Le Langage Java

1ère année

M. Bastreghi J. Beleho P. Bettens M. Codutti
A. Hallal C. Leruste D. Nabet N. Pettiaux A. Rousseau

Haute École de Bruxelles — École Supérieure d'Informatique

Année académique 2011 / 2012

Leçon 4 — L'orienté objet

- Structure générale d'un programme
- Variables et assignation
- Lire au clavier
- Constantes
- Conventions
- Commentaires
- Présentation
- Les tests
- Plan de tests
- JUnit
- Conclusion
- Constructeur
- Instanciation
- Accesseurs
- Mutateurs
- Surcharge
- this
- static
- Des objets comme attributs
- Objets et tableaux
- Héritage
- Object
- Représentation

Avertissement

Pour qu'un langage soit *orienté objet* il doit posséder 3 propriétés

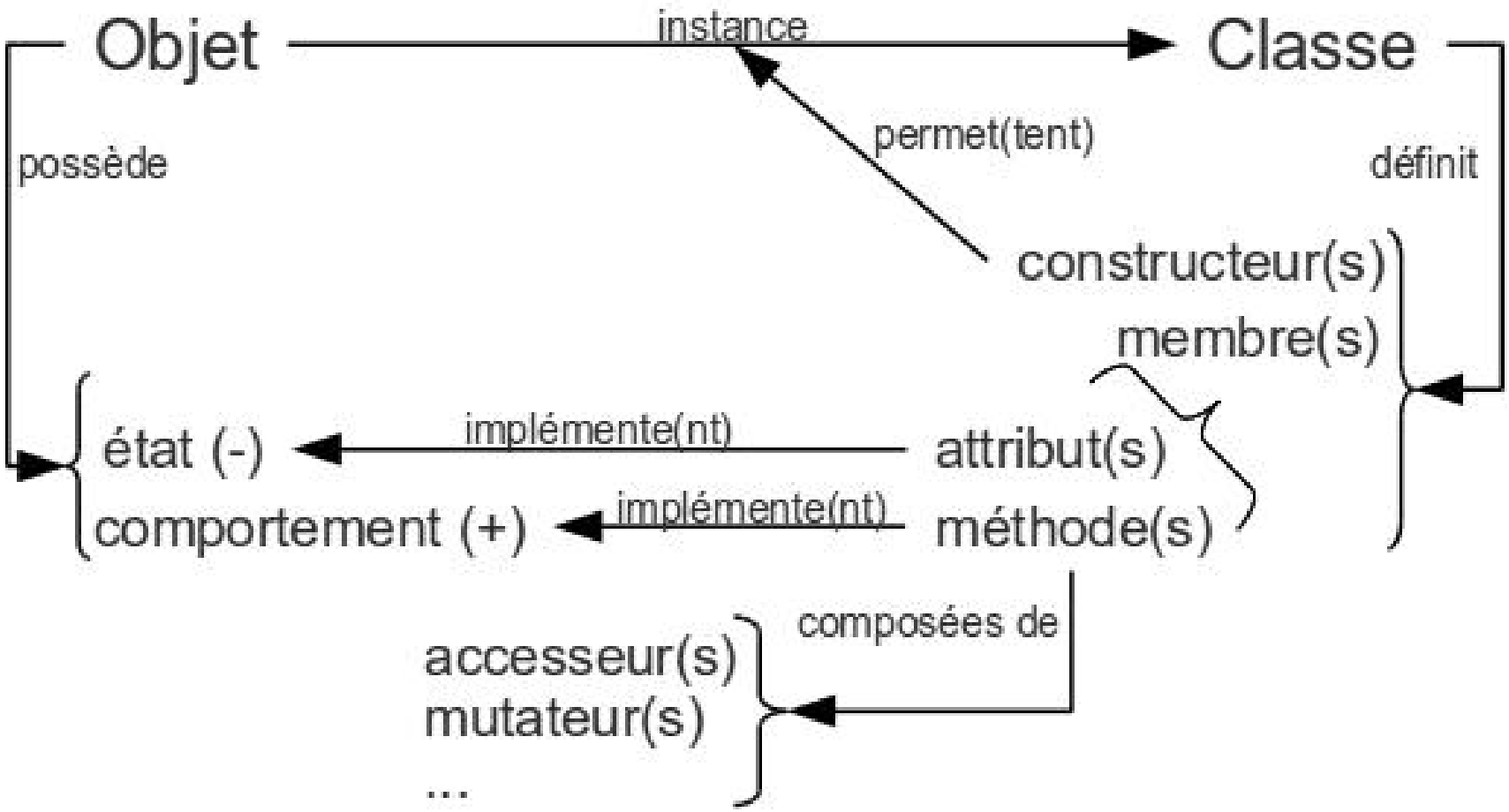
- L' **encapsulation**
- L' **héritage**
- Le **polymorphisme**

