LU2IN002
PREMIER OBJET

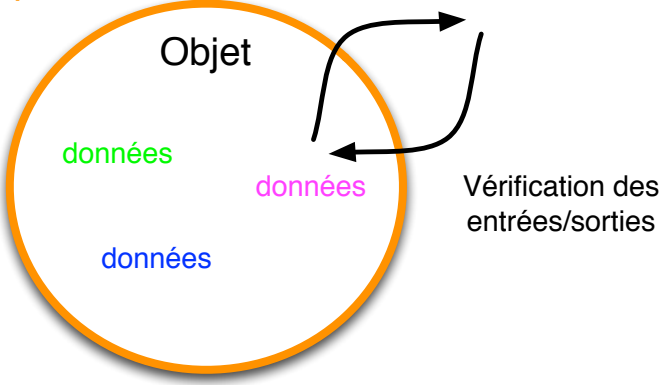
Vincent Guigue & Christophe Marsala

- Diviser un programme complexe en **objets**:
  - objet autonome: réutilisable dans plusieurs projets
    - Vecteurs, Personne, MoteurGraphique...
  - objet sécurisé: garantie de bon usage par d'autre
    - un objet intègre des *données* et des *méthodes* pour les manipuler proprement,
    - les transactions bancaires sont journalisées, les éléments d'une simulation physique ne se téléportent pas...
  - objet simple & intuitif: le client ne voit que ce qui est nécessaire
- Enjeux:
  - Réfléchir en amont au découpage & à la sécurisation
  - Documenter le code (en premier lieu, respecter les conventions pour faciliter la compréhension)

Barrière de sécurisation

=

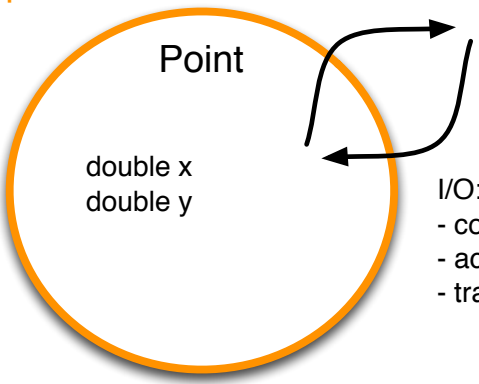
encapsulation



Barrière de sécurisation

=

encapsulation

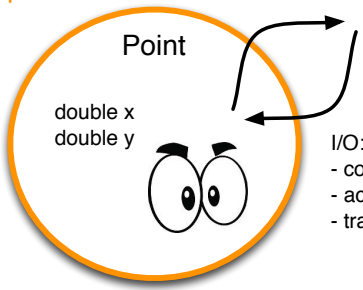


Point de vue fournisseur :

Barrière de sécurisation

=

encapsulation



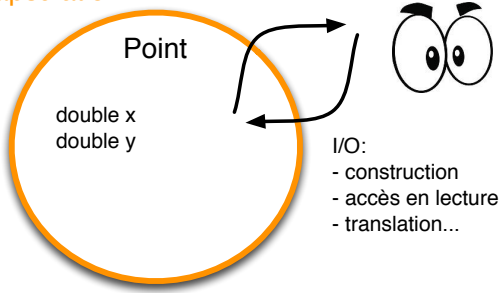
- Nouveau fichier, nouvelle classe
  - `Point.java`
- Comment construire un objet?
  - e.g. J'attends 2 valeurs réelles
  - je mets à jour `x` et `y`
- Comment établir un dialogue (IO)?
  - je définis une méthode pour observer `x`
  - je réfléchis à la manière de modifier `x` et `y`
    - Accès direct
    - Addition, soustraction, ...

