

Inf 431 – Cours 7

Modularité

Programmation par objets (1/2)

jeanjacqueslevy.net

secrétariat de l'enseignement:
Catherine Bensoussan
cb@lix.polytechnique.fr
Aile 00, LIX,
01 69 33 34 67

www.enseignement.polytechnique.fr/informatique/IF

Plan

1. Modularité
2. Paquetages et espace des noms
3. Contrôle d'accès
4. Programmation par objets
5. Sous-classes et conversions de type
6. Types disjonctifs et sous-classes

Programmes et abstraction (1/3)

- choix des structures de données
- isoler les fonctionnalités
- séparer les entrées/sorties du corps du programme
- simplicité de la signature des fonctions
- élégance de l'écriture
- découpage d'un programme en classes
- Spécification ⇒ Programmation ⇒ Correction
- Génie logiciel (*Software engineering*)
- La programmation est une des activités les plus complexes jamais entreprises par l'homme.
 - Windows XP = 50 millions lignes de code,
 - Linux = 30 millions lignes de code,
 - autres exemples.

Programmes et abstraction (2/3)

- décomposition d'un programme en modules
- un module a deux parties :
 - l'interface accessible depuis l'extérieur du module
 - l'implémentation accessible de l'intérieur du module
- principe d'encapsulation
 - Une file a 3 opérations : construction, ajouter, supprimer
 - Deux implémentations possibles : tableau circulaire, liste
 - L'extérieur n'a pas à connaître l'implémentation
 - On peut changer l'implémentation sans que l'extérieur ne s'en rende compte
- on peut composer les modules pour en faire de plus gros

Programmes et abstraction (3/3)

- structure de bloc dans les langages de programmation
 - ⇒ variables **locales** ou **globales**
 - ⇒ encapsulation
- **insuffisant**, car les variables communes à plusieurs fonctions deviennent globales
 - ⇒ accessibles depuis l'extérieur
- ⇒ **constructions spéciales** dans les langages de programmation
 - modules ou classes
 - données et fonctions privées ou publiques

Modularité en Java (2/4)

```
public static void ajouter (int x, FIFO f) {
    if (f.pleine)
        throw new Error ("File Pleine.");
    f.contenu[f.fin] = x;
    f.fin = (f.fin + 1) % f.contenu.length;
    f.vide = false; f.pleine = f.fin == f.debut;
}

public static int supprimer (FIFO f) {
    if (f.vide)
        throw new Error ("File Vide.");
    int res = f.contenu[f.debut];
    f.debut = (f.debut + 1) % f.contenu.length;
    f.vide = f.fin == f.debut; f.pleine = false;
    return res;
}
```

Modularité en Java (1/4)

Considérons l'exemple des files d'attente.

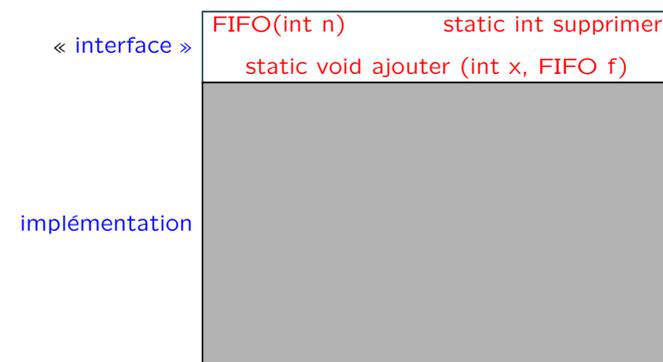
Interface **publique** : FIFO, ajouter et supprimer.

Implémentation **privée**

```
public class FIFO {
    private int debut, fin;
    private boolean pleine, vide;
    private int[] contenu;

    public FIFO (int n) {
        debut = 0; fin = 0; pleine = n == 0; vide = true;
        contenu = new int[n];
    }
}
```

Modularité en Java (3/4)



L'« interface » public est visible, l'implémentation est opaque.

En Java, le mot-clé `interface` a une autre signification. Dans certains langages de programmation (Modula, ML), il y a une distinction entre interfaces et implémentations, permettant la compilation séparée.

