Programmation Orientée Objet - Partie I

Par Nicolas Hurtubise

Inspiré et dérivé des notes de Mohamed N. Lokbani[1], de {Miklós Csűrös, Sébastien Roy, François Duranleau} et de Jian-Yun Nie
Ajouts de Sébastien Roy

IFT1025 - Programmation 2

[1] http://www.iro.umontreal.ca/ dift1020/cours/ift1020/communs/Cours/C2/ObjetsClasses1p.pdf

Au programme...

- Programmation orientée objet
- Classes & instances
- Constructeur
- Mot-clé `this`
- Modèle mémoire et objets
- Mot-clé `static`

Abstraction

- Abstraction = principe fondamental de la programmation

- À la base, un ordinateur n'est rien d'autre qu'un gros circuit électronique compliqué auquel on peut envoyer des signaux électriques
- Le binaire est une représentation des signaux électriques qu'on peut envoyer en guise d'instructions

```
0 : Ne pas envoyer de courant
1 : Envoyer du courant
```

Abstraction

- Le binaire est la seule forme qu'un ordinateur peut "comprendre"

Question : que fait ce programme ?

```
00111101001000000110010010000101011111110010000110111
11110011110001101100000010010010001000010111110100101
00001011111100011100111011000101110111000011000111101
10001011001010000011011101100000010111000010100000011
10101000111101100100101010110010101101000111001010101
11001011110001000110011110010011110001101000001000000
10011000000001010101001000001000101110001000100110010
11000101110111000011000111101100010110010100000110111
00000001001001000100001011111010010100001011111100011
```

Réponse (rot13) : Evra, p'rfg har fédhrapr qr ovgf nyéngbver, cnf ha ienv cebtenzzr. Yr cbvag égnag dh'ra gnag dh'uhznvaf, ba rfg vapncnoyrf qr snver yn qvsséerapr...

Abstraction

- La programmation procédurale permet d'abstraire le détail de ce que la machine doit faire pour nous permettre de focuser sur la logique de la tâche à accomplir

```
void exec(char instr) {

    if(instr == '.') {
        putc(mem[p]);
    } else {
        ...
    }

    ...
}
```

- On dispose de `variables` et de `procédures`, qui sont des abstractions de la mémoire et du processeur

Métaphore : cuisine

On cuisine une recette de *Crêpes au jambon, asperges et fromage*[2]

Programmation procédurale

- Avoir tous les ingrédients devant soi (données) :
  - Lait
  - Farine
  - Beurre
  - Sel
  - ...
- Avoir une liste d'actions à faire avec (procédures) :
  - *Mélanger le beurre et la farine*
  - *Remuer le mélange jusqu'à ébullition*
  - ...

[2] https://www.ricardocuisine.com/recettes/183-crepes-de-sarrasin-au-jambon-aux-asperges-et-au-fromage

Métaphore : cuisine

- Une recette procédurale : une liste d'étapes à faire dans l'ordre pour arriver au résultat voulu
- Un ordinateur ne comprend que les instructions simples
- Les instructions complexes sont des combinaisons d'instructions simples
  - Préparer la béchamel :
    1. Faire revenir l'oignon dans le beurre
    2. Ajouter la farine
    3. ...
- L'abstraction se fait à travers les *procédures* et les *données*
- Procédure : casser un oeuf VS monter des blancs en neige
- Données : oeuf VS béchamelle VS crêpe au jambon

Métaphore : cuisine

`Ingrédients : lait, farine, beurre, sel, jambon, ...`

`Préparer des Crêpes au jambon, asperges et fromage :`

Métaphore : cuisine

```
Ingrédients : lait, farine, beurre, sel, jambon, ...

Préparer des Crêpes au jambon, asperges et fromage :
  => Préparer les crêpes


     Préparer la béchamel


     Garnir
```

Métaphore : cuisine

```
Ingrédients : lait, farine, beurre, sel, jambon, ...

Préparer des Crêpes au jambon, asperges et fromage :
  => Préparer les crêpes
       => Fouetter [lait, farine, sel, ...]
          Couvrir et laisser reposer 30 minutes
          ...
     Préparer la béchamel
        => Faire revenir l'oignon dans le beurre
           Ajouter la farine
           ...
     Garnir
       => Placer une tranche de jambon sur une crêpe
          Verser 60ml de béchamel
          ...
```

Métaphore : cuisine

- On pourrait cependant décrire la recette d'une autre façon...
- Au lieu de lister les ingrédients ensemble et lister les procédures ensemble, on pourrait regrouper ensemble les ingrédients+procédures qui ont un sens lié

