hello World";
```

on crée un objet String (comme si on faisait new). Mais si on le fait une deuxième fois, c'est le même objet.

```
String s1 = "Hello World";
String s2 = "Hello World";
String s3 = new String("Hello World");
String s4 = new String("Hello World");
s1==s2 //c'est true!
s3==s4 //c'est false!
```

Morale: ne pas utiliser == avec String!

UPEC Connaissance-Action UNIVERSITÉ PARIS-EST CRÉTEIL VAL DE MARNE

Les Enveloppes

Problème avec le types primitifs: on ne peut pas faire List<int>, ou Set<boolean>: le paramètre doit être un type référence!

Types enveloppe:Boolean for boolean,Double for double, Integer for int , Character for char, etc

Ce classes

- contiennent des constructeurs à partir des types primitifs: new Integer(5); new Double(3.14);
- contiennent beaucoup de méthodes statiques intéressantes

La classe Runtime

Un objet de cette classe représente l'environnement d'exécution

- elle n'a pas de constructeur publique
- il faut utiliser la méthode statique Runtime.getRuntime();
- (elle ne crée pas un nouveau objet, mais elle renvoie un objet qui existe dans la JVM au démarrage)
- l'objet donne accès au système

Exemples de méthodes:

- `Process exec(String[] cmdarray);`
- `void gc();`

UPEC Connaissance-Action
UNIVERSITÉ PARIS-EST CRÉTEIL VAL DE MARNE

La classe Class

Un objet de la classe Class représente une classe chargée par la machine virtuelle. Cet objet est crée par la JVM quand la classe est chargée

- On récupère cet objet en invoquant la méthode statique
  `Class<?> forName(String className)`
- dans la class Object on a une méthode getClass()
- un appel obj.getClass() renvoie l'objet de la classe Class qui correspond à la classe de l'objet obj
- (c'est pas un vire-langue)
- "The class Object is represented by an object of the class Class"

UPEC
UNIVERSITÉ PARIS-EST CRÉTEIL VAL DE MARNE
Connaissance - Action

La classe Class

On peut récupérer toutes les informations de la classe:

- `Field[] getDeclaredFields();`
- `Method[] getDeclaredMethods()`

Exemple:

La méthode equals

La metode

- `boolean equals(Object other);`

est définie dans la classe Object.

Par défaut elle compare si deux références pointent vers le même objet (même comportement que ==)

On est censé la redéfinir dans les sous-classes

La méthode equals

Le contrat que la redéfinition est censé respecter est:

- elle est réflexive: x.equals(x) doit renvoyer true si x n'est pas null
- elle est symétrique: si x et y ne sont pas null x.equals(y) doit renvoyer le même que y.equals(x)
- elle est transitive: si x, y z ne sont pas null si x.equals(y) et y.equals(z) renvoient true, alors x.equals(z) doit renvoyer true.
- elle est consistante: si x et y ne sont pas null, deux appels à x.equals(y) doivent renvoyer le même résultat si entre temps les objets ne sont pas modifiés. De plus si x n'est pas null x.equals(null) doit renvoyer false.

UPEC
UNIVERSITÉ PARIS-EST CRÉTEIL VAL DE MARNE
Connaissance - Action

La méthode equals

```
class Point{
  private int x;
  private int y;
  @Override
  public boolean equals (Object other) {
    if (!(other instanceof Point)) {
        return false ;
    } else {
     Point otherPoint = (Point) other;
     return (
           this.x==otherPoint.x &&
           this.y==otherPoint.y
               );
   }
}
```

La méthode equals

```
class ColoredPoint extends Point{
  private int color;
  @Override
  public boolean equals (Object other) {
   if (!(other instanceof ColoredPoint)) {
          return false ;
   } else {
   ColoredPoint otherPoint = (ColoredPoint) other;
     return (
           super.equals(otherPoint) &&
           this.color==otherPoint.color
                );
  }
}
```

La méthode equals

Le méthode n'est pas symétrique!

