Introduction à la programmation objet

Pourquoi les objets?

Alice Jacquot (d'après un cours de Guillaume Wisniewski)
https://www.lri.fr/~jacquot/ipo/
septembre 2019

Université Paris Sud & LRI

Motivation n° 1 : abstraction & protection de l'information

Le paradigme impératif

- ce que l'on a fait jusqu'à présent et en L1
- élément de base : fonction
- moyen de décrire et de nommer une « méthode de calcul » *(computational process)*
  - abstraction : programmer à un plus haut niveau conceptuel ⇒ faciliter la compréhension
  - augmenter l'expressivité du langage

Exemple

```
r = n / 2;
while (abs( r - (n / r) ) > t) {
  r = 0.5 * (r + (n / r));
}
System.out.println("r = " + r);
```

versus

```
private double squareRootApproximation(double x, double epsilon)
  double approx = x / 2;
  while (Maths.abs(approx - (x / approx) ) > epsilon) {
    approx = 0.5 * (approx + (x / approx));
  }
  return approx;
}
System.out.println("sqrt(r) = " + squareRootApproximation(r, 1e-
```

source : https://en.wikipedia.org/wiki/Methods_of_computing_square_roots

Et maintenant...

- jusqu'à présent : type simple uniquement (`int`, `String`, `float`, ...)
- 2e moyen d'améliorer l'expressivité et l'abstraction : les données complexes/composées *(compound data)*

Exemple

Comment représenter (et manipuler) un rationnel ?

- $x \in \mathbb{Q} \Rightarrow x = \frac{a}{b}$ avec $(a, b) \in \mathbb{Z} \times \mathbb{Z}^*$
- règles de calcul spécifiques. Par exemple : $\frac{a}{b} + \frac{c}{d} = \frac{a \cdot d + c \cdot b}{b \cdot d}$

Les problèmes (pour les informaticiens)

- moralement : un rationnel se représente par deux `int`
  1. 1re solution : `int numerator` et `int denominator`
  2. 2e solution : `int[] rational`
- Aucune de ces deux représentations n'est adaptée :
  - comment garder/garantir l'association entre un numérateur et son dénominateur ?
  - comment garantir que numérateur et dénominateur ne seront jamais inversés ?
  - comment garantir la cohérence des données (p. ex. que le dénominateur n'est jamais nul)
  - ...

Tous ces problèmes sont exacerbés par les nouvelles contraintes sur les développements informatiques

Exemple n° 2

Rappel

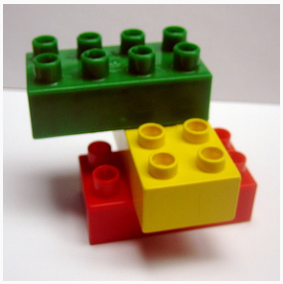
- trouver tous les triplets d'indices (i, j, k) tel que `t[i] + t[j] + t[k] = 0`
- triplets tel que $i \neq j \neq k$ et invariants par permutation

Protection de l'information

- on veut garantir dans tout le programme que (i, j, k) et (k, i, j) sont égaux
- ≪ cacher ≫ la manière dont les triplets sont implémentés ⇒ protégé cette information

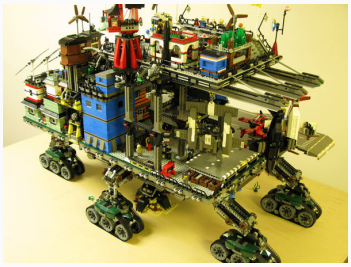
Les (nouveaux) défis de l'informatique

Programming in the Small



- ce que vous savez faire
- programme écrit par une seule personne
- peut être compris par une seule personne
- on peut le jeter / le ré-écrire entièrement dès qu'il faut :
  - le corriger / l'améliorer
  - le porter sur une nouvelle plateforme

Programming in the Large



- travail en équipe sur des projets longs et complexe
- personne ne peut connaître tous les aspects du programme
- spécifications de départ peu précises et contradictoires
- dialogue avec l'utilisateur : parler métier et non info
- correction + amélioration constante

Quelques chiffres (1)

- Évolution de la taille du noyaux Linux (millions LOC)

2001	Linux kernel 2.4.2	2,4
2003	Linux kernel 2.6.0	5,2
2009	Linux kernel 2.6.29	11,0
2009	Linux kernel 2.6.32	12,6
2010	Linux kernel 2.6.35	13,5
2012	Linux kernel 3.6	15,9
2015	Linux kernel 4.2	20,2

