Classes et Objets en Java

Mise à niveau Java

Y. Boichut & F. Moal

Paradigme Objet

- La programmation objet est un paradigme, une manière de « modéliser le monde » :
  - des objets ayant un état interne et un comportement
  - collaborent en s'échangeant des messages (pour fournir les fonctionnalités que l'on demande à l'application)
  - Java, SmallTalk, C++, C#, ...
- D'autres paradigmes :
  - programmation impérative (Pascal, C)
  - programmation fonctionnelle (Scheme, Lisp, "Java")
  - programmation logique (prolog)
  - ...

Orienté Objets

- Manipule des objets
- les programmes sont découpés suivant les types des objets manipulés
- les données sont regroupées avec les traitements qui les utilisent

Toute entité identifiable, concrète ou abstraite, peut être considérée comme un objet

- Un objet réagit à certains messages qu'on lui envoie de l'extérieur ; la façon dont il réagit détermine le comportement de l'objet
- Il ne réagit pas toujours de la même façon à un même message ; sa réaction dépend de l'état dans lequel il est

Un objet en Java

- Un objet possède
  - une adresse en mémoire (identifie l'objet)
  - un comportement (ou interface)
  - un état interne
- L'état interne est donné par des valeurs de variables
- Le comportement est défini par des fonctions ou procédures, appelées méthodes

Interactions entre objets

- Les objets interagissent en s'envoyant des messages synchrones
- Les méthodes de la classe d'un objet correspondent aux messages qu'on peut lui envoyer : quand un objet reçoit un message, il exécute la méthode correspondante

```
objet1.decrisToi();

employe.setSalaire(20000);

voiture.demarre();

voiture.vaAVitesse(50);
```

Classe : regrouper les objets

- Les objets qui collaborent dans une application sont souvent très nombreux
- Mais on peut le plus souvent dégager des types d'objets : des objets ont une structure et un comportement très proches, sinon identiques (eg tous les livres dans une application de gestion d'une bibliothèque)
- La notion de classe correspond à cette notion de types d'objets

Eléments d’une classe

- Les constructeurs (il peut y en avoir plusieurs) servent à créer les instances (les objets) de la classe
- Quand une instance est créée, son état est conservé dans les variables d’instance
- Les méthodes déterminent le comportement des instances de la classe quand elles reçoivent un message
- Les variables et les méthodes s’appellent les membres de la classe

Classe : exemple

```
public class Livre {
    private String titre, auteur;
    private int nbPages;
    // Constructeur
    public Livre(String unTitre, String unAuteur) {
        titre = unTitre;
        auteur = unAuteur;
    }
    public String getAuteur() { // accesseur
        return auteur;
    }
    public void setNbPages(int nb) { // modificateur
        nbPages = nb;
    }
}
```

Rôles d’une classe

- Une classe est
  - un type qui décrit une structure (variables d’instances) et un comportement (méthodes)
  - un module pour décomposer une application en entités plus petites
  - un générateur d’objets (par ses constructeurs)
- Une classe permet d’encapsuler les objets : les membres public sont vus de l’extérieur mais les membres private sont cachés

Conventions de nommage

- Les noms de classes commencent par une majuscule (ce sont les seuls avec les constantes) : Cercle, Object
- Les mots contenus dans un identificateur commencent par une majuscule : UneClasse, uneMethode, uneAutreVariable
- Les constantes sont en majuscules avec les mots séparés par le caractère souligné « _ » : UNE_CONSTANTE
- Si possible, des noms pour les classes et des verbes pour les méthodes : Chien, aboyer()

class en Java

- Une unique classe public par fichier
- Cette classe porte le même nom (commençant par une majuscule) que le fichier .java dans lequel elle est sauvée (public class Livre dans Livre.java)
- Il est possible d’ajouter d’autres classes (sans le mot clé public) dans le même fichier, mais elles ne seront accessibles que dans le package où elles sont définies

Opérateur instanceof

- La syntaxe est :

  objet instanceof nomClasse

- Exemple : `if (x instanceof Livre) { ...`
- Le résultat est un booléen :
  - true si x est de la classe Livre
  - false sinon
- cf compléments sur instanceof dans le cours sur l'héritage

