Programmation Objet
Java – Objets

Michail Lampis
michail.lampis@dauphine.fr

Développement de logiciel

- Problème de logiciel
  - Fiabilité
  - Coût (développement, maintenance)
  - Taille
- Solution : **Modularité**
  - « Programmation par contrat »
  - Facilite la collaboration
  - Certification
  - Réutilisation du code

Histoire

- Modularité =
  - Fonctions / Procédures (Fortran, C, Pascal)

    Années ‘60/’70
  - Typage de données (Rappel:**struct**)
  - Modules (Ada, ‘80)
  - Objets (C++, Java, ... ‘80/’90)
    - Classe, Héritage, Polymorphisme,...

Classes – Structures

- Rappel : **struct** de C/C++
  - Permets de définir un nouveau type de données, composé par plusieurs champs
  - Ex :

    struct Point { double x,y ; } ;

    struct Point p1,p2 ; p1.x=2.0 ;

    …

Classes – Structures

- Motivation :
  - Emballer ses données dans une struct
    - Rendre le code plus lisible, claire
    - Utiliser le système de typage pour trouver des erreurs
      - **struct A { int x;} ;**
        struct B { int x;} ;
        A,B sont des types différents.

Classes – Structures

- Classe = Structures + Méthodes
  - Structure = Champs, Données contenues dans un objet de la classe
  - Méthodes = fonctions associés avec la classe, qui peuvent manipuler les données

La classe Point

```
class Point {
    double x,y ;
    double distanceFromZero() ;
    //returns sqrt(x²+y²)
    void move(double dx, double dy) ;
    ...
}
```

- Attn : méthodes ne sont pas **static**

Structures vs Classes

- Ex (C):

```
struct Point {
    double x,y ;
    double (*distanceFromZero)(double x,    double y) ;
}
```

- Vs (Java/C++)

```
class Point {
    double x,y ;
    double distanceFromZero(double x,   double y) ;
}
```

Structures vs Classes

```
struct Point a,b ;//OK

a.x = 2.3 ; b.x=3.4 ; //OK, valeur de x indépendante dans
                                    //chaque « objet »

double f(double x, double y){return x+y;}

double g(double x, double y){return x-y;}

a.distanceFromZero = f ;

b.distanceFromZero = g ; //OK!!!  (en C)
```

Classes

- Classe =
  - Structure de données
    - Chaque objet de la classe contient des champs avec les mêmes noms/types
    - Valeur indépendante dans chaque objet
  - Manipulation de données
    - Chaque objet contient les mêmes méthodes : même noms/types/**implementation**

Classes de Java

- Quand on définit une classe (ex : **Point**)
  - On peut utiliser son nom comme un type

    ex : **Point p ; Point [] pArray ;**
- Type **non-primitif ! →** Variables = référence
- Pour utiliser un objet d'une classe il faut l'allouer

  p = new Point();

  pArray = new Point[10] ;

Gestion de Mémoire

- Types primitifs : variables sont alloué sur le **stack** (pile d'appel de fonctions)

  ex :

  void f(){int f1,f2 ; g() ; ..}

  void g(){int g1,g2 ; f() ; ..}

- **f()** alloue f1,f2, g1,g2,f1,f2,g1,g2,…
- Mémoire est libérée quand une fonction finit son exécution.
  - Libération en ordre inverse de l'allocation (stack – LIFO)

Gestion de Mémoire

- Types Objet : variables sont des références
  - Variables contiennent l'adresse de mémoire où un objet de leur type est stocké.
  - Cet mémoire est indépendante de la pile d'appel.

    Ex :

    Point f() { Point p = new Point() ;

    return p;}
  - Attn : l'objet de p existe après la fin de f !

La Classe Point

- Pour démontrer les idées basiques des classes en Java on va implémenter une classe pour représenter un point en 2D

```
class Point{
    double x,y ;
    double distance(Point p) ;
    ...
}
```

- Idée : donner la structure des données et les méthodes qui les manipulent ensemble.

Définir une méthode

```
class Point {
        double x,y;
        double distance(Point p){
          return Math.sqrt((x-p.x)*(x-p.x)+
                 (y-p.y)*(y-p.y));
        }
}
```

Utiliser la classe

```
class Point {
        double x,y;
        double distance(Point p){
          return Math.sqrt((x-p.x)*(x-p.x)+(y-p.y)*(y-p.y));
        }
        public static void main(String [] args){
                Point p = new Point();
                Point p2 = new Point();
                p.x=1.0; p2.y=1.0;
                System.out.println(p.distance(p2));
        }
}
```

Points à retenir

- Allocation des objets : **new**
  - Point p,p2 ; p.distance(p2) ;// ???
  - Erreur de compilation / Null Pointer !
- On utilise l'opérateur . pour accéder à l'intérieur (champs/méthodes) d'un objet
- Méthode == fonction dont le premier argument est implicit
  - **p.distance(p2)** == **distance(p,p2)** en C

Scanner

- Rappel

  Scanner s = new Scanner() ;

  int x = s.nextInt() ;

- Exemples de construction d’un objet de la classe **Scanner** et de l’appel d’une méthode sur cet objet.