Métaphore : cuisine

```
Crêpes au jambon, asperges et fromage :
  - Ingrédients : Crêpes, Béchamel, Jambon,
                  Asperges, Fromage
  - Préparer :
     => Préparer crêpes
        Préparer béchamel
        Garnir
```

Métaphore : cuisine

```
Crêpe :
  - Ingrédients : oeuf, lait, farine, beurre...
  - Préparer :
    => Fouetter [lait, farine, sel, ...]
       Couvrir et laisser reposer 30 minutes
       ...

Béchamel :
  - Ingrédients : beurre, farine, lait, sel
  - Préparer :
    => Faire revenir l'oignon dans le beurre
       Ajouter la farine
       ...
```

Métaphore : cuisine

On constate :

- Les données sont souvent intimement liées aux opérations qu'on peut faire dessus
- Un "ingrédient complexe" comme de la béchamel peut être réutilisé directement dans d'autres recettes

Également :

- On ne réinvente pas la roue : les ingrédients et les procédures sont encore tous là
- Les éléments sont simplement structurés d'une autre façon

Programmation Orientée Objet

- En programmation *Orientée Objet*, l'abstraction dans un logiciel se fait via les *Objets*
- Motivation pour la Programmation Orientée Objet :
  - Notre façon intuitive de réfléchir en tant qu'humains
  - Nous sommes entourés d'objets qui "savent" comment se comporter
    - Votre laveuse : aucune idée de comment ça se passe à l'intérieur, mais on sait ce que ça fait et comment l'utiliser
    - On se fiche du détail de son fonctionnement !
  - Nous réfléchissons naturellement en termes d'interactions entre les objets du monde
  - Idée : refléter dans la structure du code le problème tel qu'on le conçoit dans notre tête

Programmation Orientée Objet

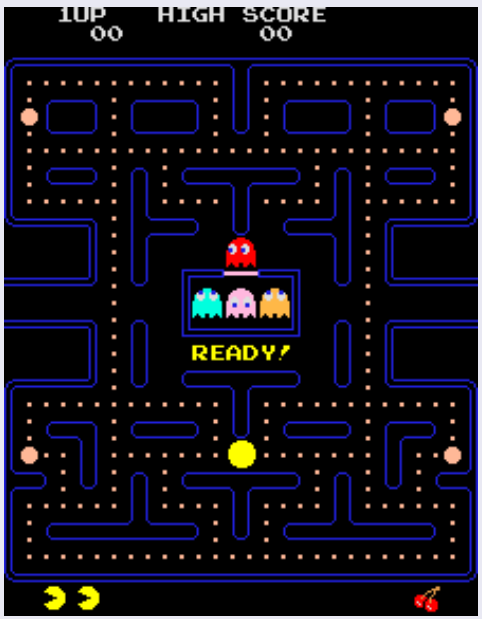
- Un "objet" de notre programme peut être n'importe quoi
  - Correspond souvent à un "objet" (tangible ou conceptuel) de la vie réelle
    - Item/personnage dans un jeu vidéo
    - Dossier étudiant
    - Compte bancaire
  - Mais pas forcément
    - Le type `String` correspond à un objet !
    - Les "objets" peuvent nous aider à organiser et structurer des concepts abstraits

Exemple d'application procédurale

Jeu de Pac-Man

Variables (état du programme) :

- Position (x, y) de Pac-Man
- Score
- Nombre de vies
- Positions des pac-gommes, super pac-gommes et fruits
- Positions (x, y) pour chacun des 4 fantômes
- ...

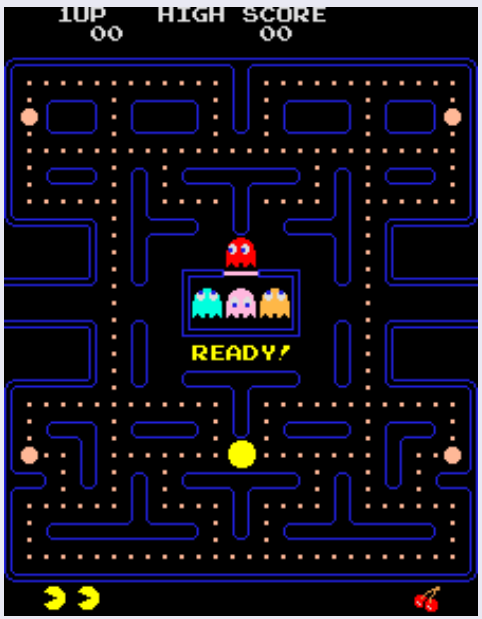


Exemple d'application procédurale

Jeu de Pac-Man

Procédures (actions possibles) :

- Bouger Pac-Man
- Manger une pac-gomme
- Manger une Super pac-gomme
- Manger un fruit
- Manger un fantôme vulnérable
- Déplacer les fantômes
- Perdre une vie
- ...