Trop pour le cours de 1ère année

- Nous allons surtout voir l'encapsulation (comme au cours de logique)
- Et effleurer le reste $\longrightarrow$ parfois imprécis

Rappels

Voici ce que vous avez déjà vu en logique



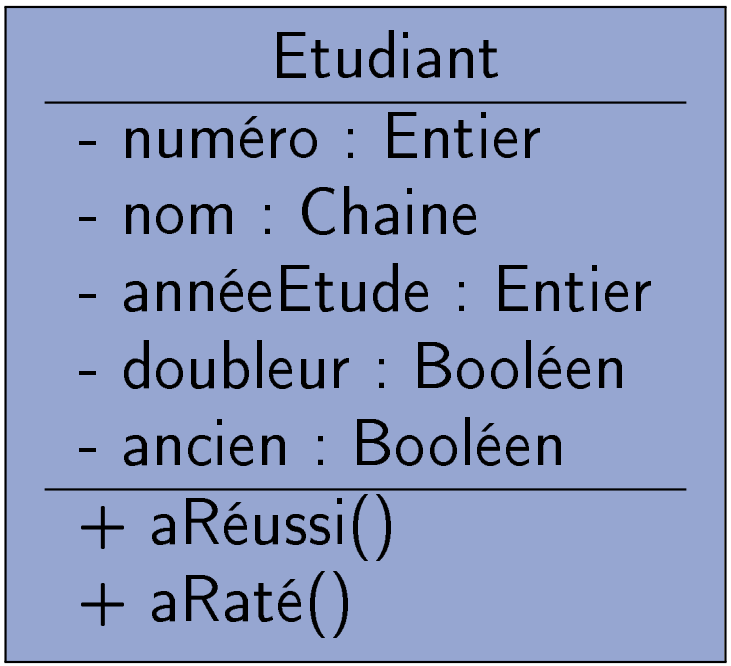
Présentation de l'exemple

Illustrons ces concepts avec la notion d'*étudiant à l'ESI*

- Un étudiant
  - possède un nom et un numéro unique
  - est inscrit dans une année d'étude
  - est doubleur ou pas
  - est un *ancien* (a terminé) ou pas
- Il peut réussir son année ou la rater

La classe

Exemple : Représentation graphique (UML) de la classe *Etudiant*



Nom de la classe

Attributs

Le "-" indique qu'ils sont **privés**

Connus uniquement dans la classe

En Java on écrira **private**

Méthodes

Le "+" car elles sont **publiques**

En Java on écrira **public**

Les objets

Exemple : Représentation graphique de 2 objets (instances) possibles :

James Gosling : Etudiant
- numéro = 34000 - nom = "James Gosling" - annéeEtude = 1 - doubleur = faux - ancien = faux
+ aRéussi() + aRaté()

Ada Lovelace : Etudiant
- numéro = 33800 - nom = "Ada Lovelace" - annéeEtude = 2 - doubleur = faux - ancien = faux
+ aRéussi() + aRaté()

Les membres

Chaque instance possède les mêmes attributs mais avec des **valeurs différentes**

Les méthodes d'une instance agissent sur les attributs de cette instance

- La méthode *aRaté()* d'un étudiant va mettre **son** attribut *doubleur* à vrai
- Que ferait la méthode *aRéussi()* ?

