Cours Programmation Orientée Objet en Java

Cours 2

Anne Parrain

UFR des Sciences - Université d'Artois
Licence Informatique
Semestre 4

Année universitaire 2018–2019

Plan du cours

1. Retour vers les objets
   - Conventions de nommage
   - Les constructeurs
   - Le mécanisme de création et d'initialisation d'un objet
   - Durée de vie des variables et des objets
   - Accessibilité
   - Accesseurs
   - static or not static
   - Les méthodes
   - This

2. Type énumération

Section 1

Retour vers les objets

Conventions pour les identificateurs

- Classes : tous les mots commencent par une lettre majuscule (ex. : `Etudiant`, `TestEtudiant`, ...)
- Variables et Attributs : comme pour les classes, mais le premier mot commence par une lettre minuscule
  (ex : `nbNotes`, `nom`, `etud1`)
- Constantes : en majuscule avec les mots séparés par le caractère souligné (ex. : `UNE_CONSTANTE`)
- Méthodes : comme pour les variables, mais le premier mot est (très souvent) un verbe (ex. : `ajouteEtudiant`, `estValide`)

Les constructeurs

- par défaut, un objet est construit à l'aide de l'opérateur `new`, et d'un constructeur par défaut :

```
public class Personne{
  private String nom;
  private String prenom;
  private int age = 18;
}
Personne p = new Personne();
```

- Par défaut, le mécanisme d'instanciation fait comme s'il y avait un constructeur vide (= le constructeur par défaut) dans la classe

```
public Personne(){}
```

- les initialisations faites dans les déclarations d'attributs sont prises en compte (ici, `p.age` vaut `18`)

Les constructeurs

- On peut décider comment se construisent les objets :

```
public class Personne{
  private String nom;
  private String prenom;
  private int age = 18;
  public Personne(String nom, String prenom){
    this.nom = nom;
    this.prenom = prenom;
  }
}
```

- la présence d'un constructeur annule le mécanisme du constructeur par défaut. Maintenant :

```
Personne p = new Personne(); // NON
Personne p = new Personne("Dupont", "Jean"); // OUI
```

Les constructeurs

On peut écrire plusieurs constructeurs (qui spécifient plusieurs manières d'instancier la classe) :

```
public class Personne{
  public static final int MAJORITE = 18;
  public static final int MAX_AGE = 120;
  private String nom;
  private String prenom;
  private int age = MAJORITE;

  public Personne(String nom, String prenom){
    this.nom = nom;
    this.prenom = prenom;
  }
```

Les constructeurs

```
  public Personne(String nom, String prenom, int age){
    this.nom = nom;
    this.prenom = prenom;
    if ((age > 0) && (age < MAX_AGE))
      this.age = age;
  }
  public Personne() {}
  public void setNom(String nom){
    if (this.nom == null)
      this.nom = nom;
  }
  public void setPrenom(String prenom){
    if (this.prenom == null)
      this.prenom = prenom;
  }
}//Personne
```

Pour instancier la classe Personne

```
Personne dupont = new Personne("Dupont","Jean");
// Jean Dupont a 18 ans. On ne peut plus changer
// aucune de ses caracteristiques

Personne durand = new Personne("Durand","Andre",52);
// Andre Durand a 52 ans. On ne peut plus changer
// aucune de ses caracteristiques.

Personne qqun = new Personne();
// qqun a 18 ans. On peut maintenant lui donner
// un nom et un prenom.
qqun.setNom("Dumoulin);
qqun.setPrenom("Nathalie");
// qqun s'appelle Nathalie Dumoulin et a 18 ans.
qqun.setNom("Duchemin");
// n'est pas pris en compte.
```

Les constructeurs

Les constructeurs font du contrôle sur l'initialisation des attributs. On préfèrera toujours éviter de dupliquer du code :

```
public class Personne {
  // declaration d'attributs ...

  public Personne(String nom, String prenom, int age){
    this.nom = nom;
    this.prenom = prenom;
    if ((age > 0) && (age < MAX_AGE))
      this.age = age;
  }
  public Personne(String nom, String prenom){
    this(nom,prenom,MAJORITE);
  }
}
```

Les constructeurs

- En général, ce sont les constructeurs avec beaucoup d'arguments qui contiennent les instructions de contrôle sur les attributs ;
- Donc, en général, ce sont les constructeurs à peu d'arguments qui appellent les constructeurs avec beaucoup d'arguments :

```
public class Personne {
  // declaration d'attributs ...

  public Personne(String nom, String prenom){
    // construire une personne sans donner son age,
    // c'est faire le choix de la valeur par defaut
    // c'est comme construire une personne de 18 ans
    this(nom,prenom,MAJORITE);
  }

}
```