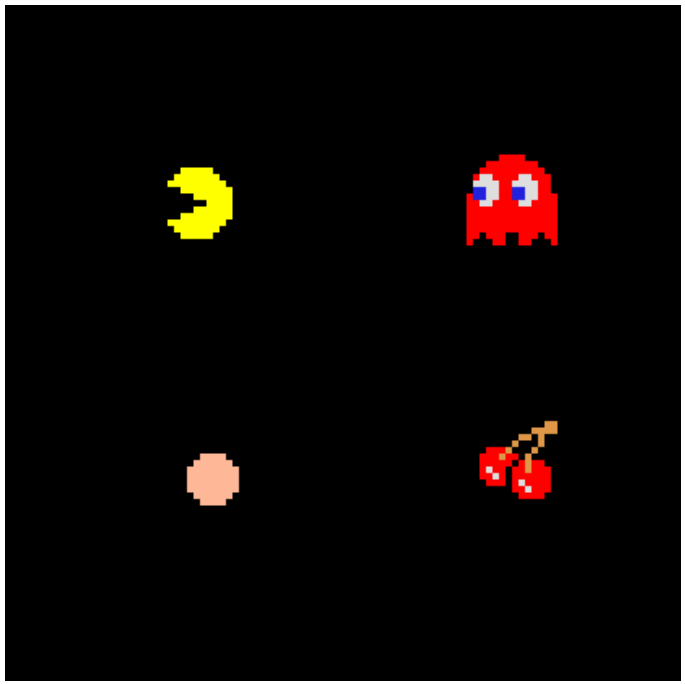
Exemple d'application orientée objet

Jeu de Pac-Man (objets)

- Pac-Man:
  - **États** :
  - Position (x, y)
  - Score
  - Nombre de vies
  - **Actions** :
  - Se déplacer
  - Manger une pac-gomme, un fruit, un fantôme...
- Fruit
  - ...
- Fantômes
  - **États** :
  - Position (x, y)
  - Vulnérable ou non
  - **Actions**
  - Se déplacer
  - Revenir à la vie
  - ...
- (Super) Pac-gomme
  - ...
- etc.

Pac-Man : procédural vs orienté objet

- Programmer en Objets peut *faciliter le découpage* du code
- Ça peut également *simplifier la réutilisation de code*



Pac-Man : procédural vs orienté objet

Mode 2 joueurs ?

Procédural :

```
int score;                =>   int score1, score2;
int nbrVies;              =>   int nbrVies1, nbrVies2;
int pacmanX, pacmanY;     =>   int pacmanX1, pacmanY1,
                                   pacmanX2, pacmanY2;
```

Pac-Man : procédural vs orienté objet

Mode 2 joueurs ?

Orienté objet :

```
PacMan joueur              =>   PacMan joueur1, joueur2;

/* La définition d'un "Pac-Man" n'a pas changé,
   on en veut seulement un deuxième ! */
```

Pac-Man : procédural vs orienté objet

Mode "party" à 16 joueurs ?

Procédural :

```
int score;                =>   int[] scores = new int[16];
int nbrVies;              =>   int[] nbrVies = new int[16];
int pacmanX, pacmanY;     =>   int[] pacmanX = new int[16];
                               int[] pacmanY = new int[16];
```

Pac-Man : procédural vs orienté objet

Mode "party" à 16 joueurs ?

Orienté objet :

```
PacMan joueur                =>    PacMan[] joueurs = new PacMan[16];
```

Programmation Orientée Objet

En programmation orientée objet :

1. On définit les objets qui constituent notre programme
2. On programme la façon dont ils vont interagir pour réaliser la tâche à faire

Programmation Orientée Objet

On doit faire la différence entre la *définition* d'un objet et les *instances* réelles

- Un cours d'université (notion abstraite)
  - Numéro de cours
  - Horaire
  - Local
  - ...
- Programmation 2 (instance réelle)
  - Numéro de cours = IFT1025
  - Horaire = Mardi et mercredi, 15h30
  - Local = 1340
  - ...

Définir une **Classe**

1. On définit les objets qui constituent notre programme

En Java, on définit ce qui constitue un objet d'un certain *type* en créant une classe.

