

INF 431



B. WERNER



Objets et Héritage

3 février 2010

Où en est-on ?

Amphi 1 : Introduction, interfaces, génériques.

Amphi 2 : héritage.

Amphi 3 : Complexité.

Amphi 4 : Arbres.

Amphi 5 : graphes (I).

Amphi 6 : graphes (II).

Amphi 7 : graphes (III).

Amphi 8 : graphes (IV).

Amphi 9 : graphes (V).

Compilation / Interprétation

Comment est exécuté un programme ?

- Interprété : Un **interpréteur** parcourt le programme et l'exécute pas-à-pas.
- Compilé : un **compilateur** traduit le *programme source* en exécutable.

Quel exécutable ?

- Code machine ou "code natif" : C, C++, Ocaml. **plus rapide**
- Machine virtuelle, facilement interprétée : Java (JVM), C#, F#, J# (CLR), Ocaml **plus portable**

Applets : code compilé mobile sur le réseau \Rightarrow machine virtuelle
(**bytecode**)

Classes et objets

```
public class Point {  
    int x, y;  
    Pt(int a, b){  
        x=a; y=b;  
    }  
    ...  
    Point p1 = new Point(1,1);  
    Point p2 = new Point(2,2);  
    ...
```

p1 et p2 sont des **instances** de la classe Point
p1 et p2 sont des **objets**

Statique / dynamique : les Variables

statique :

- Emplacement mémoire fixé à la compilation (=statiquement)
- Commun à toutes les instances de la classe

```
class Sta {  
    public static int a = 1;  
    ....
```

`Sta.a` correspond à un emplacement mémoire inchangé dans tout le programme.

```
class Sta {
    public static int a = 1;
    public Sta () { a = a+1; }

    public static void main (String[] args)
    {
        Sta s1 = new Sta();
        Sta s2 = new Sta();
        System.out.println(a);
    }
}
```

Donne 3

Idem pour `sta.a` ou `s1.a` ou `s2.a`

Champs statiques et dynamiques

```
class Mixte{
    public static int a = 1;
    public int b = 2;
    public int c;

    public Mixte(int i) {c=i};
    public Mixte() {};

public static void main (String[] args)
    {
        Mixte m1 = new Mixte(3);
        Mixte m2 = new Mixte(5);
        System.out.println(m1.b);           // Donne 2
        System.out.println(m2.c);           // Donne 5
    }
```

Les emplacements mémoires des champs dynamiques sont choisis à l'exécution

```
class Exemple {
    static int a;
    int b;
    static void mS () { ..... };
    void mD () { ... };
    public Exemple(...) {...};
}
...
Exemple e1 = new Exemple(...);
Exemple e2 = new Exemple(...);
...
```

Héritage et spécialisation

```
public class Rectangle {  
    public int topX, topY, height, width;  
    public void draw() { ... }  
    ....  
}
```

```
public class Window extends Rectangle {  
    static final int red = 0, blue=1;  
    int color = red;  
    public draw() { ... color ... }  
    public Window(...) { ... }  
}
```

Les variables comme `topX` sont **héritées** de la classe père.

La méthode `draw` est **spécialisée** ou **redéfinie** (*overwritten*).

L'héritage est un sous-typage

Tout ce qu'on fait avec un `Rectangle` on peut aussi le faire avec un `Window`.

```
int size(Rectangle a) {return(a.height * a.width); }  
...  
Window r;  
...  
... size(r) ... // OK !
```

Propriétés de l'héritage

- Une classe peut être sous-classe d'une autre classe **et d'une seule** (**héritage simple**) ; une classe peut avoir plusieurs sous-classes ;
- Une classe hérite de toutes les variables et méthodes de la classe père ;
- 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 choisit **dynamiquement** (liaison retardée) la méthode la plus spécialisée, c'est-à-dire appartenant à la plus petite sous-classe connue contenant l'objet ; il n'y a pas d'ambiguïté grâce à l'héritage simple.

Rem. En C++, Smalltalk ou Ocaml, l'héritage est multiple.

Méthodes statiques et dynamiques

Qu'est-ce qui est dynamique ?

C'est la **liaison** entre le code appelant la méthode et le code exécuté.