Point de vue client :

Barrière de sécurisation

=

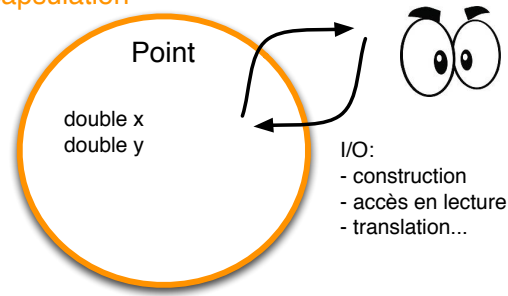
encapsulation



- Nouveau type de variable
  - `int i; Point p;`
- Comment construire un objet?
  - Je donne 2 valeurs réelles + syntaxe
  - `p = new Point(1.2, 3.1);`
- Comment dialoguer?
  - J'utilise le `.` pour accéder à l'interface de l'objet:
  - `p.getX();`

Point de vue client :

Barrière de sécurisation
=
encapsulation



- Nouveau type de variable
  - `int i; Point p;`
- Comment construire un objet?
  - Je donne 2 valeurs réelles + syntaxe
  - `p = new Point(1.2, 3.1);`
- Comment dialoguer?
  - J'utilise le `.` pour accéder à l'interface de l'objet:
  - `p.getX();`

⇒ De l'extérieur, je ne me demande pas comment marche l'objet

Trivial sur un `Point`... Mais imaginons la même chose avec une classe `Image`

1. Fichier: 1 classe = 1 fichier
   - nom de la classe + .java = nom du fichier
   - marquer l'encapsulation, faciliter la ré-utilisation
   - le nom de classe commence par une majuscule

```
1 // Creation du fichier Point.java
```

Syntaxe: signature et attributs

1. Fichier: 1 classe = 1 fichier
   - nom de la classe + .java = nom du fichier
   - marquer l'encapsulation, faciliter la ré-utilisation
   - le nom de classe commence par une majuscule
2. Signature: une classe est toujours `public` (en LU2IN002)

```
1 // Creation du fichier Point.java
2 public class Point{ // classe publique
```

Syntaxe: signature et attributs

1. Fichier: 1 classe = 1 fichier
   - nom de la classe + .java = nom du fichier
   - marquer l'encapsulation, faciliter la ré-utilisation
   - le nom de classe commence par une majuscule
2. Signature: une classe est toujours `public` (en LU2IN002)
3. Déclaration des attributs:
   - Répondre à : *De quoi est composé notre objet?*
   - Les attributs sont presque toujours `private` (cf plus loin)
   - nom des attributs en minuscules

```
1 // Creation du fichier Point.java
2 public class Point{ // classe publique
3     private double x,y; // attributs prives
```

Syntaxe: signature et attributs

1. Fichier: 1 classe = 1 fichier
   - nom de la classe + .java = nom du fichier
   - marquer l'encapsulation, faciliter la ré-utilisation
   - le nom de classe commence par une majuscule
2. Signature: une classe est toujours `public` (en LU2IN002)
3. Déclaration des attributs:
   - Répondre à : *De quoi est composé notre objet?*
   - Les attributs sont presque toujours `private` (cf plus loin)
   - nom des attributs en minuscules
4. Définir des méthodes
   - Comment construire un objet?
   - Quelles opérations effectuer sur l'objet?
   - nom des méthode en minuscules

```
1 // Creation du fichier Point.java
2 public class Point{ // classe publique
3     private double x,y; // attributs prives
```

Comment construire un `Point`?
⚠ Attention à considérer le problème des 2 points de vues

Fournisseur

- *Besoin*: 2 coordonnées fournies en argument
- *Action*: affectation des valeurs des attributs

```
1 //Fichier Point.java
2 public class Point{
3   private double x,y;
4
5   public Point(double x2, double y2){
6     x = x2;
7     y = y2;
8   }
9 }
```

Comment construire un `Point`?
⚠ Attention à considérer le problème des 2 points de vues

Fournisseur

- *Besoin*: 2 coordonnées fournies en argument
- *Action*: affectation des valeurs des attributs

```java
//Fichier Point.java
public class Point{
  private double x,y;

  public Point(double x2, double y2){
    x = x2;
    y = y2;
  }
}
```

Client

ie: utilisateur d'une classe Point existante...