- Une classe est un *type*
- Un type : une représentation des données + ce qui est possible de faire avec
- ex.: type `int` : les entiers, on peut faire : +,-,*,/, ...
- ex.: type `Etudiant` : un prénom, nom, matricule, ..., peut étudier, s'inscrire à des cours, ...
- Les classe sont des types aussi, mais pas des types primitifs

Définir une **Classe**

- On définit les **`attributs`** qui constituent l'**état** de notre objet

```java
// Fichier : Etudiant.java

public class Etudiant {
     public int matricule;
     public String prenom, nom;
}
```

Définir une **Classe**

- On définit les **méthodes** qui constituent les *commandes* auxquelles notre objet répond

```
public class Etudiant {
    public int matricule;
    public String prenom, nom;

    public String nomComplet() {
        return prenom + " " + nom;
    }

    public void etudier() {
        if(Math.random() < 0.01) {
            // Faire des exercices
        } else {
            // Procrastiner
        }
    }
}
```

Définir une **Classe**

Notez[3] :

1. Une classe nommée `NomDeLaClasse` *doit absolument* être définie dans un fichier nommé `NomDeLaClasse.java`
2. Chaque classe est définie dans son propre fichier
3. Chaque fichier contient une seule classe

[3] À certaines exceptions près, dans des cas plus avancés

Instancier un Objet

2. On programme la façon dont ils vont interagir pour réaliser la tâche à faire

On crée les *instances* des objets et on les utilise pour résoudre notre problème

- Un *entier* est un type, on peut avoir plusieurs instances d'un entier : 0, 10, 20,...
- Une classe est également un type, on va avoir différentes instances de ce type
- On utilise le mot-clé `new` pour *instancier* un objet

```
Etudiant nouvelEtudiant = new Etudiant();
```

Appel de méthodes

Une fois un objet instancié, on peut accéder à ses attributs et appeler ses méthodes avec la syntaxe suivante :

```
Etudiant nouvelEtudiant = new Etudiant();

nouvelEtudiant.matricule = 12365673;
nouvelEtudiant.prenom = "Jimmy";
nouvelEtudiant.nom = "Whooper";

System.out.println(nouvelEtudiant.prenom);

System.out.println(nouvelEtudiant.nomComplet());

nouvelEtudiant.etudier();
```

Appel de méthodes

L'appel :

```
nouvelEtudiant.nomComplet();
```

Va donc appeler la méthode :

```
public class Etudiant {
    ...

    public String nomComplet() {
        return prenom + " " + nom;
    }
```

Et les valeurs de `prenom` et `nom` seront celles propres à l'instance `nouvelEtudiant`

On dit que la méthode est appelée "sur" `nouvelEtudiant`

Portée des variables

Toutes les variables définies dans la classe (les attributs) sont accessibles dans les méthodes de la classe :

```
public class Etudiant {
    public int matricule;
    public String prenom, nom;

    public String identifiant() {
        return prenom.charAt(0) + nom.charAt(0) + matricule;
    }
}
```

... Ce qui peut poser problème quand on commence à avoir beaucoup d'attributs

Portée des variables

Est-ce qu'on doit faire attention de ne pas réutiliser des noms d'attributs pour les paramètres et variables de nos méthodes ?

```
public class Etudiant {
    public int matricule;
    public String prenom, nom;

    /**
     * Fonction appelée lorsque l'étudiant fait changer
     * légalement son nom
     */
    public void changementDeNom(String prenom, String nom) {
        // Mise à jour des attributs
        prenom = prenom; // ???
        nom = nom;       // ???
    }
}
```

Mot-clé `this`

On peut éviter l'ambiguité en utilisant le mot-clé `this`, qui est une référence vers *l'instance sur laquelle la méthode est appelée* :

```
public class Etudiant {
    public int matricule;
    public String prenom, nom;

    /**
     * Fonction appelée lorsque l'étudiant fait changer
     * légalement son nom
     */
    public void changementDeNom(String prenom, String nom) {
        // Mise à jour des attributs
        this.prenom = prenom;
        this.nom = nom;
    }
}
```

Constructeur

Généralement, la première chose qu'on fait quand on instancie un objet est de donner des valeurs aux attributs :

```
nouvelEtudiant.matricule = 12365673;
nouvelEtudiant.prenom = "Jimmy";
nouvelEtudiant.nom = "Whooper";
```

Constructeur

- Puisque c'est quelque chose de récurrent, on a une fonction spécialisée pour ça : le **constructeur** de l'objet
- On s'en sert pour *initialiser* les valeurs des attributs