La classe en Java

À ce stade la classe `Etudiant` peut s'écrire :

```
public class Etudiant {

  private int numéro;
  private String nom;
  private int annéeEtude;
  private boolean doubleur;
  private boolean ancien;

  public void aRaté() {
    doubleur = true;
  }
```

```
  public void aRéussi() {
    doubleur = false;
    annéeEtude++;
    if (annéeEtude == 4 ) {
      ancien = true;
    }
  }

}
```

- Remarquez l'absence de **static** pour les méthodes

OO or not OO ?

On utilisait déjà **class**. On faisait de l'objet ?

Oui et non ;)

- Java est un langage orienté objet
- Mais il permet une écriture non OO
- Via l'utilisation de **static** (qui a un sens plus large que nous détaillerons plus loin)

OO or not OO ?

En gros, on a 2 sortes de classes :

	approche non OO	approche OO
But	regrouper des méthodes	définir un type de données
Attribut	non (sauf constantes)	oui
Instances	non	oui
Utilisation via	le nom de la classe	une instance
static	oui	non
Fréquence	rare	fréquent
Exemples	Math	String, Scanner

En pratique, on rencontrera des situations intermédiaires

Visibilité des membres

En Java : 4 visibilités

public : visible dans **toutes** les classes

privé : n'est accessible que de **la** classe

«paqueté» : (pas de mot clé) visible dans toutes les classes du *package*

protégé : (pas vu en 1ère année)

Rappel **bonne pratique** :
attributs privés / méthodes publiques

Constructeur

Exemple : Définition d'un constructeur

```
public Etudiant (int unNuméro, String unNom) {
  numéro = unNuméro;
  nom = unNom;
  annéeEtude = 1;
  doubleur = false;
  ancien = false;
}
```

Ressemble à une méthode mais

- Pas de type de retour déclaré
- A le même nom que celui de la classe

Instanciation

Pour instancier

- On utilise l'opérateur **new**
- On fournit les paramètres au constructeur

Exemple : instanciation d'un étudiant

```
Etudiant ada = new Etudiant(33800, "Ada Lovelace");
```

- Crée un nouvel objet Etudiant
- Appelle le constructeur pour l'initialiser

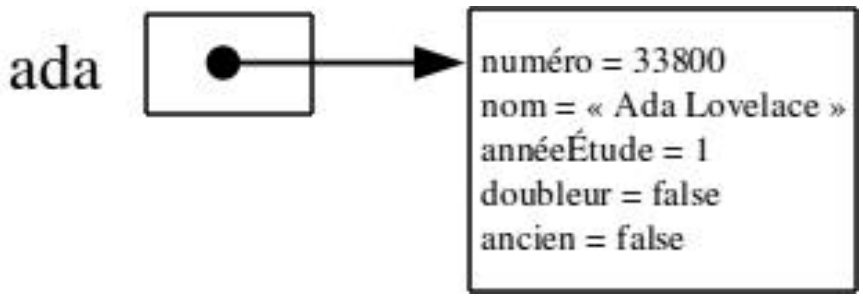
Instanciation

Une classe est un type **référence** (comme les tableaux)

Exemple :

```
Etudiant ada;  // référence créée sur la pile
```


```
ada = new Etudiant(33800,"Ada Lovelace"); // objet créé sur le tas
```



Appel d'une méthode

Utilisation de la notation *pointée* (opérateur .)