Classes

- Le but de concevoir une classe :
  - Construire une partie du programme (ex : une structure de données)
  - indépendante et **réutilisable.**
- Ex :
  - On peut écrire un programme qui utilise la classe **Scanner**
  - Comment utiliser **Point** dans un autre programme ?

Packages

- Pour donner à l'utilisateur l'opportunité d'utiliser la classe **Point**
- On commence la définition par :

  package myPoint;

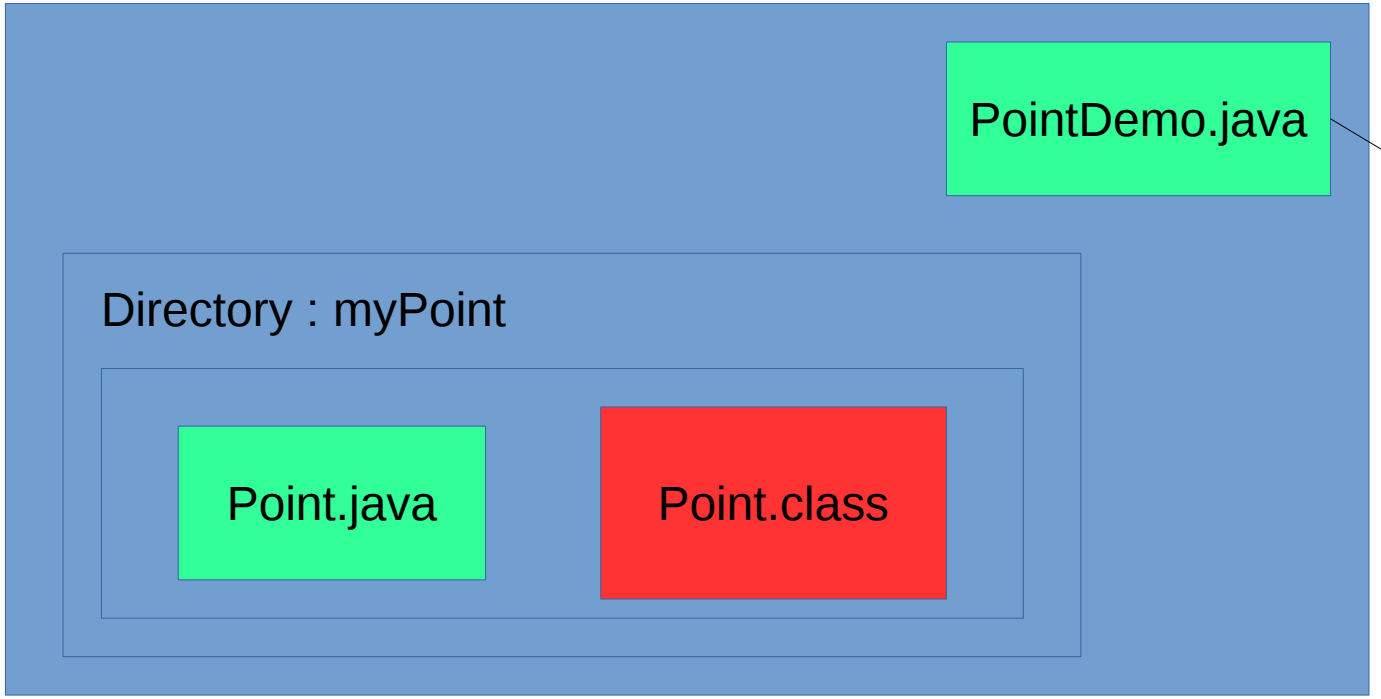
  public class Point {…
- **package** : collection des classes
- **public** : classe accessible en dehors du package
- On supprime la fonction **main**

Packages

- On met les fichiers Point.java, Point.class dans un répertoire **myPoint**
- Maintenant, dans le répertoire qui contient **myPoint** on peut écrire une classe qui fait

  import myPoint.Point ;
- ...et puis utilise la classe Point.