Modularité en Java (4/4)

```
public class FIFO {
    private Liste debut, fin;
    public FIFO (int n) { debut = null; fin = null; }

    public static void ajouter (int x, FIFO f) {
        if (f.fin == null) f.debut = f.fin = new Liste (x);
        else {
            f.fin.suivant = new Liste (x);
            f.fin = f.fin.suivant;
        }
    }

    public static int supprimer (FIFO f) {
        if (f.debut == null) throw new Error ("File Vide.");
        else {
            int res = f.debut.val;
            if (f.debut == f.fin) f.debut = f.fin = null;
            else f.debut = f.debut.suivant;
            return res;
        }
    }
}
```

⇒ **Mêmes signatures** pour les variables et fonctions publiques dans les deux implémentations (constructeur compris).

Espace des noms (2/4)

- L'instruction `package id1.id2...idk;`
qualifie les noms des champs publics d'une unité de compilation
- exemples de noms qualifiés :
 - `ma_lib1.FIFO`, `ma_lib1.FIFO.ajouter`, `ma_lib1.FIFO.supprimer` (premier paquetage)
 - `ma_lib2.FIFO`, `ma_lib2.FIFO.ajouter`, `ma_lib2.FIFO.supprimer` (deuxième paquetage)
- on référence ces champs publics avec leurs **noms qualifiés** (classe Test)
- ou avec une **forme courte** si on importe la classe avant son utilisation
`import id1.id2...idk.C;`
où *C* est un nom de **classe** ou le symbole ***** pour importer toutes les classes d'un paquetage.

Espace des noms (1/4)

- En Java, structures de données = objets (ou tableaux).
⇒ Beaucoup de classes, de fichiers .class,
⇒ risques de **collisions** dans l'espace des noms.

- les classes sont regroupées en **paquetages**.

```
package ma_lib1;           ou           package ma_lib2;
public class FIFO {...}    public class FIFO {...}
```

- On importe un paquetage pour l'utiliser :

```
import ma_lib1.FIFO;
class Test {
    public static void main (String[ ] args) {
        int n = Integer.parseInt (args[0]);
        FIFO f = new FIFO (n);
        for (int i = 1; i < args.length; ++i)
            if (args[i].equals ("-"))
                System.out.println (FIFO.supprimer(f));
            else {
                int x = Integer.parseInt (args[i]);
                FIFO.ajouter (x, f);
            }
    }
}
```

Espace des noms (3/4)

- L'emplacement des paquetages dépend de deux paramètres :
 - le **nom** `id1.id2...idk` qui désigne l'emplacement `id1/id2/.../idk` dans l'arborescence des répertoires du **système de fichiers**.
 - la **variable d'environnement** `CLASSPATH` qui donne une suite de racines à l'arborescence des paquetages.
- forme **compressée** des répertoires : fichiers `.jar` (*java archives*). Ils contiennent une arborescence de paquetages en un seul fichier (cf. la commande Unix `jar`). Les fichiers `.zip` sont aussi utilisables.
- La valeur de `CLASSPATH` est fixée par une commande Unix :
`setenv CLASSPATH " :$HOME/if431/DM1:/users/profs/info/chassignet/Jaxx"`
(pour `csh`, `tcsh`) ou
`export CLASSPATH=" :$HOME/if431/DM1:/users/profs/info/chassignet/Jaxx"`
(pour `sh`, `bash`).

Espace des noms (4/4)

- Un fichier `.java` démarre souvent comme suit :
`package ma_lib1;`
`import java.util.*;`
`import java.io.*;`
`import java.awt.*;`

Cette unité de compilation fera partie du paquetage `ma_lib1` et importe les paquetages :

- `java.util` qui contient des classes **standards** pour les tables, les piles, les tableaux de taille variable, etc.
- `java.io` qui contient les classes d'**entrées-sorties**.
- `java.awt` qui contient les classes **graphiques**.
- `java.math` qui contient les classes **mathématiques**.
- Cf. la liste des paquetages sous la rubrique *Les classes de Java* dans la page web du cours.
- Si aucun paquetage précisé, l'unité de compilation fait partie du paquetage **anonyme** localisé dans le **répertoire courant**.