- *Besoin*: créer une instance dans la mémoire avec les bonnes valeurs

```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main
                    (String[] args){
```

Comment construire un `Point`?
⚠ Attention à considérer le problème des 2 points de vues

Fournisseur

- *Besoin*: 2 coordonnées fournies en argument
- *Action*: affectation des valeurs des attributs

```java
//Fichier Point.java
public class Point{
  private double x,y;

  public Point(double x2, double y2){
    x = x2;
    y = y2;
  }
}
```

Client

ie: utilisateur d'une classe Point existante...

- *Besoin*: créer une instance dans la mémoire avec les bonnes valeurs

```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main
                    (String[] args){
    // appel du constructeur
    // avec des valeurs choisies
    Point p = new Point(2., 3.1);
  }
}
```

- `public` : accessible / visible depuis l'extérieur de l'objet (eg : main, autre objet...)
- `private` : protégé / invisible depuis l'extérieur de l'objet

- Constructeurs = `public` en général: ils sont appelés depuis l'extérieur
- Attributs = `private` en général, ils sont protégés et non accessibles depuis l'extérieur

Fournisseur

```java
//Fichier Point.java
public class Point{
  private double x,y;

  public Point(double x2, double y2){
    x = x2;
    y = y2;
  }
}
```

Client

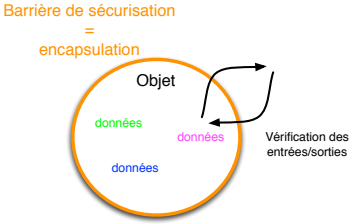
```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main
                    (String[] args){

    // opération autorisée
    Point p = new Point(2., 3.1);

   // opération impossible :
   // ERREUR DE COMPILATION
   double d = p.x;
  }
}
```

Comment manipuler un `Point`?

1. **définir des méthodes**
   eg: accéder aux attributs (en lecture)
2. les **invoquer** depuis l'extérieur



Fournisseur

```java
//Fichier Point.java
public class Point{
 private double x,y;
 public Point(double x2, double y2){
    x = x2;
    y = y2;
 }

 public double getX(){ return x; }

 }
```

Client

```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main
                    (String[] args){
    // construction d'un point:
    Point p = new Point(2., 3.1);

    double px = p.getX();

    System.out.println(
          "coord x de p:"+px);
  }
}
```

Syntaxe: surcharge du constructeur

Comment construire un `Point`? ... de plusieurs manières !

Ex:

- 2 valeurs à fournir: le plus classique
- 0 valeur: génération aléatoire de `x` et `y`
- 1 valeur: affectation de la même valeur pour `x` et `y`

Syntaxe: surcharge du constructeur

Comment construire un `Point`? ... de plusieurs manières !

Ex:
- 2 valeurs à fournir: le plus classique
- 0 valeur: génération aléatoire de `x` et `y`
- 1 valeur: affectation de la même valeur pour `x` et `y`

⇒ Syntaxe triviale : il suffit de définir plusieurs constructeurs

Fournisseur

```
1 //Fichier Point.java
2 public class Point{
3   private double x,y;
4   public Point(double x2, double y2){
5     x = x2;  y = y2;
6   }
```

Syntaxe: surcharge du constructeur

Comment construire un `Point`? ... de plusieurs manières !

Ex:
- 2 valeurs à fournir: le plus classique
- 0 valeur: génération aléatoire de `x` et `y`
- 1 valeur: affectation de la même valeur pour `x` et `y`

⇒ Syntaxe triviale : il suffit de définir plusieurs constructeurs

CONTRAINTE: signatures des constructeurs différentes

Fournisseur

```java
//Fichier Point.java
public class Point{
  private double x,y;
  public Point(double x2, double y2){
    x = x2;  y = y2;
  }
  public Point(double d){ x = d; y = d;}

  public Point(){ // autre surcharge
    // aleatoire entre 0 et 10
    x = Math.random()*10;
    y = Math.random()*10;
  }
}
```