Exemple

```
public static void main(String[] args) {
    Etudiant ada = new Etudiant(33800, "Ada Lovelace");
    ada.aRéussi();
}
```

Code que l'on peut trouver

- dans une autre classe
- dans la classe même

Accesseurs

Accesseur : méthode donnant la valeur d'un attribut

Exemple : pour notre classe étudiant

```
public int getNuméro() {return numéro;}
public String getNom() {return nom;}
public int getAnnéeEtude() {return annéeEtude;}
public boolean isDoubleur() {return doubleur;}
public boolean isAncien() {return ancien;}
```

Par convention, l'accesseur de attribut est getAttribut (isAttribut pour un booléen)

Appel d'une méthode

Exemple : Utilisation des accesseurs

```
public static void main(String[] args) {
    Etudiant ada = new Etudiant(33800, "Ada␣Lovelace");
    System.out.println(ada.getAnnéeEtude()); // 1
    ada.aRéussi();
    System.out.println(ada.getAnnéeEtude()); // 2
    System.out.println(ada.isDoubleur()); // false
    ada.aRaté();
    System.out.println(ada.getAnnéeEtude()); // 2
    System.out.println(ada.isDoubleur()); // true
}
```

Mutateurs

Mutateur : sert à modifier un attribut

Exemple : un mutateur possible pour *Etudiant*

```
public void setNom(String unNom) {nom = unNom;}
```

Par convention, le mutateur de attribut est setAttribut

Exemple : Appel d'un mutateur

```
Etudiant ada = new Etudiant(33800,"Ada Lovelace");
System.out. println ( ada.getNom() );
ada.setNom("James Gosling");
System.out. println ( ada.getNom() );
```

Mutateurs

Bonne pratique :

Bien réfléchir avant de fournir un mutateur

- Est-ce que le numéro peut changer ? Non !
- Est-ce que le nom peut changer ? Euh !
- Est-ce que l'année peut changer ? Oui !
  - Mais est-ce qu'il faut permettre de la changer directement ?
  - ou uniquement via des méthodes comme aRéussi() ? À voir au cas par cas

Tests de validité

Il est conseillé d'effectuer des **tests de validité** sur les paramètres

- Constructeur : objet créé dans un état valide
- Mutateur : l'état reste valide

Exemple :

```
public void setnuméro(int unNuméro) {
   if (unNuméro<=0) {
     throw new IllegalArgumentException("Le numéro est négatif !");
   }
   numéro = unNuméro;
}
```

Surcharge

Surcharge (overloading) : possibilité de définir plusieurs méthodes/constructeurs

- De même nom
- Si signatures différentes
- Facilité pour l'utilisateur

Très utile pour les constructeurs

- Plusieurs façons d'initialiser l'état

Surcharge

Exemple : Constructeurs pour `Etudiant`

```
public Etudiant (int unNuméro, String unNom) {
  numéro = unNuméro;
  nom = unNom;
  annéeEtude = 1;
  doubleur = false;
  ancien = false;
}

public Etudiant (int unNuméro, String unNom, int année,
                 boolean doubl, boolean anc) {
  numéro = unNuméro;
  nom = unNom;
  annéeEtude = année;
  doubleur = doubl;
  ancien = anc;
}
```

this()

Souvent, les constructeurs d'une même classe se ressemblent

- Cf. exemple précédent
- Plus facile si un constructeur appelle l'autre
- On utilise la notation **this()**
- Exemple

```
public Etudiant (int numéro, String nom) {
  this(numéro, nom, 1, false, false);
}
```

- Doit être la **première instruction**

Le mot clé «this»

Le mot clé **this** est une référence à soi-même

- Implicite lors d'une utilisation directe du membre
- **Exemple**

```
public Etudiant( String nom ) {
  setNom( nom ); // implicitement : this.setNom( nom );
}

public void setNom( String unNom ) {
  nom = unNom; // implicitement: this.nom = unNom;
}
```

Le mot clé «this»

Règle : un paramètre/une variable locale **masque** un attribut

- **this** permet d'accéder à l'attribut masqué
- **Exemple**

```
public void setNom( String nom ) {
   this.nom = nom;
}
```