- *Attention* : il s'agit d'une méthode spéciale
- Elle porte **le même nom que la classe** et elle n'a **pas de type de retour** (même pas `void` !)
- Cette méthode est appelée automatiquement lors d'un

```
new MaSuperClasse(...)
```

Constructeur

```
public class Etudiant {
    public int matricule;
    public String prenom, nom;

    public Etudiant(int matricule, String prenom, String nom) {
        this.matricule = matricule;
        this.prenom = prenom;
        this.nom = nom;
    }

    public void etudier() { ... }
    ...
}
```

Constructeur

```java
public class MonProgramme {

  public static void main(String[] args) {

    Etudiant nouvelEtudiant = new Etudiant(12365, "Jimmy", "Whooper");
    /* => Appelle automatiquement le constructeur
          avec les paramètres :
        12365, "Jimmy", "Whooper" */

    nouvelEtudiant.etudier();
  }
}
```

Conventions

Notez bien, par convention en Java :

- On utilise le CamelCase avec une **lettre majuscule au début** pour un nom de classe

```
public class EtudiantEtranger {
    ...
}
```

- On utilise le camelCase avec une **lettre minuscule au début** pour le nom d'une instance

```
Etudiant jimmyWhooper = new Etudiant(12365, "Jimmy", "Whooper");
```

Note pour le cours : respectez ces conventions dans vos programmes, ou ça va vous coûter des points à la correction...

Exemple : conception d'une classe

- On s'intéresse aux propriétés d'un Polygone Régulier :
  - Nombre de côtés
  - Taille d'un côté
  - Aire
  - Périmètre

Exemple : conception d'une classe

Quels sont les attributs ?

- Nombre de côtés ?
- Taille d'un côté ?
- Aire ?
- Périmètre ?

Exemple : conception d'une classe

Quels sont les attributs ?

- Nombre de côtés
- Taille d'un côté
- ~~Aire~~
- ~~Périmètre~~

On peut calculer l'aire et le périmètre à partir des deux autres paramètres

Exemple : conception d'une classe

Quelles sont les méthodes ?

- Aire
- Périmètre

Exemple : conception d'une classe

```java
// Fichier : PolygoneReg.java
public class PolygoneReg {
    public int nbrCotes;
    public double tailleCote;

    public PolygoneReg(int nbrCotes, double tailleCote) {
        this.nbrCotes = nbrCotes;
        this.tailleCote = tailleCote;
    }

    public double perimetre() {
        return nbrCotes * tailleCote;
    }

    public double aire() {
        return Math.pow(tailleCote, 2) * nbrCotes /
                    (4 * Math.tan(180.0 / nbrCotes));
    }
}
```

Exemple : conception d'une classe

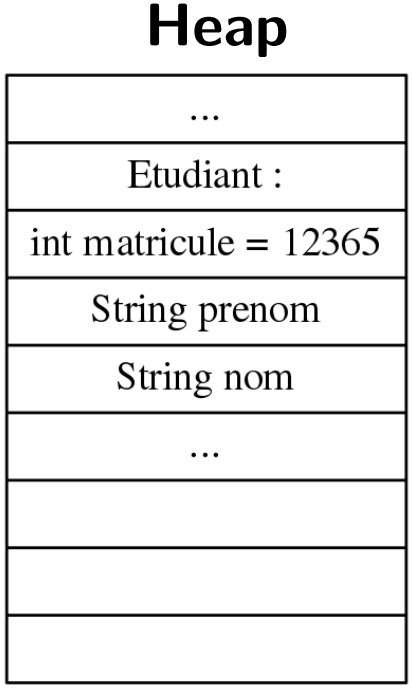
On peut tester la classe dans notre `main()` :

```java
// Fichier : MonProgramme.java
public class MonProgramme {

    public static void main(String[] args) {
        PolygoneReg carre = new PolygoneReg(4, 2.0);
        PolygoneReg hexagone = new PolygoneReg(6, 1.4);

        System.out.println(carre.aire());
        // => Affiche 4.000000000000001
        System.out.println(carre.perimetre());
        // => Affiche 8.0
        System.out.println(hexagone.aire());
        // => Affiche 5.092229374252499
    }

}
```

Comment ça se passe en mémoire ?

```
Etudiant jimmyWhooper = new Etudiant(12365, "Jimmy", "Whooper");
```