Client

```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main(String[] args){

    // construction d'un point:
    Point p = new Point(2., 3.1);
    // construction d'un autre point:
    Point p2 = new Point();
    // construction d'un 3e point:
    Point p3 = new Point(4.2);
  }
}
```

Créer un point avec deux coordonnées *a* et *b*:

1. le `main` / le client doit donner *a* et *b*: $\Rightarrow$ on ne peut pas inventer les valeurs !
2. La classe `Point` / le fournisseur doit avoir prévu de recevoir deux valeurs

ATTENTION: informatique $\neq$ infomagique

Créer un point avec deux coordonnées *a* et *b*:

1. le `main` / le client doit donner *a* et *b*: $\Rightarrow$ on ne peut pas inventer les valeurs !
2. La classe `Point` / le fournisseur doit avoir prévu de recevoir deux valeurs

L'exécution d'un programme n'est pas magique non plus:

1. on peut suivre l'exécution ligne par ligne
2. on peut comprendre les choix de la machine
   $\Rightarrow$ il ne peut pas y avoir deux méthodes qui s'appellent pareil

Créer un point avec deux coordonnées *a* et *b*:

1. le `main` / le client doit donner *a* et *b*: $\Rightarrow$ on ne peut pas inventer les valeurs !
2. La classe `Point` / le fournisseur doit avoir prévu de recevoir deux valeurs

L'exécution d'un programme n'est pas magique non plus:

1. on peut suivre l'exécution ligne par ligne
2. on peut comprendre les choix de la machine

$\Rightarrow$ il ne peut pas y avoir deux méthodes qui s'appellent pareil

$\Rightarrow$ Du bon sens et encore du bon sens !!!!

Faîtes quelques erreurs de syntaxe [mais pas trop]...

mais n'écrivez pas n'importe quoi !

- Les méthodes standards existent sur tous les objets (cf cours héritage)...

Mais le comportement n'est pas satisfaisant

Parce que l'infomagique n'existe pas

`equals`, `toString`, ...

- Ex: conversion d'un objet en chaine de caractères `public String toString()`

Fournisseur

```
1 //Fichier Point.java
2 public class Point{
3   ...
4
5 }
```

Client

```
1  //Fichier TestPoint.java
2  public class TestPoint{
3    public static void main
4                       (String[] args){
5      // construction d'un point:
6      Point p = new Point(2., 3.1);
7
8      String str = p.toString();
9      System.out.println("p:" + str);
10   }
11 }
```

```
» p:  Point@8764152
```

- Les méthodes standards existent sur tous les objets (cf cours héritage)...

Mais le comportement n'est pas satisfaisant

Parce que l'infomagique n'existe pas

`equals`, `toString`, ...

- Ex: conversion d'un objet en chaine de caractères `public String toString()`

Fournisseur

```
1 //Fichier Point.java
2 public class Point{
3  ...
4  public String toString(){
5     return "["+ x +","+ y +"]";
6  }
7 }
```

Client

```
1  //Fichier TestPoint.java
2  public class TestPoint{
3    public static void main
4                       (String[] args){
5      // construction d'un point:
6      Point p = new Point(2., 3.1);
7
8      String str = p.toString();
9      System.out.println("p:" + str);
10   }
11 }
```

```
» p:  [2, 3.1]
```

Exemple type: addition entre 2 `Point`

Réfléchir à la signature d'une méthode `add` permettant d'additionner 2 instances de `Point` (en retournant une nouvelle instance dont les coordonnées sont les sommes respectives des `x` et `y` des attributs des opérandes)

$\Rightarrow$ Quelle signature de méthode pour `add`? Argument(s) / retour?