Les constructeurs

Mais ce n'est pas obligatoire ! On aurait pu écrire :

```
public class Personne {
  // declaration d'attributs ...

  public Personne(String nom, String prenom){
    this.nom = nom;
    this.prenom = prenom;
  }
  public Personne(String nom, String prenom, int age){
    this(nom,prenom);
    if ((age > 0) && (age < MAX_AGE))
      this.age = age;
  }
}
```

Les constructeurs

Contraintes sur l'emploi de `this()` (le constructeur) :

- il ne peut être appelé qu'à l'intérieur d'un constructeur ;
- il doit être la première instruction du constructeur dans lequel il est appelé
- il peut être appelé en cascade :

```
public class Personne {
  // comme d'habitude, plus :
  public static final char HOMME = 'M';
  public static final char FEMME = 'F';

  private char genre = FEMME;
```

Les constructeurs

```
  Personne(String nom, String prenom){
    this.nom = nom;
    this.prenom = prenom;
  }
  Personne(String nom, String prenom, int age){
    this(nom,prenom);
    if ((age > 0) && (age < MAX_AGE))
      this.age = age;
  }
  Personne(String nom, String prenom, int age, char genre){
    this(nom,prenom,age);
    if ((genre == HOMME)||(genre == FEMME))
      this.genre = genre;
  }
}
```

Le mécanisme de création et d'initialisation d'un objet

1. les variables de classe sont créées dès les premier appel à la classe (soit pour une instanciation – appel à `new`, soit par un accès direct à une variable de classe) :
   1.1 initialisées au niveau de la déclaration ;
   1.2 ou par un constructeur de statiques (plus rares) ;
2. les variables d'instance initialisées hors constructeur (à la déclaration) ;
3. exécution du constructeur ;

Exemple

```
public class FeuTricolore{
  public static final String[] COULEURS =
      {"VERT","ORANGE","ROUGE"};
  private int feu;
  public FeuTricolore(int feu){
    this.feu = feu%COULEURS.length;
  }
  public FeuTricolore(){}
  public String toString(){
    return COULEURS[feu];
  }
  public void tourne(){
    feu = (feu+1)%COULEURS.length;
  }
}//FeuTricolore
```

Exemple

```
public static void main(String[] args){
  System.out.println("Quelles sont les " + "couleurs
      possibles?");
  for(int i=0; i < FeuTricolore.COULEURS.length; i++)
    System.out.println(FeuTricolore.COULEURS[i]);
  FeuTricolore ft = new FeuTricolore(2);
  System.out.println(ft);
  for (int i=0;i<10;i++){
    ft.tourne();
    System.out.println(ft);
  }
}//main
```

Durée de vie des variables et des objets

- un objet existe tant qu'il existe une référence vers lui ;
- une variable d'instance d'un type primitif existe tant qu'il existe une référence vers l'objet auquel elle appartient.

```
public Personne dummy(){
  String dup = "Dupont";
  Personne dupont = new
      Personne(dup,"Jean",23,Personne.HOMME);
  Personne dupont2 = new
      Personne(dup,"Paul",23,Personne.HOMME);
  Personne durand = new Personne("Durand","Andree",52);
  dupont = durand;
  return durand;
}
.... // ailleurs, beaucoup plus loin
Personne p = dummy();
```

Initialisation des variables

- un attribut (une variable d'instance ou de classe) est toujours initialisé à la création de l'objet (ou de la classe) à laquelle il appartient que ce soit
  - par une initialisation explicite lors de la déclaration
  - par une initialisation dans le constructeur
  - par une initialisation implicite sinon
- une variable locale n'est jamais initialisée par défaut

Les modificateurs d'accessibilité

A l'intérieur d'une classe, une variable ou une méthode peut être définie avec un modificateur d'accès. Les différents modificateurs d'accessibilité sont :

private l'élément (variable ou méthode, d'instance ou de classe) est privé, il n'est accessible que depuis la classe elle-même (le code de la classe dans laquelle il est défini) ;

pas de modificateur d'accès l'accès est dit *package*.

protected l'accès est étendu (par rapport à `private` au code des classes du même package et aux sous-classes de la classe. *Nous reviendrons plus tard sur cette notion, liée à l'héritage ;*

public accessible à partir de tout code qui a accès à la classe où l'élément est défini.

Accesseurs (1)

Ajoutons un nouvel accesseur dans la classe `Promotion` :

```
private Etudiant[] lesEtudiants;

public Etudiant[] getLesEtudiants(){ return lesEtudiants;}
```

et maintenant...

```
// ailleurs...
Promotion lesL1;
...
Etudiant[] desEtudiants = lesL1.getLesEtudiants();
for(int i = 0; i < desEtudiants.length; i++)
   desEtudiants[i] = null;
```

ERREUR !!