Constructeurs

Classes / instances

- Une instance d’une classe est créée par un des constructeurs de la classe
- Une fois qu’elle est créée, l’instance
  - a son propre état interne (les valeurs des variables d’instance)
  - partage le code qui détermine son comportement (les méthodes) avec les autres instances de la classe
- Chaque classe a un ou plusieurs constructeurs qui servent à
  - créer les instances
  - initialiser l’état de ces instances
- Un constructeur
  - a le même nom que la classe
  - n’a pas de type retour

Exemple de création d'instance

```
public class Employe {
     private String nom, prenom;
     private double salaire;
     // Constructeur
     public Employe(String n, String p) {
          nom = n;
          prenom = p;
     }
     . . .
     public static void main(String[] args) {
          Employe e1;
          e1 = new Employe("Dupond", "Pierre");
          e1.setSalaire(1200);
          . . .
     }
}
```

Plusieurs constructeurs

```
public class Employe {
    private String nom, prenom;
    private double salaire;
    // 2 Constructeurs : surcharge
    public Employe(String n, String p) {
        nom = n;
        prenom = p;
    }
    public Employe(String n, String p, double s) {
        nom = n;
        prenom = p;
        salaire = s;
    }
    ...
    e1 = new Employe("Dupond", "Pierre");
    e2 = new Employe("Durand", "Jacques", 1500);
}
```

Plusieurs constructeurs : this

```
public class Employe {
    private String nom, prenom;
    private double salaire;
    // 2 Constructeurs : appel du second par this
    public Employe(String n, String p) {
        this(n, p, 0.0);
    }
    public Employe(String n, String p, double s) {
        nom = n;
        prenom = p;
        salaire = s;
    }
    ...
    e1 = new Employe("Dupond", "Pierre");
    e2 = new Employe("Durand", "Jacques", 1500);
}
```

Plusieurs constructeurs : this

```
public class Employe {
    private String nom, prenom;
    private double salaire;
    // 2 Constructeurs : appel du second par this
    private static final double DEFAULT_SALAIRE = 0.0;
    public Employe(String n, String p) {
        this(n, p, DEFAULT_SALAIRE);
    }
    public Employe(String n, String p, double s) {
        nom = n;
        prenom = p;
        salaire = s;
    }
    ...
    e1 = new Employe("Dupond", "Pierre");
    e2 = new Employe("Durand", "Jacques", 1500);
}
```

Constructeur par défaut

- Lorsque le code d'une classe ne comporte pas de constructeur, un constructeur sera automatiquement ajouté par Java
- Pour une classe Classe, ce constructeur par défaut sera :

  [public] Classe() { }

- Il est conseillé de toujours écrire au moins un constructeur !

Méthodes

Méthodes / Accesseurs

Deux types particuliers de méthodes servent juste à donner accès aux variables depuis l'extérieur de la classe :

- les accesseurs en lecture pour lire les valeurs des variables ; « accesseur en lecture » est souvent abrégé en « accesseur » ; getter en anglais
- les accesseurs en écriture, ou modificateurs, ou mutateurs, pour modifier leur valeur ; setter en anglais

Les autres types de méthodes permettent aux instances de la classe d'offrir des services plus complexes aux autres instances

Enfin, des méthodes (private) servent de « sous-programmes » utilitaires aux autres méthodes de la classe

Paramètres des méthodes

- Souvent les méthodes ou les constructeurs ont besoin qu'on leur passe des données initiales sous la forme de paramètres
- On doit indiquer le type des paramètres dans la déclaration de la méthode :

```
setSalaire(double unSalaire)

calculerSalaire(int indice, double prime)
```

- Quand la méthode ou le constructeur n'a pas de paramètre, on ne met rien entre les parenthèses :

```
getSalaire()
```

Type de retour d'une méthode

- Quand la méthode renvoie une valeur, on doit indiquer le type de la valeur renvoyée dans la déclaration de la méthode :

  ```
  double calculSalaire(int indice, double prime)
  ```

- Le pseudo-type void indique qu'aucune valeur n'est renvoyée :

  ```
  void setSalaire(double unSalaire)
  ```