Exemple type: addition entre 2 `Point`

Réfléchir à la signature d'une méthode `add` permettant d'additionner 2 instances de `Point` (en retournant une nouvelle instance dont les coordonnées sont les sommes respectives des `x` et `y` des attributs des opérandes)

Client

```
1  //Fichier TestPoint.java
2  public class TestPoint{
3    public static void main
4                      (String[] args){
5
6      // construction d'un point:
7      Point p  = new Point(2., 3.1);
8      Point p2 = new Point(0.5, 1);
9
10     Point p3 = p.add(p2);
11   }
12 }
```

Exemple type: addition entre 2 `Point`

Réfléchir à la signature d'une méthode `add` permettant d'additionner 2 instances de `Point` (en retournant une nouvelle instance dont les coordonnées sont les sommes respectives des `x` et `y` des attributs des opérandes)

Client

```java
//Fichier TestPoint.java
public class TestPoint{
  public static void main
                    (String[] args){

    // construction d'un point:
    Point p  = new Point(2., 3.1);
    Point p2 = new Point(0.5, 1);

    Point p3 = p.add(p2);
  }
}
```

Fournisseur

```java
//Fichier Point.java
public class Point{
 private double x,y;
 public Point(double x2, double y2){
    x = x2;
    y = y2;
 }

 public Point add(Point p){
   return new Point(x+p.x, y+p.y);
 }
```

Syntaxe objet = un truc à prendre... Pas évident au début !

- Même nom de fonction, arguments différents
- Le type de retour ne compte pas

```java
public class Point {
...
    public void move(double dx, double dy){
        x+=dx; y+=dy;
    }
    public void move(double dx, double dy, double scale){
        x+=dx*scale; y+=dy*scale;
    }
    public void move(int dx, int dy){
        x+=dx; y+=dy;
    }
    public void move(Point p){
        x+=p.x; y+=p.y;
    }
```

Rappel: dans la classe `Point`, vous avez accès aux attributs privés des autres instances de `Point`

Nous avions vu précédemment comment compiler et executer *UNE* classe... Comment faire maintenant qu'il y en a plusieurs?

```
» javac Point.java
» javac TestPoint.java
» java TestPoint
```

Syntaxe réduite:

```
» javac Point.java TestPoint.java OU javac *.java
» java TestPoint
```

Remarque: il peut y avoir plusieurs `main`

(mais pas plus de 1 par classe)

- Compilation de tous les `main` d'un coup
- Execution d'un seul (appel à la classe correspondante)

Cas ambigus: `this.`

Un argument de méthode et un attribut portent le même nom: il faut les distinguer!

Exemple le plus classique: le constructeur

```
1 public class Point{
2  private double x,y; // attributs
3  public Point(double x, double y){ // arguments du constructeur
4    // distinguer l'attribut et l'argument
5    //pour faire l'affectation dans le bon sens
```

Un argument de méthode et un attribut portent le même nom: il faut les distinguer!

Exemple le plus classique: le constructeur

```
1 public class Point{
2  private double x,y; // attributs
3  public Point(double x, double y){ // arguments du constructeur
4    // distinguer l'attribut et l'argument
5    //pour faire l'affectation dans le bon sens
6    this.x = x; // utiliser l'argument pour init. l'attribut
7    this.y = y;
8  }
9 }
```

Vous avez toujours le droit d'utiliser la syntaxe `this.attr` pour désigner l'attribut `attr` dans la classe (même dans les cas non ambigus)

Vous pouvez utiliser `this.maMethode()` pour invoquer `maMethode` lorsque vous êtes dans une autre méthode de l'objet.

Constructeur multiple: usage du `this()`

Approche standard pour écrire plusieurs constructeurs dans une classe:

1. Ecrire le constructeur général, prenant le plus d'arguments
2. *Appeler* le constructeur ① avec des arguments spécifiques

$\Rightarrow$ éviter les copier-coller, fiabiliser le code

```
1  public class Point{
2   private  double x,y; // attributs
3
4   public Point(double x, double y){ // constructeur 1
5     this.x = x;
6     this.y = y;
7   }
8
9   public Point(){ // constructeur 2
10  // ATTENTION : aucun code avant this()
11    this(Math.random()*10, Math.random()*10); // invocation
12                                              // du constructeur 1
13  }
14 }
```