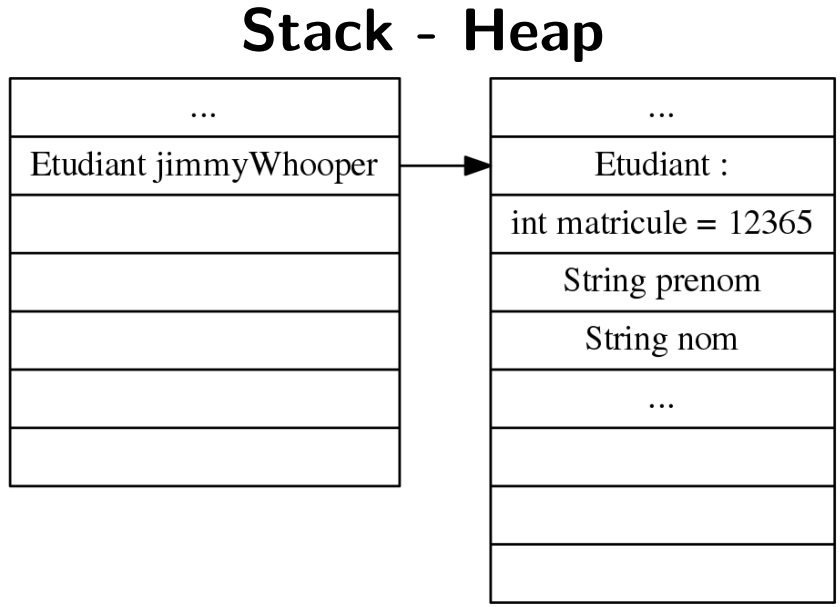
- Quand on crée une instance d'une certaine classe, on doit allouer de la mémoire quelque part pour stocker ses attributs
- `new Etudiant(...)` crée un bloc sur le Heap capable de contenir tous les attributs d'un `Etudiant`



Comment ça se passe en mémoire ?

```
Etudiant jimmyWhooper = new Etudiant(12365, "Jimmy", "Whooper");
```

- La référence à l'espace alloué en mémoire est retournée au moment du `new`. Dans cet exemple, la référence est mise dans la variable `jimmyWhooper`



Comment ça se passe en mémoire ?

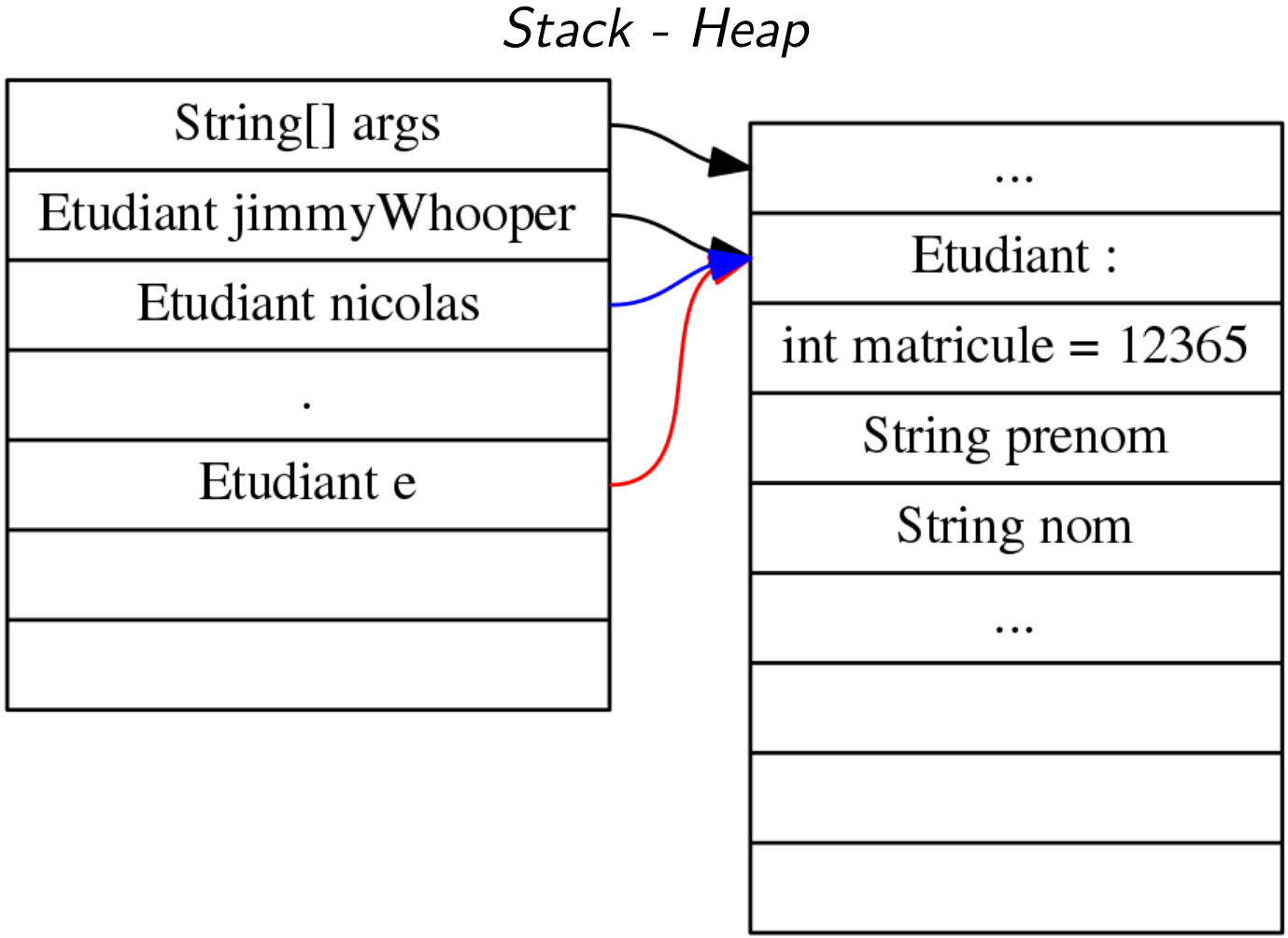
Comme pour les tableaux :