- Complexité de quelques « problèmes » (LOC)
  - Firefox : 10 millions
  - Large Hardron Collider : 50 millions
  - Voiture : 100 millions
  - Google : 2 milliards

Quelques chiffres (2)

Répartition des coûts de développement :

- 6% : spécification
- 5% : conception
- 7% : codage
- 15% : test et validation
- 67% : maintenance

Les erreurs proviennent de :

- exigence et spécification : 56%
- conception : 24%
- codage : 10%
- autres : 10%

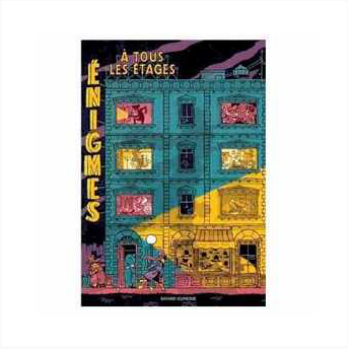
La Grande Loi de l'Informatique

- l'informatique est plus un problème social que technique
  - travailler avec d'autres développeur
  - travailler pour des besoins changeants

Quelques chiffres (3)

- 31% des développements sont abandonnés
- 91% des projets ne respectent pas les délais

Remarque



Problèmes similaires dans tous les « domaines »

Exemple : The HoTT book

`http://math.andrej.com/2013/06/20/the-hott-book/`

Le résultat

- 600 pages sur la *Homotopy Type Theory*
- $\simeq$ 40 mathématiciens
- écrit en moins de 6 mois

Comment

- méthode d'écriture = principe de développement de logiciels
- logiciel de partage de code (github)

Dans le même ordre d'idées : Open Digital Research Environment Toolkit for the Advancement of Mathematics (OpenDreamKit)

Retour à notre problème

Rappel

- représenter et manipuler des rationnels (ou des triplets)
- en garantissant la protection de l'information

Solution : encapsulation
Séparer

- la manière dont une donnée complexe est manipulée
- la manière dont une donnée complexe est construite à partir de types simples

Concrètement i

Hypothèses

- Supposons qu'il existe des fonctions :
  - `num`, `den` : accès au numérateur et au dénominateur
  - `create_rat` : création d'un rationnel
- toutes ces méthodes dépendent de la manière dont les rationnels sont représentés

Concrètement ii

Addition de deux rationnels

```
def add_rat(x, y):
  return create_rat(num(x) * den(y) + num(y) * den(x),
                    den(x) * den(y))
```

(ce n'est pas du java)

Concrètement iii

Programme principal

```
x = create_rat(2, 5)
y = create_rat(3, 13)
z = add_rat(x, y)
```

(toujours pas du java)

Les barrières d'abstraction

`Programme qui utilise des nombres rationnels`

nombres rationnels liés au problème

`add_rat, sub_rat, ...`

méthodes de manipulation ≪ haut niveau ≫

`den, num, create_rat`

méthodes de manipulation ≪ bas niveau ≫

`détails de la représentation des rationnels`

Intérêt

- plus de flexibilité : on peut changer la manière dont les données sont représentées
- plus facile à développer / maintenir : pas besoin de comprendre/connaître tous les niveaux
- plus facile à comprendre : niveau conceptuel plus élevé

Ce qu'il manque

- l'encapsulation est gérée à la main → convention de programmation
- langage orienté objet : formalisation du concept ⊕ vérification par le compilateur

Motivation n° 2 : propagation des modifications

Exemple

Contexte de l'exemple

- application de gestion des étudiants
- état civil, inscriptions, moyennes, ...
- notes de 0 à 20

⇒ données au cœur de l'application (comme souvent)

Structure de données

```
Structure Etudiant
  Dim nom As String
  Dim prenom As String
  Dim email As String
  Dim cours As List
End Structure
```


Utilisation de la structure dans le programme

Spécification
Lire les données relative à un étudiant et les afficher

```
Dim et As Etudiant
et.nom = "guillaume"

Console.ReadLine(et.nom)
Console.ReadLine(et.prenom)
Console.ReadLine(et.email)

Sub AfficheEtudiant(ByVal et As Etudiant)
  Console.WriteLine("Etudiant {1} {0}", _
                    et.nom, et.prenom)
End Sub
```

⇒ structure utilisée partout dans le programme

Changement de spécification

Proverbe pour informaticien-s-nes
≪ Programmer avec des specs, c'est comme marcher sur l'eau : c'est plus facile quand c'est gelé. ≫