Contrôle d'accès (2/2)

Pour les membres d'un paquetage, il y a 2 types d'accès :

- une **classe publique** peut être accédée de l'extérieur de son paquetage
- une classe sans qualificatif n'est accessible que depuis son paquetage
- de l'extérieur du paquetage, double-protection : classe + champ doivent être publics
- **Remarque** : le *class loader* ne vérifie pas cette discipline, car il accède à la méthode `main` sans que sa classe ne soit publique !
- **une seule** classe publique *C* par unité de compilation
⇒ une seule classe publique par fichier *C.java*
(Certains compilateurs sont laxistes. Mais mieux vaut privilégier la portabilité)

Contrôle d'accès (1/2)

Pour les membres d'une classe, il y a 3 types d'accès :

- **public** pour permettre l'accès depuis toutes les classes.
- **private** pour restreindre l'accès aux seules expressions ou fonctions la classe courante.
- **par défaut** pour autoriser l'accès depuis toutes les classes du même paquetage.

Exercice 1 Montrer qu'on accède facilement aux champs ou fonctions des classes compilées dans le même répertoire.

Programmation par objets (1/3)

- programmation procédurale \simeq Fortran, Algol, Pascal, C, Ada, ML, Caml, Haskell, programmation fonctionnelle
- programmation dirigée par les données = programmation par objets, Simula, Smalltalk, C++, Eiffel, Java

	modification des données	modification du programme
programmation procédurale	changement global	changement local
programmation par objets	changement local	changement global

- Dans un univers où les données sont proches les unes des autres, la programmation par objets peut être avantageuse.
- L'Exemple est une boîte à outils graphique, où on ne code que des **incréments**.

Programmation par objets (2/3)

Un autre argument est souvent avancé par les pro-objets :

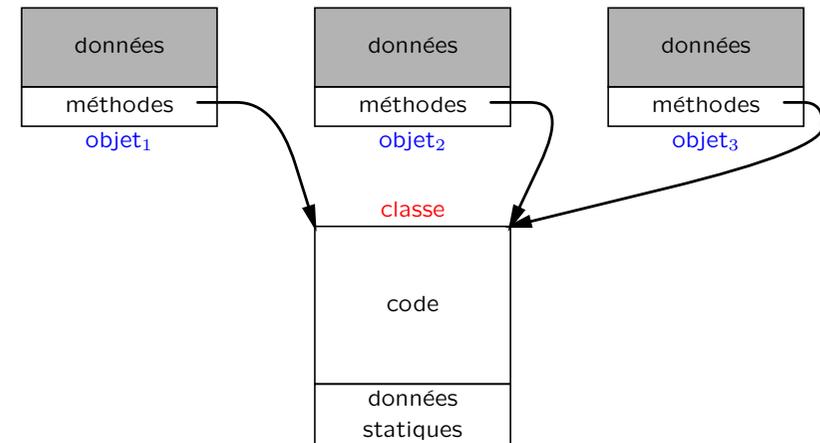
- la **modularité** : les objets favorisent le regroupement des fonctions et des données \Rightarrow structuration.

Mais

- La modularité provient aussi de l'**accessibilité** des champs définis dans chaque classe et de la **compositionnalité** (un module est construit à partir d'autres modules).
- modularité \neq programmation par objets (cf. Modula, ML)

Objets

Les objets sont les **instances** d'une classe. Ils ont un **état** (la valeur de leurs champs de données) et un ensemble de **méthodes** attachées.



Programmation par objets (3/3)

L'argument supplémentaire est celui de la

programmation incrémentale

- l'**héritage** :
 - une classe peut être sous-classe d'une autre classe.
 - Dans la sous-classe, on ajoute ou on redéfinit des champs ou méthodes. On dit qu'on a **spécialisé** ces champs ou méthodes.
 - Quand une méthode est appelée, on utilise la méthode la plus spécialisée, c'est-à-dire appartenant à la plus petite sous-classe contenant l'objet.
- héritage **simple** en Java : une classe n'est sous-classe que d'une **seule** classe \Rightarrow pas d'ambiguïté sur la méthode à appeler. En C++, Smalltalk ou Ocaml, l'héritage est multiple.

Sous-classe et sous-typage (1/6)

```
class Point { // ----- Version 1
    double x, y;
    static void translation (Point p, double dx, double dy) {
        p.x = p.x + dx; p.y = p.y + dy;
    }

    static void rotation (Point p, double theta) {
        double x = p.x * Math.cos(theta) - p.y * Math.sin(theta);
        double y = p.x * Math.sin(theta) + p.y * Math.cos(theta);
        p.x = x; p.y = y;
    }
}

class PointC extends Point {
    final static int JAUNE = 0, ROUGE = 1;
    int c;
    PointC (double x0, double y0, int col) { x = x0; y = y0; c = col;}

    public static void main (String[] args) {
        PointC p = new PointC(3, 4, JAUNE);
        rotation (p, Math.PI/2);
    }
}
```

Sous-classe et sous-typage (2/6)

Version 1

- La classe `PointC` est une sous-classe de `Point`.
- Conversion **implicite** de `PointC` en `Point`.