Structure d’un Package



Peut utiliser la classe Point après
import myPoint.Point ;

Packages

- Pourquoi cette organisation ?
  - La structure des packages nous permets de séparer les parties différentes d'un grand projet de manière **claire** et bien-définie.
  - On peut construire une hiérarchie

    Ex : **java.util…**
- Facilite la collaboration, réutilisation du code

Encapsulation

- Principe no. 1 de l'OOP
- Une fois une classe/un package défini il faut séparer ses spécifications de son implémentation.
- Motivation :
  - Les programmes extérieures qui utilisent la classe doivent rester valides après modification
  - Mais, le code de la classe peut évoluer

Encapsulation

- Deux rôles
  - Développeur de la classe
  - Développeur qui utilise la classe
- Il faut que les deux restent indépendantes
  - L’utilisateur peut-il utiliser la classe sans connaître son implémentation ?
  - Si l’implémentation change ?

Encapsulation – Java

- Java nous permets **d'imposer** l'encapsulation.
  - Quand on programme une classe, on peut définir quelles sont les parties accessibles.
  - Attn : On promets de maintenir leur fonctionnalité, même si la classe évolue !!
  - Mot clé : **public**

PointDemo

Ne compile pas !!!

```
import myPoint.Point;
class PointDemo{
        public static void main(String [] args){
                Point p = new Point();
                Point p2 = new Point();
                p.x=1.0; p2.y=1.0;
                System.out.println(p.distance(p2));
        }
}
```

PointDemo

- Le programme ne compile pas, bien qu'on ait fait **import** la classe **Point**.
  - Cause : on a utilisé le champ **x**.
  - Par défaut le mode d'accès d'un champ est **package private**

Modificateurs d’accès

- Quatre types :
  - **public:** accessible à tous
  - **private:** accessible **que** dans la classe
  - (rien) : package private, accessible dans le même package
  - **protected :** accessible dans le package et les sous-classes (on verra plus tard)

PointDemo

- Pour que le programme PointDemo compile
  - Soit on met PointDemo dans le même package (et répertoire).
    - Mauvaise idée en générale
  - Soit on déclare **x,distance** comme **public**
    - Mauvaise idée (implémentation non-encapsulée)

Encapsulation

- Pourquoi on ne déclare pas tous les champs comme **public** ??
- Point v2 : coordonnées polaires

```
public class Point{
    double dist, angle ;
    double distance(Point p){…} ;
}
```

Encapsulation

- Nouvelle implémentation
  - (meilleure ? Oui, dans certains cas)
- On veut que les programmes qui utilisent la classe fonctionnent sans rien changer.
  - Mais **x** n’existe plus !!
- Conseil : exposer le minimum possible, pour se donner la liberté de faire évoluer la classe.
- Conseil : par défaut, mettre **private** partout.

Getters/Setters

- Si on avait adopté d'abord la définition

```
public class Point{
    private double x,y ;
    public double getX() { return x;}
    public double getY() { return y;}
    public void setX(double newX) {
        x = newX;}
        …
}
```

Getters/Setters

- Une telle implémentation ne complique pas (beaucoup) un programme utilisateur :

  p.getX() ; à la place de **p.x ;**

- Mais elle nous donne une marge pour manœuvrer :
  - Ex : si on adopte les coordonnées polaires, on peut redéfinir les fonction get/set, pour qu'elles calculent les coordonnées x,y correctes à partir de **dist, angle.**

Getters/Setters

- Toujours une bonne idée ?
  - Pas forcément. Parfois du code répétitif sans sens.
- Utiliser votre jugement
  - Quelle est la partie de votre implémentation qui va rester toujours inchangé ?
  - On signe un « contrat » avec les utilisateurs de la classe !

Classes – Initialisation

- Si l'interne de notre classe est caché, comment un utilisateur peut-il initialiser un objet ?
- **Constructeur :**
  - Une méthode spéciale, appelée avec **new**, qui initialise un nouveau objet
  - Attn : doit être public (sinon, aucun sens...)

Constructor

```
class Point {
    private double x,y ;
    Point(double myX,myY) {
        x=myX ; y=myY ; }
        //accès permis:je suis dans la classe
}
```

- Utiliser comme :

  Point p = new Point(2.0,3.1) ;

  - Même en dehors de la classe

Constructor

- Méthode qui ne retourne rien
- A le même nom que la classe
- Si pas défini :
  - Default constructor (n’initialise rien)
- Surcharge possible
  - Plusieurs versions avec des paramètres différents.