À la demande du ≪ client ≫

- vérifier la validité de l'adresse mail
- gérer des noms étrangers (χατζοπούλου έλλη)
- `nom` et `prenom` avec une majuscule
- notes de A à E

Répartition des coûts de maintenances

- changement des demandes de l'utilisateur : 41,8%
- changement dans le format des données : 17,6%
- erreurs ≪ urgentes ≫ : 12,4%
- erreurs ≪ non-urgents ≫ : 9%
- changement de matériel : 6,2%
- documentation : 5,5%
- amélioration de l'efficacité : 4%
- autre : 3,4%

source : (Meyer, 2008)

Problème

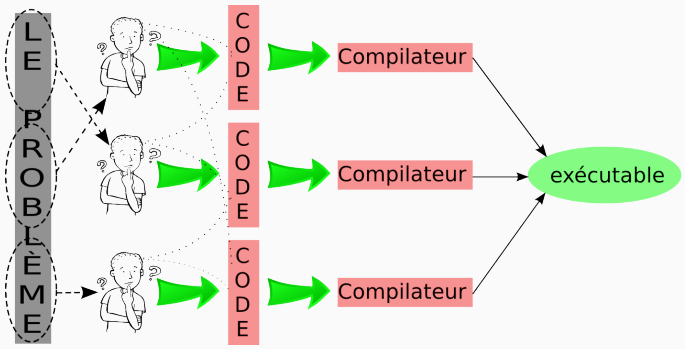
Avec une structure

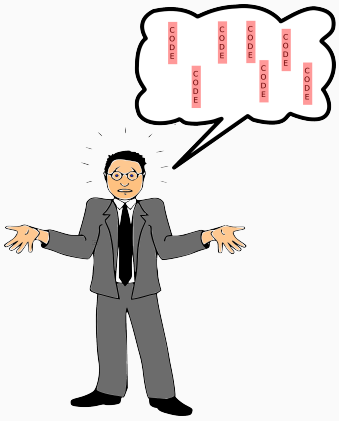
- nécessite de changer la structure
- propagation des changements ⇒ mise-à-jour de tout le code utilisant ou manipulant la structure

Conclusions

- le code ≪ marche ≫
- mais le programme n'est pas satisfaisant ⇒ difficilement maintenable

Plus généralement





Problèmes

- pas de structure
- besoin de comprendre tout le code avant de pouvoir faire une modification.

Encapsulation et programmation orientée objet

Solution apportée par la POO

Changement de point de vue
On ne définit plus une donnée composée par la manière dont elle est représentée, mais par les opérations que l'on peut effectuer sur celle-ci

En pratique
un rationnel peut être :

- additionné à un autre rationnel ;
- additionné à un entier ;
- représenté sous forme d'une `String`

Interface

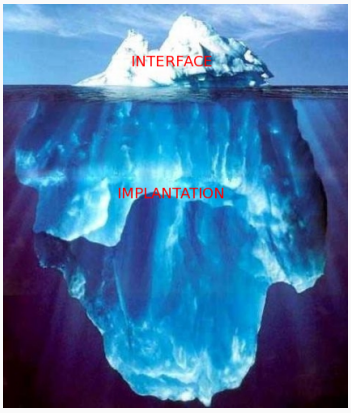
- ce que l'on vient de faire = description d'une interface
  - = ce que fait un rationnel
  - ≠ comment il le fait (implantation)
- interface = contrat = spécification

codeur du module

utilisateur

sur les services fournis par le module

Le concept d'encapsulation

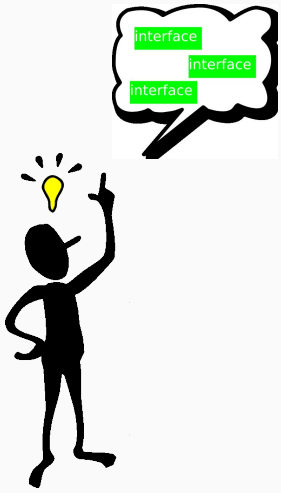


- la séparation entre interface et implantation permet de s'abstraire des détails de fonctionnement
- *information hiding*, protection de l'information, ...