Exemple

```
public class Employe {
    . . .
    public void setSalaire(double unSalaire) {
        if (unSalaire >= 0.0)
            salaire = unSalaire;
    }
    public double getSalaire() {
        return salaire;
    }
    public boolean accomplir(Tache t) {
        ...
    }
}
```

Surcharge d'une méthode

- En Java, on peut surcharger une méthode, c'est-à-dire, ajouter une méthode qui a le même nom mais pas la même signature qu'une autre méthode :

```
calculerSalaire(int)
calculerSalaire(int, double)
```

- Il est interdit de surcharger une méthode en changeant seulement le type de retour

```
int calculerSalaire(int)
double calculerSalaire(int)
```

Redéfinition de toString( )

Il est conseillé d'inclure une méthode toString() dans les classes que l'on écrit

- Cette méthode renvoie une chaîne de caractères qui décrit l'instance
- Une description compacte et précise peut être très utile lors de la mise au point des programmes
- System.out.println(objet) affiche la valeur retournée par objet.toString()

```
public class Livre {
    public String toString() {
        return "Livre [titre=" + titre
        + ",auteur=" + auteur
        + ",nbPages=" + nbPages
        + "]";
    }
}
```

Variables

Natures des variables

- Les variables d’instances
  - sont déclarées en dehors de toute méthode
  - conservent l’état d’un objet, instance de la classe
  - sont accessibles et partagées par toutes les méthodes de la classe
- Les variables locales
  - sont déclarées à l’intérieur d’une méthode
  - conservent une valeur utilisée pendant l’exécution de la méthode
  - ne sont accessibles que dans le bloc dans lequel elles ont été déclarées
- Laquelle choisir pour une variable de type objet ?

  Si cet objet est utilisé par plusieurs méthodes de la classe, l’objet devra être référencé par une variable d’instance

Déclaration des variables

- Toute variable doit être déclarée avant d'être utilisée
- Déclaration d'une variable : on indique au compilateur que le programme va utiliser une variable de ce nom et de ce type

```
double prime;

Employe e1;

Point centre;
```

Affectation

- L’affectation d’une valeur à une variable est effectuée par l’instruction

variable = expression;

- L’expression est calculée et ensuite la valeur calculée est affectée à la variable

Exemple :

x = 3;

x = z + 1;

livre = null;

Initialisation

- Une variable doit être initialisée avant d'être utilisée dans une expression
- Si elles ne sont pas initialisées par le programmeur, les variables d'instance (et les variables de classe étudiées plus loin) reçoivent les valeurs par défaut de leur type (0 pour les types numériques, null pour les objets)
- L'utilisation d'une variable locale non initialisée par le programmeur provoque une erreur (pas d'initialisation par défaut) à la compilation
- On peut initialiser une variable en la déclarant

```
double prime = 200.0;
Employe e1 = new Employe("Dupond", "Jean");
double salaire = prime + 500.0;
```

Déclaration / Initialisation

- Il ne faut pas confondre
  - déclaration d'une variable
  - création d'un objet référencé par cette variable

« Employe e1; »

déclare que l'on va utiliser une variable e1 qui référencera un objet de la classe Employe, mais aucun objet n'est créé

```
Employe e1;
e1.setSalaire(1200);
…
```

Désignation des variables d'instance

- Soit un objet o1 ; la valeur d'une variable v de o1 est désignée par o1.v

```
Cercle c1 = new Cercle(p1, 10);

System.out.println(c1.rayon); // affiche 10
```

- Remarque : le plus souvent les variables sont private et on ne peut pas y accéder directement en dehors de leur classe

Variables d’instance final

Une variable d’instance final est constante pour chaque instance ; mais elle peut avoir 2 valeurs différentes pour 2 instances

Une variable d'instance final peut ne pas être initialisée à sa déclaration mais elle doit avoir une valeur à la sortie de tous les constructeurs

Accès aux membres

Plusieurs autorisation d'accès aux membres/var

- **`private`** : seule la classe dans laquelle il est déclaré a accès (à ce membre ou constructeur)
- **`public`** : toutes les classes sans exception y ont accès
- Sinon, par défaut, seules les classes du même paquetage que la classe dans lequel il est déclaré y ont accès (un paquetage est un regroupement de classes ; cf plus loin dans le cours)
- **`protected`** cf cours sur l'héritage