Version 2

On change la signature de `Rotation` pour rendre un `Point` comme résultat :

- conversion **implicite** pour passer un `PointC` en `Point` comme argument à `Rotation`;
- conversion **explicite** (`cast`) de `Point` en `PointC` pour utiliser le résultat de `Rotation` comme `PointC`.

Sous-classe et sous-typage (3/6)

```
class Point { // ————— Version 2
    double x, y;
    static void translation (Point p, double dx, double dy) {
        p.x = p.x + dx; p.y = p.y + dy;
    }

    static Point rotation (Point p, double theta) {
        double x = p.x * Math.cos(theta) - p.y * Math.sin(theta);
        double y = p.x * Math.sin(theta) + p.y * Math.cos(theta);
        p.x = x; p.y = y;
        return p;
    }
}

class PointC extends Point {
    final static int JAUNE = 0, ROUGE = 1;
    int c;
    PointC (double x0, double y0, int col) { x = x0; y = 0; c = col;}

    public static void main (String[ ] args) {
        PointC p = new PointC(3, 4, JAUNE) ;
        p = (PointC) rotation (p, Math.PI/2); // cast nécessaire
    }
}
```

Sous-classe et sous-typage (4/6)

Version 2

et

Version 3

- En Java, chaque objet possède un certain type lors de sa création.
- L'objet conserve ce type pendant toute sa durée de vie (contrairement à Pascal, ML, C, C++) : **typage dynamique** des objets.
- L'opérateur `instanceof` teste l'appartenance d'un objet à une classe. Ainsi :

```
if (p instanceof PointC)
    system.out.println ("couleur = " + p.c);
```

- Donc l'expression `(PointC) p` équivaut à :

```
if (p instanceof PointC)
    p
else
    throw new ClassCastException();
```

Sous-classe et sous-typage (5/6)

```
class Point { // ————— Version 3
    double x, y;
    static void translation (Point p, double dx, double dy) {
        p.x = p.x + dx; p.y = p.y + dy;
    }

    static Point rotation (Point p, double theta) {
        double x = p.x * Math.cos(theta) - p.y * Math.sin(theta);
        double y = p.x * Math.sin(theta) + p.y * Math.cos(theta);
        return new Point(x, y);
    }
}

class PointC extends Point {
    final static int JAUNE = 0, ROUGE = 1;
    int c;
    PointC (double x0, double y0, int col) { x = x0; y = 0; c = col;}

    public static void main (String[ ] args) {
        PointC p = new PointC(3, 4, JAUNE) ;
        p = (PointC) rotation (p, Math.PI/2); // lève ClassCastException
    }
}
```

Sous-classe et sous-typage (6/6)

L'opérateur + de concaténation appelle toujours la méthode (publique) `toString` de la classe de son argument. On peut **spécialiser** cette méthode :

```
class Point {
    double x, y;
    ...
    public String toString() { return "(" + x + ", " + y + ")"; }
}

class PointC extends Point {
    final static int JAUNE = 0, ROUGE = 1;
    int c;
    ...
    public String toString() { return "(" + x + ", " + y + ", " + c + ")"; }

    public static void main (String[ ] args) {
        PointC p = new PointC(3, 4, JAUNE) ;
        System.out.println (p);
    } }
```

Remarque : `System.out.print(x) = System.out.print(x.toString())`

Contrôle d'accès et sous-classes

Les champs ou méthodes d'une classe peuvent être :

- **public** pour permettre l'accès depuis toutes les classes,
- **private** pour restreindre l'accès aux seules expressions ou fonctions la classe courante,
- **par défaut** pour autoriser l'accès depuis toutes les classes du même paquetage,
- **protected** pour restreindre l'accès aux seules expressions ou fonctions de sous-classes de la classe courante.

Exercice 2 Comment donner un accès public aux objets d'une classe, tout en contrôlant leurs créations, i.e. en interdisant les accès publics aux constructeurs ?