Le client :

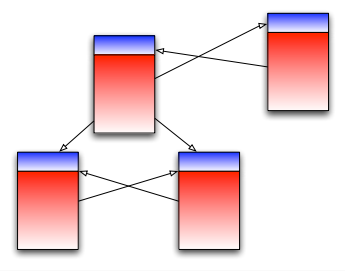
- ne manipule que l'interface
- n'a pas accès aux données / manière dont sont stockées les éléments

Intérêt (1)



- on a juste besoin de connaître les interfaces et pas toute l'implantation
- facilite la compréhension
- programmation de plus haut niveau (meilleure lisibilité)

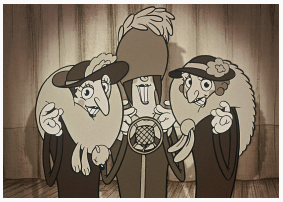
Intérêt (2)



- dépendance uniquement par rapport à l'interface
- tant que l'interface ne change pas : pas de propagation des changements

La syntaxe

Exemple : une classe pour décrire les triplets



Un triplet (d'entiers) est décrit par :

- trois entiers

On peut le manipuler en :

- fixant ses éléments ;
- en le comparant à un autre triplet.

Grand Principe

- regrouper dans une même unité syntaxique (la classe), les variables décrivant une donnée complexe et les méthodes permettant de les manipuler
- interdire tout accès (lecture/écritures) aux variables à l'extérieur de la classe

⇒ vérifiable par le compilateur

En pratique : la classe Triplet i

```java
public class Triplet {

  // les attributs
  private int x;
  private int y;
  private int z;

  // le constructeur
  public Triplet(int x, int y, int z) {
    this.x = x;
    this.y = y;
    this.z = z;
```


En pratique : la classe Triplet ii

```java
  }

  // les méthodes
  public boolean equals(Triplet t) {
    return this.x == t.x && this.y == t.y && this.z ==
  }

  public void changeValue(int x, int y, int z) {
    // ...
  }

}
```


Anatomie d'une classe (1)

```java
public class Triplet
```

- nom de classe ⇒ commence par une majuscule
- dans le fichier `Triplet.java`
- autant de fichier que de classes dans un projet
- définit une classe
  - modèle (moule) d'un objet (élément) du programme
  - abstraction ;
  - à partir de ce modèle, on peut créer autant d'objet que l'on veut ;
  - vocabulaire : objet = instance de classe ;
  - un objet représente souvent une entité du monde réel.
- pour instancier une classe (dans n'importe quelle classe du projet) :

```java
Triplet t = new Triplet(1, 2, 3);
```

On obtient alors une référence sur l'objet que l'on peut manipuler : `t.equals(t)`.

Remarque importante

Vocabulaire

- Par abus de langage : classe = objet = instance de classe
- Mais ce sont deux concepts différents

Anatomie d'une classe (2)

```java
private int x;
```

- attributs = variables attachées à une instance
- même déclaration qu'une variable, mais
  - définis tout au début d'une classe
  - précédés d'un mode d'accès (`private`)
- les attributs décrivent l'état d'une instance
- accessible uniquement depuis les méthodes de la classe

```java
public static void main(String[] s) {
  Triplet t = new Triplet(1, 2, 3);
  t.x = 19;
}
```

⇒ impossible : permet de garantir la protection des données

Anatomie d'une classe (3) : méthodes

```
public boolean equals(Triplet t)
```

- méthode = permet de manipuler l'état (=les attributs) d'un objet
- méthode ≠ fonction (pas de `static`) ⇒ une méthode est toujours rattachée à une instance
- une méthode a toujours un paramètre implicite `this`
  - `this` = référence sur l'instance courante
  - cf. appel d'une méthode : `nomReference.nomMethode(param1, param2, ...);`
  - `this` permet d'accéder aux méthodes et aux attributs de l'instance courante : `this.tempMax`

Anatomie d'une classe (4) : constructeur

```
public Triplet(int x, int y, int z)
```

- identification :
  - méthode sans type de retour
  - même nom que la classe (avec la majuscule !)
- rôle : initialiser tous les attributs de la classe
- moyen mnémotechnique : un attribut, une ligne dans le constructeur
- peut prendre des paramètres si nécessaire

Exemple d'utilisation (1)

Dans n'importe quelle classe du projet :

```
1  public static ArrayList<Triplet> findZeroSumTriplet(int[] t) {
2    ArrayList<Triplet> res = new ArrayList<>();
3    for (int i = 0; i < t.length; i++) {
4      for (int j = i + 1; j < t.length; j++) {
5        for (int k = j + 1; k < t.length; k++) {
6          if (t[i] + t[j] + t[k] == 0) {
7            res.append(new Triplet(i, j, k));
8          }
9        }
10     }
11   }
12   return res;
13 }
```