Droits associés à la classe pas à l'objet

En Java, la protection des attributs se fait classe par classe, et pas objet par objet

Un objet a accès à tous les attributs d'un objet de la même classe, même les attributs privés !

```
public class Entier {
    private int laValeur;
    private void plus(Entier autreEntier) {
        laValeur = laValeur + autreEntier.laValeur;
    }
```

Bonnes pratiques pour les droits

[Autant que possible] l'état interne d'un objet (les variables d'instance) doit être **private**

Si on veut autoriser la lecture d'une variable depuis l'extérieur de la classe, on lui associe un accesseur, avec le niveau d'accessibilité que l'on veut

Si on veut autoriser la modification d'une variable, on lui associe un modificateur, qui permet la modification tout en contrôlant la validité de la modification

this

this

- Le code d’une méthode d’instance désigne
  - l’instance qui a reçu le message (l’instance courante), par le mot-clé this
  - donc, les membres de l’instance courante en les préfixant par « this. »
- Lorsqu’il n’y a pas d’ambiguïté, this est optionnel pour désigner un membre de l’instance courante

```
public class Employe {
    private double salaire;
    . . .
    public void setSalaire(double unSalaire) {
        salaire = unSalaire;
    }
    public double getSalaire() {
        return salaire;
    }
```

this explicite

this est utilisé surtout dans 2 occasions :

- pour distinguer une variable d'instance et un paramètre qui ont le même nom :

```
public void setSalaire(double salaire)
    this.salaire = salaire;
}
```

- un objet passe une référence de lui-même à un autre objet :

```
salaire = comptable.calculeSalaire(this);
```

this est final !

this se comporte comme une variable final (mot-clé étudié plus loin), c'est-à-dire qu'on ne peut le modifier ; le code suivant est interdit :

~~this = valeur;~~

Méthodes et variables de Classe

Variables de classe

Certaines variables sont partagées par toutes les instances d'une classe. Ce sont les variables de classe (modificateur `static`)

Si une variable de classe est initialisée dans sa déclaration, cette initialisation est exécutée une seule fois quand la classe est chargée en mémoire

Exemple de variable de classe

```
public class Employe {
    private String nom, prenom;
    private double salaire;
    private static int nbEmployes = 0;
    // Constructeur
    public Employe(String n, String p) {
        nom = n;
        prenom = p;
        nbEmployes++;
    }
    . . .
}
```

Variable de classe final

- Une variable de classe static final est constante dans tout le programme
- exemple :

  static final double PI = 3.14;

- Une variable de classe static final peut ne pas être initialisée à sa déclaration mais elle doit alors recevoir sa valeur dans un bloc d'initialisation static

Méthodes de classe

- Une méthode de classe (modificateur static en Java) exécute une action indépendante d'une instance particulière de la classe
- Une méthode de classe peut être considérée comme un message envoyé à une classe
- Exemple :

```
public static int getNbEmployes() {
    return nbEmployes;
}
```

Appeler une méthode de classe

- Depuis une autre classe, on la préfixe par le nom de la classe :

```
int n = Employe.getNbEmploye();
```

- Depuis sa classe, le nom de la méthode suffit
- On peut aussi la préfixer par une instance quelconque de la classe (à éviter car cela nuit à la lisibilité : on ne voit pas que la méthode est static) :

```
int n = e1.getNbEmploye();
```

Méthodes de classe

Comme une méthode de classe exécute une action indépendante d'une instance particulière de la classe, elle ne peut utiliser de référence à une instance courante (this)

Il est interdit d'écrire

```
static double tripleSalaire() {
    return this.salaire * 3;
}
```

Une méthode de classe ne peut avoir la même signature qu'une méthode d'instance

Question

La méthode main() est nécessairement static.

Pourquoi ?