Accesseurs (2)

Conclusion : il faut se méfier des accesseurs en lecture et conserver cachées les propriétés qui ne concernent pas les autres...

Oui, mais alors

```
//class Personne
public String getNom(){
  return nom;
}
```

est-ce dangereux ?

Accesseurs (3)

Non !

```
// class Test
Personne p = new Personne("Dumoulin","Isabelle",20);
String unNom = p.getNom();
unNom = unNom.toUpperCase();
// unNom contient "DUMOULIN"
// p.nom contient "Dumoulin"
```

Les instances de `String` sont non modifiables (comme les instances de classes enveloppes).

Attributs de classe et attributs d'instance

- un attribut de classe (`static`) est un attribut qui n'est créé qu'une seule fois par classe, et qui est partagé par toutes les instances de cette classe ;
- un attribut d'instance est un attribut qui existe pour chaque instance d'une classe, et qui contient une valeur relative à l'instance à laquelle il appartient.

Exemples :

- les constantes d'une classe
- les attributs qui contiennent une information relative à l'ensemble des instances d'une classe (compteur d'instances,. . .)

Méthodes d'instance, méthodes de classe

Une méthode qui ne manipule aucune variable d'instance (par exemple, une méthode qui effectue un calcul uniquement à partir de ses paramètres) a vocation à être déclarée `static`.

```
public static boolean estValide(int age){
  return (age > 0 && age <= MAXAGE);
}
```

La surcharge de méthodes

- C'est la possibilité d'avoir un même nom de méthode avec des signatures différentes (et donc des comportements différents).
- C'est la nature des paramètres effectifs qui détermine la méthode exécutée.
- Le compilateur choisit la méthode à utiliser par concordance de l'appel et de la signature.
- La signature d'une méthode correspond à son nom, son nombre de paramètres, leur type et l'ordre de ces paramètres et les types de ces paramètres.
- Attention : le type de retour d'une méthode ne fait pas partie de la signature
  - deux méthodes ne peuvent pas être différenciées uniquement par leur type de retour

```
public class Artiste {
   public void dessine(String s) { ... }
   public void dessine(int i) { ... }
   public void dessine(int i, double f) { ... }
}
```

Paramètres

- Types permis : types primitifs, tableaux et classes.
- Les paramètres en Java sont passés par valeur.
- Pour définir une méthode avec un nombre arbitraire de paramètres :

```
public Polygone creePolygone(Point... pt) {
  ...
  carreCote1 = (pt[1].x - pt[0].x)*(pt[1].x - pt[0].x)+
               (pt[1].y - pt[0].y)*(pt[1].y - pt[0].y);
  ...
}
```

Paramètres (2)

- Le passage de paramètres se fait par valeur :

```
public class PassageParValeur {
    public static void main(String[] args) {
        int x = 3;

        passeMethode(x);
        System.out.println("Apres passage x = " + x);
    }

    public static void passeMethode(int p) {
        p = 10;
    }
}
```

- Sortie :

```
Apres passage x = 3
```

Paramètres (3)

- Attention : une variable qui est un objet, contient la référence de l'objet !

```
public void deplaceCercle(Cercle cercle, int deltaX,
                                         int deltaY) {
    // deplace l'origine du cercle a x+deltaX, y+deltaY
    cercle.setX(cercle.getX() + deltaX);
    cercle.setY(cercle.getY() + deltaY);

    // affecte une nouvelle reference au cercle
    cercle = new Cercle(0, 0);
}
```

- Que se passe-t-il ici :

```
deplaceCercle(monCercle, 23, 56);
```

Une méthode particulière : `toString`

```
public String toString()
```

- est la méthode pour retourner une représentation textuelle d'un objet
- est définie par défaut
- est appelée de manière implicite par la méthode `println()` de `System.out`
- équivalent de `__str__()` en Python

Une méthode particulière : `equals`

`public boolean equals(Object o)`

- permet de définir comment sont équivalents deux objets
- `==` teste si deux objets sont égaux : s'ils ont le même identifiant
- `.equals()` teste si deux objets sont équivalents

```
// dans la classe Etudiant
public boolean equals(Etudiant e){
  return e.nom == this.nom && e.prenom == this.prenom;
}
...

// ailleurs
Etudiant etud1 = new Etudiant("Toto","Titi");
Etudiant etud2 = new Etudiant("Toto","Titi");
if (etud1 == etud2) ... // Faux
if (etud1.equals(etud2)) ... // Vrai
```

- équivalent de `__eq()__` en Python

This

Le mot-clé `this` désigne l'*instance courante*.