- Plusieurs variables peuvent référer à la même case en mémoire
- C'est la *référence* des objets qui est passée en paramètre lors d'appels de fonction

Comment ça se passe en mémoire ?

```
void static void main() {
    Etudiant jimmyWhooper = new Etudiant(12365, "Jimmy", "Whooper");
    Etudiant nicolas = jimmyWhooper; // Copie la référence

    fct(nicolas); // Passe la référence en paramètre
    // => Affiche "Jimmy Whooper"

    System.out.println(nicolas.matricule);
    /* => Affiche 0, c'est la référence de l'objet qui a été
          passée en paramètre, donc c'est l'objet original
          qui a été modifié dans la fonction foo */
}

public static void fct(Etudiant e) {
    System.out.println(e.nomComplet());
    e.matricule = 0;
}
```

Comment ça se passe en mémoire ?



Références `null`

C'est toujours une bonne idée d'assigner une valeur à une variable dès l'assignation

```
Etudiant jimmy = new Etudiant(12365, "Jimmy", "Whooper");

System.out.println(jimmy);
// => Etudiant@6d06d69c (affiche la référence de l'objet)

System.out.println(jimmy.getPrenom());
// => Jimmy
```

Références `null`

Mais ce n'est pas obligatoire : une variable se fait attribuer la valeur spéciale `null` tant qu'on ne lui assigne pas de référence vers une instance

```
// Crée un espace en mémoire pour une référence de type Etudiant
Etudiant jimmy;

// Mais aucune instance de type Etudiant n'est créée !

System.out.println(jimmy);
// => null

System.out.println(jimmy.getPrenom());
/* Erreur : on essaie d'accéder à une
   valeur d'une référence 'null'
   => Erreur : NullPointerException ! */
```

Tableaux d'objets

C'est ce qui se passe quand on crée un tableau d'objets :

- On crée une structure en mémoire qui peut contenir des références vers un certain type
- *Mais*, on n'instancie pas encore d'objet de ce type

```
// Crée un tableau pouvant contenir 10 étudiants
Etudiant[] inscriptions = new Etudiant[10];

System.out.println(inscriptions[0]);
// => null

System.out.println(inscriptions[0].nomComplet());
// => Erreur : NullPointerException
```

Tableaux d'objets

```java
// Crée un tableau pouvant contenir 10 étudiants
Etudiant[] inscriptions = new Etudiant[10];

/* Note : aucun étudiant n'a été créé pour l'instant
   On a seulement alloué de l'espace pour des
   *références* à des objets de type Etudiant */

System.out.println(inscriptions[4]);
// => null

// Initialisation : on crée les étudiants
for(int i=0; i<10; i++)
    inscriptions[i] = new Etudiant(i, "Prenom " + i, "Nom " + i);

System.out.println(inscriptions[4]);
// => Etudiant@7852e922

System.out.println(inscriptions[4].nomComplet());
// => Prenom 4 Nom 4
```