Blocs d'initialisation static

- Ils permettent d'initialiser les variables static trop complexes à initialiser dans leur déclaration :

```
class UneClasse {
    private static int[] tab = new int[25];
    static {
        for (int i = 0; i < 25; i++) {
            tab[i] = -1;
        }
    }
    ...
```

- Ils sont exécutés une seule fois, quand la classe est chargée en mémoire

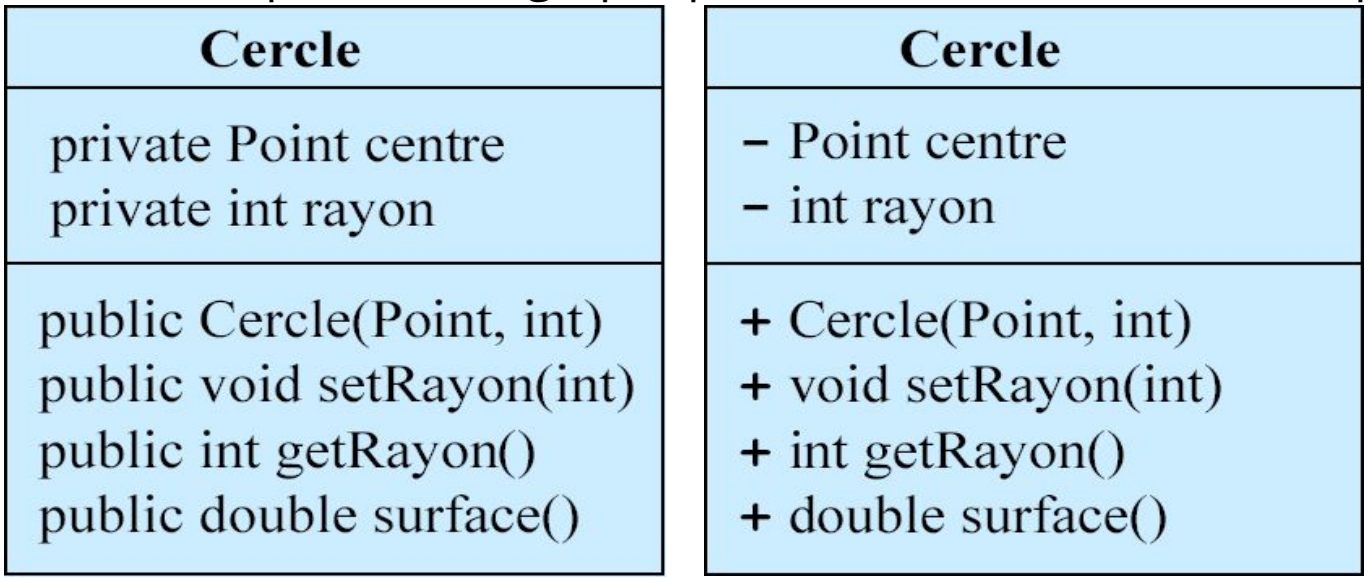
Blocs d'initialisation non static

- Ils servent à initialiser les variables d'instance (ou toute autre initialisation)
- Ils peuvent être utiles en particulier pour les classes internes anonymes (étudiées dans un autre cours) et pour partager du code entre plusieurs constructeurs (leur code est répété par tous les constructeurs)
- La syntaxe est celle des blocs static sans le mot-clé static

```
class UneClasse {
    private int[] tab = new int[25];
    {
        for (int i = 0; i < 25; i++) {
            tab[i] = -1;
        }
    }
```

Représentation “à la UML”

Les classes sont représentées graphiquement sous cette forme simplifiée :



(- : private, # : protected, + : public, $ (ou souligné) : static)

Classes enveloppes

Enveloppes des types primitifs

- En Java certaines manipulations nécessitent de travailler avec des objets (instances de classes) et pas avec des valeurs de types primitifs
- Le paquetage java.lang fournit des classes pour envelopper les types primitifs : Byte, Short, Integer, Long, Float, Double, Boolean, Character
- Attention, les instances de ces classes ne sont pas modifiables (idem String)

Méthodes utilitaires

- les classes enveloppes offrent des méthodes utilitaires (le plus souvent static) pour faire des conversions avec les types primitifs (et avec la classe String)
- Elles offrent aussi des constantes, en particulier, MAX_VALUE et MIN_VALUE

Exemples de conversions : entiers

- String → int : (méthode de Integer)

  static int parseInt(String ch [,int base])

- int → String (méthode de String) :

  static String valueOf(int i)