Intérêt de l'encapsulation

- On veut garantir que $(i,j) == (j,i)$ passage à des paires, pour réduire le code
- Il suffit d'implémenter la méthode `equals` correspondant :

```
1  public boolean equals(Triplet t) {
2    return ((this.x == t.x && this.y == t.y) || \
3            (this.x == t.y && this.y == t.x));
4  }
```

- On est **sûr** que tout le monde utilisera la même définition de l'égalité (seul moyen d'accéder aux attributs)

Encore mieux

- on peut choisir une représentation adaptée pour faciliter le test d'égalité :

```
1  public Pair(int x, int y) {
2    if (x < y) {
3      this.x = x;
4      this.y = y
5    } else {
6      this.x = y;
7      this.y = x;
8    }
9  }
```

- la méthode d'`equals` est triviale :

```
1  return this.x == t.x && this.y == t.y;
```

- possible car l'information est protégée : on est sûr que les triplets sont rangés par ordre croissant (une fois que l'on a modifié toutes les méthodes)

Une classe pour les étudiants

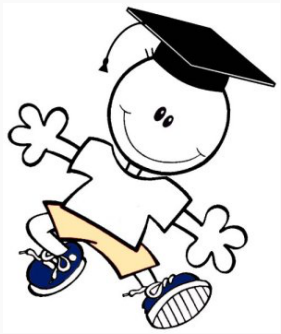
Exemple

Développer une application permettant de gérer une année de licence :

- étudiants
- cours
- notes
- diplome

1re étape : la classe `Étudiant`

Un étudiant c'est quoi ?



Caractéristiques :

- un nom
- un prénom
- une adresse mail

Actions possibles :

- envoyer un message

Classe correspondante

Rappel :

- classe = modèle (générique) d'un étudiant
- objet = une instance de classe = un « cas » particulier

De manière plus jolie :

- classe = type de donnée abstrait
- état (inconnu / inaccessible) + méthodes pour manipuler cet état

Écrire une classe : la recette magique

1. attributs
2. constructeurs
3. méthodes

Attributs

- décrivent l'état d'un objet
- juste après la définition de la classe
- précédé d'un mode d'accès : `private`
- accessible (=utilisable) uniquement dans la classe

```
public class Etudiant {

  private String nom;
  private String prenom;
  // on suppose qu'il existe une classe Email
  private Email email;
```

Les constructeurs

- initialise un objet
- une ligne par attribut
- il peut y avoir plusieurs constructeurs

```
public Etudiant(String nom, String prenom, Email email)
  this.nom = nom;
  this.prenom = prenom;
  this.email = email;
}
```

Rappel :

- utilisation du `this` pour accéder aux attributs

Intérêt POO : protection de l'information

- l'initialisation se fait toujours par l'intermédiaire d'un constructeur
- = quelque chose que l'on contrôle
- garantir que l'objet est toujours dans un état « cohérent »
- contrainte : tous les noms sont en majuscule et les prénoms en minuscule, sauf la première lettre

```
public Etudiant(String nom, String prenom, Email email)
  this.nom = nom.toUpperCase();
  this.prenom = prenom.substring(0, 1).toUpperCase()
       + prenom.substring(1).toLowerCase();
  this.email = email;
}
```

⇒ impossible de modifier `nom` et `prenom`

Les méthodes

- comme les fonctions habituelles
- permettent de modifier/interroger l'état

Par exemple :

```
public void sendMessage(String msg) {
  this.email.send(msg)
}
```


Intérêt POO : pas de propagation des modifications

- arrivée du 21e siècle : plus personne n'utilise de mail
- modification de la méthode `sendMessage` (p.ex. : message facebook/google talk)
- ou plus simplement : la classe `Email` change
- mais pas de modification du code qui utilise la méthode `sendMessage`
- dépendance avec l'interface (ce que l'on fait) et non avec l'implémentation (comment on le fait)

Complément sur les méthodes

Les méthodes standards

- tous les objets sont créés avec certaines méthodes standards
- techniquement : méthodes héritées de la classe `Object`
  - cf. la javadoc de la classe `Object`
- comportement par défaut qu'il faut généralement redéfinir

La méthode `equals` (1)