Constructor

```
class Point {
    private double x,y ;
    Point(double myX,myY) {
        x=myX ; y=myY ; }
        //accès permis:je suis dans la classe
    Point() { x=0.0 ; y=0.0;}
}
```

- Utiliser comme **Point p = new Point() ;**

Constructor

```
class Point {
    private double x,y ;
    Point(double myX,myY) {
        x=myX ; y=myY ; }
        //accès permis:je suis dans la classe
    Point() { this(0.0,0.0) ;}
}
```

- Utiliser comme **Point p = new Point() ;**

Le mot clé this

- Deux sens
  - Dans un constructor, on peut avoir, **comme la première ligne**, un appel du type **this(..) ;**
  - C’est un appel pour un autre constructor (selon le type de paramètres)

Le mot clé this

- Deux sens
  - Dans une méthode, **this** est une référence pour l'objet principale de l'appel
  - Ex :

    public void f() { this.x = 2.2;}

    ..

    Point p =.. ;

    p.f() ; //this == p pour cet appel

Le mot clé this

- Constructor

```
public Point(double myX,myY) {
    x=myX ; y=myY ; }
```

- Ou

```
public Point(double x,y) {
    this.x=x ; this.y=y ; }
```

  - Sans ambiguïté...

Static

- Par défaut, chaque champ d'une classe est associé avec un objet de la classe quand on l'utilise
  - **p.x ;** //le champ x de cet objet
  - **p.distance(p2) ;**//appelle distance sur cet //objet
- On peut aussi définir des champs communs pour tous les objets d'une classe

Static

- Ex : compteur

```
public class A{
    public static int compt = 0 ;
    //initialisé la première fois qu'on utilise la classe
    public A() { compt++;}
}
A a1 = new A() ;
A a2 = new A() ;
A.comp == 2
```

- Pour accéder à un champ static, on utilise le nom de la classe
  - Pas d'objet spécifique !

Static

- Méthode **static**
  - Fonction qui ne prend pas comme argument implicite un objet

    class A{ static void f(){..} }

    A.f() ; //ou simplement **f()** dans A

Accès – Méthode static

```
class A{
   int x ;
   static int y ;
   void f() { x=1 ; y=2 ; }
   static void g() { x=1 ; y=2 ; }
}
```

Accès – Méthode static

```
class A{
    int x ;
    static int y ;
    void f() { x=1 ; y=2 ; }
    static void g() { x=1 ; y=2 ; }
}
```

- **x** de quel objet ?
- **f()** est bonne.

Final

- Mot clé **final:** un champ qui ne peut pas être modifié après l'initialisation (hors constructor)

```
class A {
    final int x ;
    public A() { x=2 ; } //OK
    void f() { x = 3 ; } // !!!
}
```

Final

- Souvent utilisé avec **static** pour définir une constante.

```
class A {
    static final int x = 17 ;
    ..
    //A.x == 17 toujours
}
```

- Ex : Math.PI == 3.14159...

Initialisation

- Si l'initialisation des champs est simple, on peut donner des valeurs par défaut :

```
class A{
    int x=1 ;
}
```

- Précédence :
  - 1. valeurs par défaut partout (0, null,…)
  - 2. initialisation directe
  - 3. constructor

Surcharge

- On a vu qu'on peut surcharger le constructor
- On peut surcharger toute méthode
  - Type de paramètre doit différencier les versions

```
class A{
    int f(int x){..}
    int f(double x){..} //OK
    double f(int x){…} // !!
}
```

Surcharge

- Evitez les appels ambiguës !
- **class A{**

  int f(int x, double y){..}

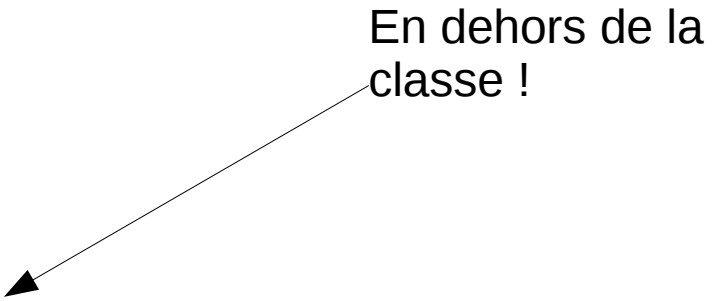
  int f(double x, int y){..} //OK, but...

  ..

  a.f(2,3) ; // ????

  }

Call-by-ref

- Toutes les variables de type objet sont des références :
  - → Une méthode peut changer (indirectement) ses paramètres

```
class A{
    private int x = 1 ;
    void setX(int x){this.x=x;}
}
..
void f(A a){ a.setX(3) ; }
void g(){ A a=new A() ; f(a) ; /* a.x==3 */}
```