- Autre moyen pour int → String :

  concaténer avec la chaîne vide (par exemple, "" + 3)

Exemple de la classe Integer

- int → Integer :

  new Integer(int i)

- Integer → int :

  int intValue()

- String → Integer :

  static Integer valueOf(String ch [,int base])

- Integer → String :

  String toString()

Collections et types primitifs

- Le code est alourdi lorsqu'une manipulation nécessite d'envelopper une valeur d'un type primitif
- Ainsi on verra qu'une liste ne peut contenir de type primitif et on sera obligé d'écrire :

```
liste.add(new Integer(89));

int i = liste.get(n).intValue();
```

AutoBoxing

- Le « autoboxing » (mise en boîte) automatise le passage des types primitifs vers les classes qui les enveloppent
- Cette mise en boîte automatique a été introduite par la version 5 du JDK
- L'opération inverse s'appelle « unboxing »
- Le code précédent peut maintenant s'écrire :

```
liste.add(89);

int i = liste.get(n);
```

- Mais il fait la même chose !!

Exemples de boxing/unboxing

- Integer a = 89;
  a++;
  int i = a;

- Integer b = new Integer(1); // unboxing suivi de boxing
  b = b + 2;

- Double d = 56.9;
  d = d / 56.9;

- Tranformer un int en Long :
  Long l = (long)i;

- Attention, une tentative de unboxing avec la valeur null va lancer une NullPointerException

Attention à certains comportements !

```
Integer a = new Integer(1);
int b = 1;
Integer c = new Integer(1);
if (a == b)
    System.out.println("a = b");
else
    System.out.println("a != b");
if (b == c)
    System.out.println("b = c");
else
    System.out.println("b != c");
if (a == c)
    System.out.println("a = c");
else
    System.out.println("a != c");
```

Jeu 1 ! [enfin !!!] : compilo ?

```
class TapeDeck {
    boolean canRecord = false;
    void playTape() {
        System.out.println(“tape playing”);
    }
    void recordTape() {
        System.out.println(“tape recording”);
    }
}
class TapeDeckTestDrive {
    public static void main(String [] args) {
        t.canRecord = true;
        t.playTape();
        if (t.canRecord == true) {
            t.recordTape();
        }
    }
}
```

Jeu 2 : compilo ?

```
class DVDPlayer {
    boolean canRecord = false;
    void recordDVD() {
        System.out.println("DVD recording");
    }
}
class DVDPlayerTestDrive {
    public static void main(String [] args) {
        DVDPlayer d = new DVDPlayer();
        d.canRecord = true;
        d.playDVD();
        if (d.canRecord == true) {
            d.recordDVD();
        }
    }
}
```

Jeu 3 : chamboule tout

```
void jouerCymbale() {
    System.out.println("ding ding
da-ding");
}
```

```
b.jouerCaisseClaire()
```

```
Batterie b = new Batterie();
```

```
class BatterieTestDrive {
```

```
b.caisseClaire = false;
```

```
  if (b.caisseClaire == true) {
      b.jouerCaisseClaire();
}
```

```
void jouerCaisseClaire() {
    System.out.println("bang
bang da-bang");
}
```

```
boolean cymbales = true;
boolean caisseClaire = true;
```

```
public static void main(String… args) {
```

```
b.jouerCymbale();
```

```
class Batterie {
```

Exercice de programmation

Créer une classe Etudiant en considérant les données suivantes :

- Un étudiant a un nom, un prénom, une année de naissance et un identifiant automatiquement créé à la génération
- Nous devons permettre l'accès en lecture à chacune de ses informations (attention les champs doivent rester privés)
- Nous devons permettre l'accès en écriture à tous les champs sauf l'identifiant
- Nous devons développer une méthode permettant de retourner une String avec toutes les informations de l'étudiant

Exercice avancé

Nous voulons créer une classe *DieuJava* qui n'a aucune variable d'instance particulière. La seule contrainte que nous voulons définir est qu'un utilisateur ne puisse pas créer deux instances différentes de cette classe. Imaginez un mécanisme obligeant le développeur possédant votre classe à respecter cette contrainte.

Indice : une variable de classe bien pensée et des constructeurs inaccessibles