```
Point p = new Point(4,5);
ColoredPoint cp = new ColoredPoint(4,5,21);
p.equals(cp); //c'est true
cp.equals(p)); // c'est false
```

D'ailleurs ce n'est pas logique qu'un objet Point puisse être égal à un objet ColoredPoint
Solution: instanceof n'est pas assez précis: il faut déterminer la classe exacte

La méthode equals

```
//Dans Point
  if (other== null ||
       this.getClass()!=other.getClass()) {
          return false ;
  } else {
  Point otherPoint = (Point) other;
     return (
            this.x==otherPoint.x &&
            this.y==otherPoint.y
                );
  }
```

La méthode equals

```
//Dans ColoredPoint
  if (other== null ||
      this.getClass()!=other.getClass()) {
          return false ;
  } else {
  ColoredPoint otherPoint = (ColoredPoint) other;
      return (
            super.equals(otherPoint) &&
            this.color==otherPoint.color
                );
```

Retour sur les types génériques

List<E>: E est un paramètre de type. Il peut apparaître dans les méthodes de l'API

- `boolean add(E e);`
- `E get(int index);`
- `List<E> subList(int from,int to);`

Si on a un objet ls de type List<String>,

- on peut ajouter un String s: ls.add(s);
- on peut obtenir un String: ls.get(3);

UPEC Connaissance - Action | UNIVERSITÉ PARIS-EST CRÉTEIL VAL DE MARNE

Covariance et contravariance

Soient:

- A,B,C trois types: C sous-type de B et B sous-type de A
- a,b,c trois objets de type A,B,C respectivement
- ls un objet de type List<B>

L'écriture est *covariante*: je peux écrire des objets de type B et plus petit

- on peut ajouter l'objet b et l'objet c
- ls.add(b), ls.add(c):
- mais ls.add(a) n'est pas accepté, car ls ne peut pas contenir des objets de type A

UPEC Connaissance - Action | UNIVERSITÉ PARIS-EST CRÉTEIL VAL DE MARNE

Covariance et contravariance

Soient:

- A,B,C trois types: C sous-type de B et B sous-type de A
- a,b,c trois objets de type A,B,C respectivement
- ls un objet de type List<B>

La lecture est *contravariante*: je peux lire des objets de type A et plus grand

- je peux faire a = ls.get(2); b=ls.get(2)
- mais c=ls.get(2) n'est pas accepté, car ls peut contenir des objet de type plus grand que C

UPEC
UNIVERSITÉ PARIS-EST CRÉTEIL VAL DE MARNE
Connaissance - Action

Retour sur les types génériques

On peut avoir plusieurs paramètres: Map<K,V>
On peut définir nos classes paramétriques:

```
class Pair<E1,E2> {
private E1 first;
private E2 second;
        public E1 getFirst();
....
}
...
Pair<String,String>
    = new Pair<String,String>("Hallo","World");
```

Retour sur les types génériques

On peut avoir plusieurs paramètres: Map<K,V>
On peut définir nos classes paramétriques:

```
class Pair<E1,E2> {
private E1 first;
private E2 second;
        public E1 getFirst();
....
}
...
Pair<String,String> = new Pair<>("Hallo","World");
```

L'interface Comparable

Une interface intéressante: Comparable<T> Elle contient une seule méthode:

```
int compareTo(T obj)
```

La spécification de cette méthode est

- x.compareTo(y) renvoie un entier négatif si x est "plus petit" que y
- x.compareTo(y) renvoie un entier positif si x est "plus grand" que y
- x.compareTo(y) renvoie 0 si x est "est égal" à y

elle défini un ordre

UPEC UNIVERSITÉ PARIS-EST CRÉTEIL VAL DE MARNE Connaissance - Action

L'interface Comparable

Comparable<T>
Les objets d'un classe qui implémente cette interface peuvent se comparer à des objet de type T.