- teste l'égalité entre deux objets
- par défaut : égalité entre les références :

```
String a = new String("chat");
String b = new String("chat");
```

  les objets `a` et `b` ne sont pas égaux au sens des références
- exemple : deux étudiants sont égaux s'ils ont même nom et même prénom

La méthode `equals` (2)

```
@Override
public boolean equals(Object o) {
  if (!(o instanceof Etudiant)) {
    return false;
  }

  Etudiant oe = (Etudiant) o;
  return oe.nom.equals(this.nom) && oe.prenom.equals(th
}
```

- `Object` n'importe quelle référence (quelque soit sa classe)
- toujours le même principe
  1. vérifier si l'objet et de bonne classe
  2. le convertir
  3. faire la comparaison

La méthode `toString`

- retourne une représentation de l'objet sous forme d'une chaine de caractères
- appeler automatiquement, par exemple par la méthode `System.out.println`
- par défaut : nom de la classe + adresse mémoire de l'objet

Exemple :

```
@Override
public String toString() {
  return this.nom + " " + this.prenom;
}
```


2e étape : utiliser la classe `Etudiant`

Instanciation

Rappel :

- classe = modèle, objet = instance, cas particulier
- création d'un objet = instanciation :

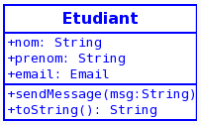
```
Etudiant e1 = new Etudiant("WiSniEwski",
                           "guillaume",
                           "pouet@pouet.com");
Etudiant e2 = new Etudiant("allauzen",
                           "alex",
                           "pouet2@pouet.com");
```

- utilisation d'un objet :

```
e1.sendMessage("coucou");
System.out.println(e1);
```


Que se passe-t-il ?

1. classe = modèle = pas de données

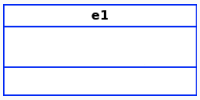


Que se passe-t-il ?

1. classe = modèle = pas de données
2. instanciation :

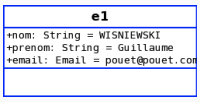
Que se passe-t-il ?

1. classe = modèle = pas de données
2. instanciation :
   - réserve la mémoire



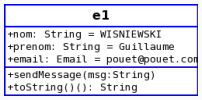
Que se passe-t-il ?

1. classe = modèle = pas de données
2. instanciation :
   - réserve la mémoire
   - initialise les attributs



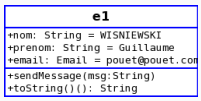
Que se passe-t-il ?

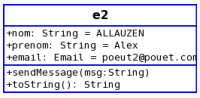
1. classe = modèle = pas de données
2. instanciation :
   - réserve la mémoire
   - initialise les attributs
   - ≪ attache ≫ les méthodes à ces attributs (chaque instance de méthode ne peut modifier que ses attributs)



Que se passe-t-il ?

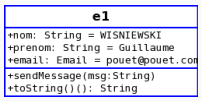
1. classe = modèle = pas de données
2. instanciation :
   - réserve la mémoire
   - initialise les attributs
   - ≪ attache ≫ les méthodes à ces attributs (chaque instance de méthode ne peut modifier que ses attributs)

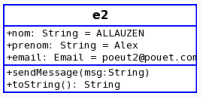




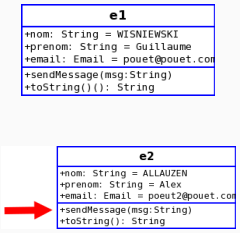
Que se passe-t-il ?

1. classe = modèle = pas de données
2. instanciation :
   - réserve la mémoire
   - initialise les attributs
   - ≪ attache ≫ les méthodes à ces attributs (chaque instance de méthode ne peut modifier que ses attributs)
3. appel d'une méthode



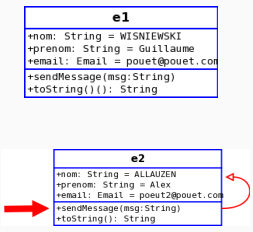


Que se passe-t-il ?



1. classe = modèle = pas de données
2. instanciation :
   - réserve la mémoire
   - initialise les attributs
   - « attache » les méthodes à ces attributs (chaque instance de méthode ne peut modifier que ses attributs)
3. appel d'une méthode
   - envoie d'un message à l'objet

Que se passe-t-il ?