- Certains l'utilisent systématiquement pour une meilleure lisibilité

Comprendre «static»

static s'applique aux membres (attributs + méthodes)

- N'est plus un membre de l'objet (instance de la classe) mais un membre de la classe
- Est **partagé** par toutes les instances

Comprendre «static»

Attribut statique

- Existe en un seul exemplaire
- Est initialisé lors du chargement de la classe (une seule fois)
- Utilisation courante : constantes

Exemple

```
public Board {
  public static final int NB_LIGNES = 8;
  public static final int NB_COLONNES = 7;
}
```

Comprendre «static»

Méthode statique

- Elle ne possède ni ne peut modifier les attributs de l'objet
- Utilisation courante : méthodes non objets

Exemple

```
public class Outils {
  public static int abs(int nb) {
    return nb < 0 ? -nb : nb;
  }
}
```

Comprendre «static»

À l'extérieur de la classe,

- on préfixe par le nom de la classe

```
int abs = Outils.abs(-3);
int lg = Board.NB_LIGNES;
double un = Math.log (Math.E);
```

- ou un objet de la classe (non recommandé)

```
int lg = monBoard.NB_LIGNES;
```

Comprendre «static»

import static crée un raccourci pour l'accès aux membres statiques

Exemple

```
import static java.lang.Math.log;
import static java.lang.Math.E;
public class Test {
  public static void main( String[] args ) {
    System.out.println ( log(E) );
  }
}
```

Un mot sur les structures

En Logique, vous avez vu le concept de structure

- On peut imiter cette construction
- **Exemple** : une structure Adresse

```
structure Adresse composée de
    rue : chaine
    numéro : chaine
    code : entier
    localité : chaine
fin structure
```

```
public class Adresse {
    public String rue;
    public String numéro;
    public int code;
    public String localité ;
}
```

- Mais, on préfère une classe normale qui permet de contrôler la valeur des champs

Précision sur l'instanciation

Attributs initialisés à une **valeur par défaut**
(comme pour les tableaux)

- Numérique : 0
- Booléen : **false**
- référence : **null** (référence vers **rien**)

Rarement ce qui est souhaité
$\longrightarrow$ toujours donner des valeurs explicites

Précision sur les constructeurs

Il existe un **constructeur par défaut**

- sans paramètre
- ne fait rien
- uniquement **si pas de constructeur explicite**
- Rarement une bonne idée

$\longrightarrow$ toujours écrire explicitement un constructeur

Des objets comme attributs

Une classe définit un type à part entière

$\longrightarrow$ peut être attribut d'une autre classe

- Ex : `String` pour le nom d'un étudiant
- Ex : `Date` (de naissance) d'un étudiant

Des objets comme attributs

Une classe définit un type à part entière

$\longrightarrow$ peut être attribut d'une autre classe

- **Ex** : `String` pour le nom d'un étudiant
- **Ex** : `Date` (de naissance) d'un étudiant

Exemple : Définissons le concept d'adresse

```
public class Adresse {
  private String rue;
  private String numéro;
  private int codePostal;
  private String  localité ;

  public Adresse (String uneRue, String unNuméro,
          int unCodePostal, String  unelocalité ) {
    // ...
  }
  // + accesseurs
  // pas de mutateur ! Pourquoi ?
}
```

Des objets comme attributs

Exemple : Ajoutons une adresse à un étudiant

```
public class Etudiant {
  private Adresse adresse;

  public Etudiant (int unNuméro, String unNom, Adresse uneAdresse) {
    adresse = uneAdresse;
    // ...
  }

  public Adresse getAdressse() {return adresse;}

  public void setAdresse(Adresse uneAdresse) {
    adresse = uneAdresse;
  }
  // ...
}
```