Il peut être utilisé :

- pour accéder à un membre de l'instance
  `this.nom = nom;`
- pour passer sa référence à un autre objet

This

```
public class Etudiant {
  // comme d'habitude, mais on ajoute un attribut :
  private Promotion promo;
  public Etudiant(String nom, String prenom, Promotion promo){
    this.promo = promo // idem pour les autres attributs
    ...
  }
}
public class Promotion {
  public boolean ajouteEtudiant(String nom, String prenom){
    Etudiant etud = new Etudiant(nom,prenom,this);
    ...
  }
}
```

Section 2

Type énumération

Type énumération

- Un type énumeration est une classe dont les instances sont limitées à un ensemble de constantes.
- En Java on utilise le mot-clé `enum` :

```
enum Jour {
  LUNDI, MARDI, MERCREDI, JEUDI,
  VENDREDI, SAMEDI, DIMANCHE
}
```

Exemple

```
class DemoEnum {
  Jour jour;

  public DemoEnum(Jour jour) {
    this.jour = jour;
  }
  public void agenda() {
    switch (jour) {
      case Jour.LUNDI:
        System.out.println("TP et TD."); break;
      case Jour.MARDI: case Jour.MERCREDI:
        System.out.println("Cours."); break;
      case Jour.VENDREDI:
        System.out.println("TP"); break
      default:
        System.out.println("Rien a faire!!"); break;
    }
  }
```

Exemple (2)

```
  public static void main(String[] args) {
    new DemoEnum(Jour.LUNDI).agenda();
    new DemoEnum(Jour.MARDI).agenda();
    new DemoEnum(Jour.MERCREDI).agenda();
    new DemoEnum(Jour.JEUDI).agenda();
    new DemoEnum(Jour.VENDREDI).agenda();
    new DemoEnum(Jour.SAMEDI).agenda();
    new DemoEnum(Jour.DIMANCHE).agenda();
  }
}
```

Type énumération (2)

- Un type énumération est une classe.
- Il peut contenir des méthodes et d'autres attributs.
- Le compilateur créé par défaut la méthode `values()`, qui retourne un tableau contenant les valeurs de l'énumération dans l'ordre dans lequelles elles sont déclarées :

```
Jour[] semaine = Jour.values();
```

La commande *for-each*

- Cette commande peut être utilisée avec les tableaux (donc, aussi avec les énumérations) :

```
for (Type Identificateur : Expression) Commande
```

- Exemple :

```
for (int item : {1, 2, 3, 4, 5}) {
  System.out.println("Numero courant : " + item);
}
```

- équivaut à :

```
int[] tableau = {1, 2, 3, 4, 5};
for (int i = 0; i < tableau.length; i++) {
  int item = tableau[i];
  System.out.println("Numero courant : " + item);
}
```

La commande *for-each* avec un type énumération

- Exemple :

```
for(Jour j : Jour.values()) {
  new DemoEnum(j).agenda();
}
```

- équivaut à :

```
Jour[] tableau = Jour.values();
for (int i = 0; i < tableau.length; i++) {
  Jour j = tableau[i];
  new DemoEnum(j).agenda();
}
```

Type énumération avec attributs et méthodes

```
enum Planete {
 MERCURE (3.303e+23, 2.4397e6),
 VENUS  (4.869e+24, 6.0518e6),
 TERRE  (5.976e+24, 6.37814e6),
 MARS   (6.421e+23, 3.3972e6),
 JUPITER (1.9e+27, 7.1492e7),
 SATURN (5.688e+26, 6.0268e7),
 URANUS (8.686e+25, 2.5559e7),
 NEPTUNE (1.024e+26, 2.4746e7);

 private final double masse;
 private final double rayon;
 // constante gravitationnelle
 public static final double G = 6.67300E-11;

 Planete(double masse, double rayon) {
   this.masse = masse;
   this.rayon = rayon;
 }
```

Type énumération avec attributs et méthodes (2)

```
 private double masse() { return masse; }
 private double rayon() { return rayon; }

 double gravSurface() {
   return G * masse / (rayon * rayon);
 }
 double poidsSurface(double autreMasse) {
   return autreMasse * gravSurface();
 }

 public static void main(String[] args) {
   double poidsSurTerre = Double.parseDouble(args[0]);
   double masse = poidsSurTerre / TERRE.gravSurface();
   for (Planete p : Planete.values())
     System.out.println("Poids sur " + p +
       " : " + p.poidsSurface(masse));
 }
}
```