1. classe = modèle = pas de données
2. instanciation :
   - réserve la mémoire
   - initialise les attributs
   - « attache » les méthodes à ces attributs (chaque instance de méthode ne peut modifier que ses attributs)
3. appel d'une méthode
   - envoie d'un message à l'objet
   - la méthode peut accéder à l'état de celui-ci

Les choses qu'il faut retenir

- objet = données + méthodes attachées
- méthodes = moyen de manipuler l'état d'une instance particulière

Cas particulier

- méthode statique = méthode qui ne peuvent pas accéder à l'état d'un objet
- déclaration :

```
public static int maMethode(...)
```

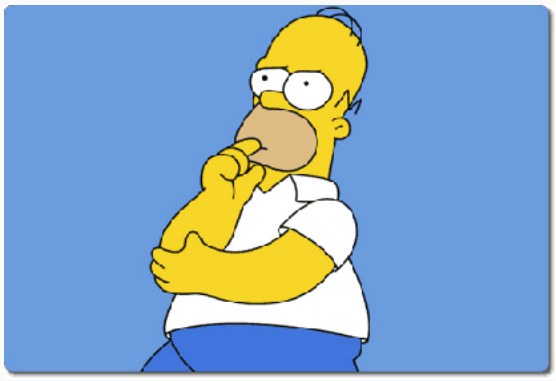
- impossible d'accéder aux attributs (erreur de compilation)
- en pratique : méthode « partagée » entre toutes les instances
- intérêt : méthode outils (`Integer.parseInt`, `Math.pow`, ...), « simuler » les fonctions

3e étape : les cours

Spécification d'un cours

- un cours a un titre + un professeur responsable
- possibilité d'inscrire des étudiants s'il reste de la place
- nombre d'étudiants max = 20, sauf cas particuliers
- chaque étudiants a des notes
- coefficient + notes = moyenne

Les attributs



Les attributs

Les cas évidents :

```
private String nom;
private String nomResponsable;
```

Les autres :

```
private int nombreEtudiantsMax;
private ArrayList<Etudiant> etudiants;
private HashMap<Etudiant, ArrayList<Integer>> notes;
private ArrayList<Integer> coef;
```

⇒ tous les éléments nécessaire à la description de l'état d'un objet

Représentation des notes et des coefficients

- deux éléments à décrire :
  1. association étudiant–liste de notes
  2. association note–coefficient
- 1re décrite explicitement (tableau associatif)
- 2e décrite implicitement : `note[i]` ↔ `coef[i]`

⇒ choix de conception arbitraire

Intérêt de la POO

- encapsulation = cache ce choix
- on peut manipuler l'objet/classe sans savoir quelle modélisation a été faite.
- compréhension ⊕ facile
- choix de conception peut être modifié

Dans le même ordre d'idée...

Pourquoi stocker les coefficients par des entiers ?

Dans le même ordre d'idée...

Pourquoi stocker les coefficients par des entiers ?

⇒ on s'en fout, c'est la responsabilité de la personne qui code la classe

Constructeur i

Attributs :

```
private String nom;
private String nomResponsable;
private int nombreEtudiantsMax;
private ArrayList<Etudiant> etudiants;
private HashMap<Etudiant, ArrayList<Integer>> notes;
private ArrayList<Integer> coef;
```


Constructeur ii

```
public Cours(String nom, String rNom, int etuMax) {
  this.nom = nom;
  this.nomResponsable = rNom;
  this.nombreEtudiantsMax = etuMax;
  this.etudiants = new ArrayList<Etudiant>();
  this.notes = new HashMap<Etudiant, ArrayList<Integer>
  this.coef = new ArrayList<Integer>>();
}
```


Problème...

- valeur par défaut pour la taille des groupes
- pour le moment : obligation de la spécifier à chaque instanciation

⇒ concept de surcharge

Surcharge de fonction

Fonction surchargée =

- 2 fonctions avec le même nom
- même type de retour
- mais le nombre/type des arguments est différent
- machine virtuelle choisie la fonction à appeler en fonction des paramètres

Exemple :

```
// retourne x ** a
public double pow(double x, int a)
// retourne 10 ** a
public double pow(int a)
```

⇒ utilisation la plus courante = valeur par défaut pour les paramètres

Pour notre constructeur i

Définition de deux constructeurs :

```
public Cours(String nom, String rNom, int etuMax) {
  this.nom = nom;
  this.nomResponsable = rNom;
  this.nombreEtudiantsMax = etuMax;
  this.etudiants = new ArrayList<Etudiant>();
  this.notes = new HashMap<Etudiant, ArrayList<Integer>
  this.coef = new ArrayList<Integer>>();
}

public Cours(String nom, String rNom) {
  this.nom = nom;
```