Des objets comme attributs

Exemple : Créons un étudiant

```
Adresse adresse = new Adresse("Rue Royale", "67", 1000, "Bruxelles");
Etudiant james = new Etudiant( 34000, "James Gosling", adresse );
```

ou, en condensé

```
Etudiant james = new Etudiant (
                        34000, "James Gosling",
                        new Adresse("Rue Royale", "67", 1000, "Bruxelles")
                );
```

Des tableaux d'objets

Une classe définit un type de données

$\longrightarrow$ on peut définir des tableaux d'objets

Exemple : un tableau d'étudiants Etudiant []

```
// Affiche un tableau d'étudiants
public static void afficher (Etudiant [] étudiants) {
    System.out.println ("Il y a " + étudiants.length + " étudiants");
    for( Etudiant étudiant : étudiants ) {
        System.out.println ( étudiant );
    }
}
```

Des tableaux d'objets

```
// Faire réussir tous les étudiants
public static void tournéeGénérale(Etudiant[] étudiants) {
    // Faire un schéma pour comprendre que le foreach est correct
    for( Etudiant étudiant : étudiants ) {
        étudiant.aRéussi();
    }
}
```

```
// Test
Etudiant[] groupe11 = {
    new Etudiant("20000", "Tintin"),
    new Etudiant("20001", "Milou"),
    new Etudiant("20002", "Professeur Tournesol"),
    new Etudiant("20003", "Capitaine Haddock");
};
afficher(groupe11);
tournéeGénérale(groupe11);
afficher(groupe11);
```

Des tableaux **dans** les objets

Un tableau définit un type de données
$\longrightarrow$ on peut le trouver comme attribut

Exemple : Définissons, la classe Groupe (d'étudiants)

- La taille (maximale) du groupe sera donnée à la construction
- Une méthode permet d'ajouter un étudiant au groupe

Des tableaux **dans** les objets

```
public class Groupe {
    private Etudiant[] étudiants;
    private int nbEtudiants;

    public Groupe(int taille ) {
        if ( taille < 1)
          throw new IllegalArgumentException("Pas de groupe vide");
        étudiants = new Etudiant[taille ]; // Faire un schéma !
        nbEtudiants = 0;
    }

    public void ajouter(Etudiant étudiant) {
        if (nbEtudiants == étudiants.length)
          throw new IllegalStateException("Plus de place !");
        étudiants[nbEtudiants] = étudiant;
        nbEtudiants++;
    }
    // ...
```

Des tableaux **dans** les objets

```
    public void afficher () {
        System.out.println ("Il␣y␣a␣" + nbEtudiants + "␣étudiants");
        // Pourquoi pas un foreach ?
        for( int i=0; i<nbEtudiants; i++ ) {
            System.out.println ( étudiants[i] );
    }

    // On pourrait encore définir beaucoup de méthodes utiles
}
```

```
// Test
Groupe groupe11 = new Groupe(10);
groupe11.ajouter(new Etudiant("20000", "Tintin"));
groupe11.ajouter(new Etudiant("20001", "Milou"));
groupe11.ajouter(new Etudiant("20002", "Professeur␣Tournesol"));
groupe11.ajouter(new Etudiant("20003", "Capitaine␣Haddock"));
groupe11.afficher ();
```

Rappel

Pour qu'un langage soit *orienté objet* il doit posséder 3 propriétés

- L' **encapsulation**
- L' **héritage**
- Le **polymorphisme**

Nous avons vu l'encapsulation ; survolons le reste

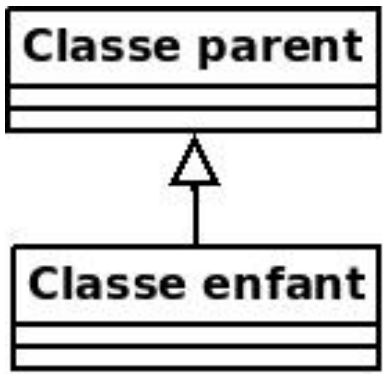
Héritage

Héritage : Permet de définir une classe à partir d'une autre

- Un peu comme du *copier-coller*
- On récupère ainsi tous les attributs et toutes les méthodes
- Terminologie
  - Classe **parent** : celle dont on hérite
  - Classe **enfant** : celle qui hérite