Le mot-clé `static`

- Parfois, des variables/fonctions peuvent être associées à des objets sans être associées à une instance en particulier
  - Ex.: attribuer un matricule aux nouveaux étudiants
  - Générer une liste alphabétique des étudiants inscrits
  - $\rightarrow$ action reliée aux étudiants, mais à aucun étudiant individuel
  - On voudrait des attributs et méthodes qui ont rapport avec *la `class` au complet*

- Solution : on peut définir des attributs statiques et des méthodes *statiques*
  - Aussi appelés *attributs de classe* et *méthodes de classe*

```
// Utilisation:
System.out.println(NomDeLaClasse.attributStatique);
System.out.println(NomDeLaClasse.methodeStatique(1, 2, 3, ...));
```

Le mot-clé static

```
public class Etudiant {
    /* Variable de classe (attribut statique) qui sert à garder
       le prochain matricule disponible */
    public static int prochainMatricule = 0;

    public int matricule; // Matricule d'un étudiant en particulier
    public String prenom, nom;

    public Etudiant(String prenom, String nom) {
        this.prenom = prenom;
        this.nom = nom;

        // Attribution automatique du matricule
        this.matricule = Etudiant.prochainMatricule;
        Etudiant.prochainMatricule++;
    }

    ...
}
```

Le mot-clé static

```java
// Fichier : Etudiant.java
public class Etudiant {
  ...
  public static void genererListeAlphabetique(Etudiant[] etudiants) {
    // Créer la liste des étudiants et la sauvegarder en .pdf
    // ...
  }
}

// Fichier : MonProgramme.java
public class MonProgramme {
  public static void main(String[] args) {
    Etudiant jeanne = new Etudiant("Jeanne", "Whooper");
    Etudiant jimmy = new Etudiant("Jimmy", "Whooper");
    Etudiant[] etudiants = new Etudiant[]{jeanne, jimmy};

    Etudiant.genererListeAlphabetique(etudiants);
  }
}
```

Le mot-clé `static`

Supposons qu'on cherche à trouver l'intersection de deux ensembles de nombres :

- Ensemble $A = \{1, 2, 3\}$
- Ensemble $B = \{3, 4\}$

On veut calculer l'intersection des deux :

- Ensemble C : $A \cap B = \{3\}$

Le mot-clé `static`

On pourrait modéliser cette situation avec une classe `Ensemble`

```
Ensemble a = new Ensemble();
a.ajouter(1);
a.ajouter(2);
a.ajouter(3);

Ensemble b = new Ensemble();
b.ajouter(3);
b.ajouter(4);

// Intersection des ensembles A et B
Ensemble c = ...
```

Le mot-clé `static`

Devrait-on utiliser une méthode statique ou une méthode d'instance ?

```
// Méthode d'instance
Ensemble c = a.intersection(b);

// Méthode statique
Ensemble c = Ensemble.intersection(a, b);
```

Les deux semblent marcher, mais on doit choisir...

Le mot-clé `static`

L'expression :

```
a.intersection(b)
```

Semble donner une *importance particulière* à l'ensemble a dans le calcul, comme si le traitement se faisait *avant tout* sur a.

Tandis que :

```
Ensemble.intersection(a, b)
```

Est simplement une fonction à deux paramètres de type `Ensemble`.

- => La fonction `static` est probablement plus appropriée dans le contexte, mais ça reste une question de style
- Le plus important est de rester consistant
  - `insersection`, `union`, ... devraient suivre le même style

Le mot-clé `static`

Si on s'était plutôt demandé :

- Ensemble $A = \{1, 2, 3, 4, 5\}$
- Ensemble $B = \{3, 4\}$

On veut retirer de A tous les éléments de B :

- $A = A - B = \{1, 2, 5\}$

Le mot-clé `static`

Dans le contexte, a va avoir une importance particulière : c'est l'objet qui va être modifié par l'opération

```
Ensemble a = new Ensemble(...); // Valeurs 1 à 5
Ensemble b = new Ensemble(...); // Valeurs 3, 4

a.retirer(b); // a contient maintenant {1, 2, 5}
```

Une fonction `static` n'aurait pas de sens ici

Le mot-clé `static`

Autres exemples (tirés des de la librairie standard de Java) :

- `Math.PI`
  - Les constantes en général sont déclarées `static`, inutile d'en avoir une version différente pour chaque instance
- `Math.sqrt()`, `Math.cos()`, `Math.sin()`, ...
  - Les fonctions mathématiques sont regroupées dans l'objet `Math`, mais une instance de `Math` n'aurait pas de sens
- `String.valueOf(123)`
- `Integer.valueOf("123")`
- ...