Pour notre constructeur ii

```
  this.nomResponsable = rNom;
  this.nombreEtudiantsMax = 20;
  this.etudiants = new ArrayList<Etudiant>();
  this.notes = new HashMap<Etudiant, ArrayList<Integer>
  this.coef = new ArrayList<Integer>>();
}
```

De manière plus succincte

```
public Cours(String nom, String rNom, int etuMax) {
  this.nom = nom;
  this.nomResponsable = rNom;
  this.nombreEtudiantsMax = etuMax;
  this.etudiants = new ArrayList<Etudiant>();
  this.notes = new HashMap<Etudiant, ArrayList<Integer>
  this.coef = new ArrayList<Integer>>();
}

public Cours(String nom, String rNom) {
  this(nom, rNom, 20);
}
```


Avec un peu de sucre syntaxique

```
public Cours(String nom, String rNom, int etuMax) {
  this.nom = nom;
  this.nomResponsable = rNom;
  this.nombreEtudiantsMax = etuMax;
  this.etudiants = new ArrayList<Etudiant>();
  this.notes = new HashMap<Etudiant, ArrayList<Integer>
  this.coef = new ArrayList<Integer>>();
}

public Cours(String nom, String rNom) {
  // appel au constructeur principal
  this(nom, rNom, 20);
}
```


Où en est-on ?

- un cours a un titre + un professeur responsable
- possibilité d'inscrire des étudiants s'il reste de la place
- nombre d'étudiants max = 20, sauf cas particuliers
- chaque étudiants a des notes
- coefficient + notes = moyenne

Ajout d'un étudiant

```
public boolean addEtudiant(Etudiant e) {
  if (this.etudiants.size() >= this.nombreEtudiantsMax)
    return false;
  }

  this.etudiants.add(e);
  return true;
}
```

- seul moyen de modifier la liste des étudiants
- la contrainte imposée par le nombre maximum d'étudiants sera toujours respectée

Méthode pour savoir si un étudiant à valider le cours

```
public boolean validate(Etudiant e) {

  ArrayList<Integer> note = this.notes.get(e);
  if (note == null) {
    // étudiant inconnu : lancer une exception
  }

  double moyenne = 0.0
  for (int i = 0; i < note.size(); i++) {
    double c = (double) this.coef.get(i) / 100.0;
    moyenne += c * note.get(i);
  }

  return moyenne >= 10;
}
```


Almost there : la classe **Annee**

Spécification

- regroupe tout ce qui est relié à une année :
  - liste des étudiants
  - liste des cours
- « interface » principale du programme
  - partie qui va gérer tous les objets
  - « boucle principale »

Attributs + constructeurs

```java
public class Annee {
  private ArrayList<Etudiant> etudiants;
  private ArrayList<Cours> cours;

  public Annee() {
    this.cours = new ArrayList<Cours>();
    this.etudiants = new ArrayList<Etudiant>();
  }
```


Méthodes

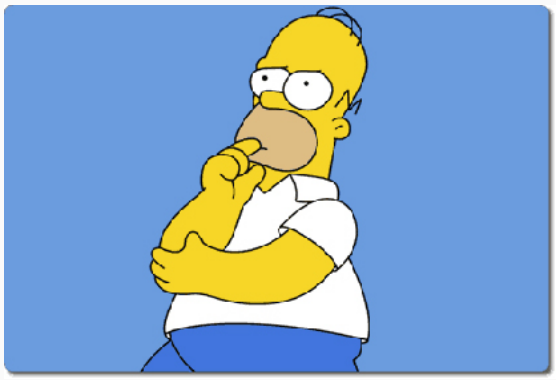
ce que l'on ne fera pas :

- ajout/suppression d'un cours
- ajout/suppression d'un étudiant
- inscription

ce que l'on fera :

- calculer la moyenne générale d'un étudiant

Calcul de la moyenne (1)



Calcul de la moyenne (2)

Impossible

(dans la situation actuelle)

- notes = attributs privés de la classe `Cours`
- pas accessibles depuis une autre classe
- protection de l'information : elles sont toujours « stockées » avec les coefficients

Calcul de la moyenne (3)

Comment faire :

- se demander si c'est vraiment nécessaire
  - information validé/pas validé suffisante ?
- ajouter une méthode moyenne à la classe `Cours`
- on casse l'encapsulation : que se passe-t-il si les notes ne sont plus des `Integer` mais des `Char` ?