Héritage

- Graphiquement, on le note ainsi



- L'héritage peut se lire dans la javadoc
- Par défaut, on hérite de la classe Object

Object

Que trouve-t-on dans Object ? (cf. API)

- String toString()
  - Représentation textuelle de (l'état de) l'objet
  - Surtout à des fins de **déverminage**
  - Appelée implicitement par println
- **boolean** equals(Object o)
  - Compare 2 objets

Overriding

Java permet la réécriture (**overriding**) d'une méthode dans une classe enfant

- Le travail fait par la méthode dans la classe parent ne convient plus dans la classe enfant, je récris la méthode

Remarque

- À ne pas confondre avec l'**overloading** (la surcharge) d'une méthode

Représentation textuelle

Par défaut, l'affichage d'un objet est peu clair (utilisation de la version de toString héritée d'Object)

Exemple :

```
Etudiant ada = new Etudiant(33800, "Ada Lovelace");
System.out. println (ada);
```

affiche

```
be.heb. esi .java1 .Etudiant@19189e
```

Représentation textuelle

On peut redéfinir la méthode toString

Exemple :

```
public String toString () {
    String res = "(" + nom + ", " + numéro;
    if (ancien) {
        res = res + ", ancien";
    } else {
        res = res + ", " + annéeEtude;
        if (doubleur) {
            res = res + ", doubleur";
        }
    }
    res = res + ")";
    return res;
}
```

Représentation textuelle

L'affichage est à présent plus clair

Exemple :

```
Etudiant ada = new Etudiant(33800, "Ada Lovelace");
System.out. println (ada);
ada.aRéussi ();
System.out. println (ada);
```

affiche

```
(Lovelace,33800,1)
(Lovelace,33800,2)
```

La méthode equals()

Les objets sont des types références

$\longrightarrow$ l'opérateur == teste si c'est le **même** objet

Exemple

```
Etudiant ada = new Etudiant(33800, "Ada Lovelace");
Etudiant ada2 = new Etudiant(33800, "Ada Lovelace");
System.out. println ( ada == ada2 ); // false
```

La méthode equals permet de tester

- que les 2 objets sont dans le **même état**
- même si c'est dupliqué en mémoire

La méthode equals()

La méthode par défaut dans Object se contente de comparer les références $\longrightarrow$ besoin de la récrire

- Il faut respecter la signature
- Doit répondre « faux » si on compare à autre chose qu'un étudiant (ou **null**)

Exemple : Redéfinissons l'égalité pour les étudiants

Le polymorphisme

La signature de la méthode equals peut surprendre

```
public boolean equals(Object o) { // ...
```

- Elle attend un Object en paramètre
- On peut lui passer un Etudiant
- C'est grâce au polymorphisme

Polymorphisme : Là où on attend un objet d'une classe « parent » on peut donner un objet d'une classe « enfant »

Le polymorphisme

Si on peut recevoir n'importe quelle sorte d'objet, comment savoir ce qu'on reçoit vraiment ?

Grâce à l'opérateur **instanceof**

- Dit si un objet appartient à une classe donnée (ou un de ses enfants)
- Par définition, **null** n'est instance de rien

Notre méthode equals devient

```
public boolean equals(Object o) {
  if ( ! (o instanceof Etudiant) ) return false;
  // ...
```

Le polymorphisme

Le **casting**

- Nous savons que o est un étudiant (puisque nous faisons le test juste avant)
- Le compilateur, lui, ne le sait pas
- Le compilateur se base sur la déclaration et considère o comme un Object
- Il refuserait dès lors l'appel de méthodes propres à un Etudiant
- Le **casting** (Classe) demande au compilateur de voir l'objet comme le type enfant

La méthode equals()