Résumé

- une **classe** contient un **ensemble de champs** (données ou fonctions)
- une **sous-classe** contient au moins tous les champs de sa classe parente, **et** un certain nombre de **nouveaux champs**.
- une **sous-classe** peut **redéfinir** (spécialiser) un certain nombre de champs de sa classe parente (*overriding*).
- les méthodes redéfinies doivent avoir la même signature (arguments et résultat) que dans la sur-classe.
- les méthodes redéfinies doivent fournir au moins les mêmes droits d'accès.
- spécialisation \Rightarrow héritage (cf. plus tard)
- En Java, sous-classe \Rightarrow sous-typage \Rightarrow conversion implicite pour passer à une sur-classe.
- Conversion explicite pour passer dans une sous-classe.

Types disjonctifs et classes (1/4)

Programmation récursive simple des ASA :

Données : t terme, e liste d'association de valeurs aux variables.

But : calcul la valeur du terme t dans l'**environnement** e .

Méthode utilisée : **induction structurelle** sur les ASA.

```
static int evaluer (Terme t, Environnement e) {
    switch (t.nature) {
        case ADD: return evaluer (t.a1, e) + evaluer (t.a2, e);
        case SUB: return evaluer (t.a1, e) - evaluer (t.a2, e);
        case MUL: return evaluer (t.a1, e) * evaluer (t.a2, e);
        case DIV: return evaluer (t.a1, e) / evaluer (t.a2, e);
        case CONST: return t.val;
        case VAR: return assoc (t.nom, e);
        default: throw new Error ("Erreur dans evaluation");
    } }

static int assoc (String s, Environnement e) {
    if (e == null) throw new Error ("Variable non définie");
    if (e.nom.equals(s)) return e.val;
    else return assoc (s, e.suivant);
} }
```

Types disjonctifs et classes (2/4)

Programmation par objets des ASA :

Une classe abstraite peut contenir quelques champs indéfinis.

```
abstract class Terme {
    abstract int eval (Environnement e);
    abstract public String toString ();
}
```

```
class Add extends Terme {
    Terme a1, a2;
    Add (Terme x, Terme y) {a1 = x; a2 = y; }
    int eval (Environnement e) { return a1.eval(e) + a2.eval(e); }
    public String toString () { return "(" + a1 + " + " + a2 + ")"; }
}
```

```
class Mul extends Terme {
    Terme a1, a2;
    Mul (Terme x, Terme y) {a1 = x; a2 = y; }
    int eval (Environnement e) { return a1.eval(e) * a2.eval(e); }
    public String toString () { return "(" + a1 + " * " + a2 + ")"; }
}
```

Types disjonctifs et classes (4/4)

```
class Test {
    public static void main (String[] args) {
        ...
        Terme t = expression();
        System.out.println (t);
        Environnement e = ... ;
        System.out.println (t.eval(e));
    }
}
```

Exercice 3 Ecrire la méthode `equals` qui teste l'égalité (structurelle) de deux termes.

Exercice 4 Calculer la dérivée d'un terme arithmétique en programmation par objets.

Exercice 5 Ecrire la méthode de belle-impression de termes arithmétiques en programmation par objets.

Types disjonctifs et classes (3/4)

```
class Const extends Terme {
    int valeur;
    Const (int n) { valeur = n; }
    int eval (Environnement e) { return valeur; }
    public String toString () { return valeur + ""; }
}
```

```
class Var extends Terme {
    String nom;
    Var (String s) { nom = s; }
    int eval (Environnement e) { return Environnement.assoc(nom, e); }
    public String toString () { return nom; }
}
```

```
class Sub extends Terme {
    Terme a1, a2;
    Sub (Terme x, Terme y) {a1 = x; a2 = y; }
    int eval (Environnement e) { return a1.eval(e) - a2.eval(e); }
    public String toString () { return "(" + a1 + " - " + a2 + ")"; }
}
```

Procédural ou Objets ?

- choix du **contrôle** : par les fonctions ou par les données ?
- le style de programmation diffère entre petits programmes et **gros** programmes (> 10⁴ lignes).
- dépend de la stratégie de **modification**.
- en dernière analyse, c'est affaire de **goût**.
- objets **≠** modularité.
- programmation par objets utilisée dans les boîtes à outils **graphiques** (*look* commun à toutes les fenêtres) et **réseau** (données et méthodes se déplacent simultanément).
- programmation incrémentale très **populaire** dans l'**industrie**.