```
class Myclass implements Comparable<String>{...}
```

Je peux comparer un objet de Myclass à un String. Mais pas le contraire!

UPEC
UNIVERSITÉ PARIS-EST CRÉTEIL VAL DE MARNE
Connaissance - Action

L'interface Comparable

Comparable<T>
Utilisation typique:

```
class Myclass implements Comparable<Myclass>
```

Exemples dans l'api:

- String implémente Comparable<String>
- Integer implémente Comparable<Integer>

L'interface Comparable

Un exemple plus complexe, pour comparer les collections par taille:

```
abstract class CmpColl<T> implements
                                Comparable<CmpColl<T>>,
                                Collection<T> {
    public int compareTo(CmpColl<T> other) {
            return (this.size() - other.size());
    }
}
```

Sous-typage et types génériques

Question: soit B un sous-type de A. Est-ce que List<B> est une sous-type de List<A>

Est-ce que un objet de type List<B> peut respecter tous les contrats promis par le type List<A>?

Le paramètres bornés

Dans tous ces exemples, une classe paramétrique doit accepter tout type comme paramètre.
On peut également restreindre les types acceptés en paramètre

```
class Myclass<T extends String> {...}
```

Dans ce cas je peux seulement passer en paramètre une sous-classe de String:
Myclass<Integer> ce n'est pas légal

Le paramètres bornés

Avantage: je peux utiliser les méthodes de la classe String!

```
class Myclass<T extends String> {
        private T s;
        public boolean test() {
                return s.getChar()=='r';
        }
}
```

Quoi que ce soit T, ses objets ont toutes les methodes de String

Le paramètres bornés

Attention: utiliser extends même si c'est une interface!

```
class MyClass<T extends MyInterface>
```

Le Wildcard

Si B est sous-type de A, List<B> n'est pas un sous-type de List<A>. Dommage!
Solution: le *wildcard!* List<? extends A>

- déclarer un variable ls de type List<? extends A> signifie que ls peut contenir un objet de type List<A>, mais aussi un objet de type List<B> pour B un sous-type de A
- rappel: une variable de type List<A> ne peut pas contenir un objet de type List<B> même si B est un sous-type de A!

Le Wildcard

- List<A> est un sous-type de List<? extends A>
- Si B est un sous-type de A, alors List<? extends B> est un sous-type de List<? extends A>

Le Wildcard

Pourquoi ne pas utiliser toujours List<? extends A> ? Car le paramètre peut être seulement utilisé de façon *covariante*!

```
List<? extends A> ls;
...
A a = ls.get(3);
```

OK! je ne sais pas le type de l'objet renvoyé, mais c'est sûrement un sous-type de A

```
ls.add(new A());
```

PAS OK! je ne sais pas le type des objets que ls peut contenir si ça se trouve ls est un objet de type List<B>

UPEC Connaissance-Action | UNIVERSITÉ PARIS-EST CRÉTEIL VAL DE MARNE

Le Wildcard contravariant

On peut aussi faire l'inverse: List<? super A> ls

- déclarer un variable ls de type List<? super A> signifie que ls peut contenir un objet de type List<A>, mais aussi un objet de type List<C> pour C un super-type de A
- List<A> est un sous-type de List<? super A>
- Si B est un sous-type de A, alors List<? super B> est un *super*-type de List<? extends A>

Le Wildcard contravariant

List<? super A>? peut être seulement utilisé de façon *contravariante*!

```
List<? super A> ls;
...
A a = ls.get(3);
```

Pas OK! je ne sais pas le type de l'objet renvoyé, si ça se trouve c'est un super-type de A.

```
ls.add(new A());
```

OK! je ne sais pas le type des objets que ls peut contenir mais c'est surement un super-type de A, donc il peut contenir aussi un objet de type A.

UPEC Connaissance-Action | UNIVERSITÉ PARIS-EST CRÉTEIL VAL DE MARNE

Un exemple de parametre borné

Je veux créer des ensembles qui peuvent se comparer si leur éléments peuvent se comparer

```
abstract class Domain<E extends Comparable<E>>
         implements Comparable<Domain<E>>, Set<E>{

  private E max (){
     E candidate=null;
     for (E element:this) {
       if (candidate==null || candidate.compareTo(element)>0)
          candidate=element;
     }
     return candidate;
  }

  public compareTo(Domain<E> other) {
        if (this.isEmpty() || other.isEmpty()) throw new SomeException
        return this.max.compareTo(other.max);
   }
}
```