Au final, on a

```
public boolean equals(Object o) {
  if ( ! (o instanceof Etudiant) ) return false;
  Etudiant autre = (Etudiant) o;
  return this.numéro == autre.numéro
      && this.nom.equals(autre.nom)
      && this.anneeEtude == autre.anneeEtude
      && this.doubleur == autre.doubleur
      && this.ancien == autre.ancien;
}
```

```
Etudiant ada = new Etudiant(33800, "Ada␣Lovelace");
Etudiant ada2 = new Etudiant(33800, "Ada␣Lovelace");
System.out.println( ada == ada2 ); // false
System.out.println( ada.equals(ada2) ); // true
```

La méthode equals()

Précision sur la méthode equals

- Si un attribut sert d'identifiant, on peut comparer seulement celui-là
- **Exemple** : un étudiant est identifié par son numéro (pas 2 étudiants de même numéro)

```
public boolean equals(Object o) {
   if ( ! (o instanceof Etudiant) ) return false;
   Etudiant autre = (Etudiant) o;
   return this.numéro == autre.numéro;
}
```

La méthode hashCode()

La documentation de equals précise qu'il faut aussi redéfinir hashCode

- Méthode liée au **hachage** (sera vu en 2ème)
- Prenons déjà la bonne habitude de la redéfinir aussi
- Facilité par la classe Objects (à ne pas confondre avec Object)

Objects : Classe offrant des méthodes statiques facilitant la manipulation des objets

La méthode hashCode()

La méthode Objects.hashCode() crée un code à partir des paramètres fournis

- Si un attribut sert d'identifiant, donner celui-là
- Sinon, donner un ensemble d'identifiants avec des valeurs fort différentes d'un objet à l'autre

Exemple : Pour un étudiant

```
public int hashCode() {
  return Objects.hashCode(this.numéro);
}
```

Plus sur Objects

Objects fournit aussi des méthodes facilitant les tests.

Exemple : Si on doit comparer 2 adresses de personnes mais que l'adresse est un attribut facultatif.

Ceci n'est pas suffisant

```
if (adresse.equals(autre.adresse)) { //...
```

On peut le rendre plus sûr en écrivant

```
if (    adresse == autre.adresse // ok si les 2 sont null
    || adresse != null && adresse.equals(autre.adresse) ) { //...
```

Objects.equals fait la même chose en plus court

```
if ( Objects.equals(adresse, autre.adresse) ) { //...
```

Une illustration du polymorphisme

Revenons un instant sur les exceptions

- On a vu qu'on peut attraper une exception en général

```
catch(Exception ex)
```

- Mais aussi en spécifiant exactement l'exception

```
catch(IllegalArgumentException ex)
```

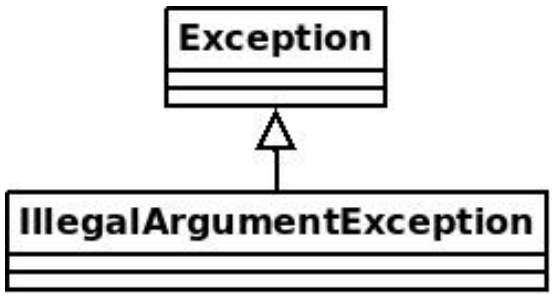
- Comment ça fonctionne ?

Une illustration du polymorphisme

- Une exception est un objet

```
throw new IllegalArgumentException("L'âge doit être positif !");
```

- Grâce à l'héritage et au polymorphisme



si on écrit Exception dans le **catch**

- on attrape IllegalArgumentException
- mais aussi d'autres exceptions $\Longrightarrow$ à éviter

Crédits

Ce document a été produit avec les outils suivants

- La distribution `Ubuntu` du système d'exploitation `Linux`
- `LaTeX` comme système d'édition
- La classe `Beamer` pour les transparents
- Les packages `listings`, `fancyvrb`, ...
- Les outils `make`, `rubber`, `pdfnup`, ...