Ex. Quelle est la valeur affichée par le programme suivant ?

```
class C{
    void f() { g(); }
    void g() { System.out.println(1); }
class D extends C{
    void g() { System.out.println(2); }
    public static void main(String[] args){
        D x = new D();
        x.f();
    } }
```

Héritage vs. surcharge

- La liaison due à l'héritage est déterminé dynamiquement
- la surcharge est déterminée à la **compilation** ;

```
class C {  
    void aff (int i) {  
        System.out.println("C'est un entier"); }  
    void aff (String s) {  
        System.out.println(s); }  
}
```

...

```
aff(5);
```

C'est un entier

```
aff("Bonjour !");
```

Bonjour!

`aff(int i)` et `aff(String s)` pourraient porter des noms différents sans changer le comportement du programme.

Super!

Spécialisons encore :

```
public class BorderWindow extends Window {
    public int thickness = 10;
    ...

    public draw() {
        super.draw();
        ... // le code pour ajouter le cadre
    }
}
```

Pour `draw` on veut utiliser *en partie* la méthode définie pour `Window`

- Héritage **simple** \Rightarrow chaque classe a une seule classe parente, accessible par le mot clef **super** ; on ne peut l'utiliser que dans des méthodes d'objets ;
- **super.f** est la méthode **f** de la classe parente ;
- **super()** (avec d'éventuels arguments) est le constructeur (appelé par défaut) de la classe parente ;

Une parenthèse : les exceptions en Java

But : rattraper (certaines) erreurs de façon à ne pas faire planter le programme.

```
public static void main(String[] args){
    try{
        int x = Integer.parseInt(args[0]);
        System.out.println(x);
    } catch (NumberFormatException e) {
        System.err.println("Mauvais argument : "
            + e.getMessage());
    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("Mauvais nombre d'arguments");
    } finally {
        System.out.println("On s'en est sorti");
    }
}
```

On isole le cas anormal. Le cas normal s'écrit indépendamment du cas anormal.

On utilise **finally** pour effectuer une opération finale s'il y a exception ou pas (cela évite la duplication de code).

Parenthèse : le fonctionnement des exceptions

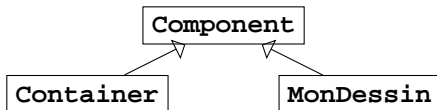
Une exception se propage dans le bloc de la méthode où elle est levée. Si elle n'est pas récupérée dans le bloc, elle se propage à la méthode d'appel et ainsi de suite.

Si aucune méthode ne la récupère, elle arrive dans `main`, et c'est l'interpréteur Java qui la récupère, affiche la pile d'appels avant de sortir.

On s'en est sorti

```
Exception in thread "main" java.lang.NumberFormatException: 1
    at Excep.parseInt(Excep.java:7)
    at Excep.main(Excep.java:15)
```

Un exemple de liaison retardée (à la awt)



```
public class Component{
    ...
    public void paint(){
        // ne fait rien c'est normal
    }
    ...
}
public class Container extends Component{
    ...
    public void add(Component comp){
        components[nbcomponents++] = comp;
    }
    public void paint(){
        for(int i = 0; i < nbcomponents ; i++)
            components[i].paint();
    }
}}
```

```
public class MonDessin extends Component{
    ...
    public void paint(){
        // mon joli dessin
    }
}

public class Essai{
    public static void main(String[] args){
        MonDessin dessin = new MonDessin(...);
        Container fenetre = new Container("titre");
        fenetre.add(dessin);
        fenetre.paint();
    }
}
```

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 de la classe courante (on a accès aux champs privés d'une autre instance de sa classe),
- **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.

Contrôle d'accès (suite)

- un champ **final** ne peut être **redéfini** (on peut donc optimiser sa compilation) ; un champ **final** ne peut être affecté qu'une seule fois (soit dans chaque constructeur, soit par **final float x = 1.23;**).
- une classe **final** ne peut être spécialisée (ex. **String**).
- une méthode peut être déclarée **final** (pas besoin de déclarer toute la classe **final**).

Typage

Quelques notions de typage :

- Un type est un ensemble de valeurs partageant une même propriété.
- Types primitifs : `int`, `char`, etc.
- En Java, une classe est un type.
- Classes prédéfinies : `string`, etc.

Vérification du typage : statique (à la compilation) ou dynamique (à l'exécution).

Intérêt du typage statique : détection d'erreurs, optimisation de code, sécurité (pas d'atteinte à la mémoire).

Intérêt du typage dynamique : plus précis
(`if(a) return 1; else return false;`).

La théorie des types est introduite dans le cours Théorie des Langages de Programmation d'année 3.

Le typage en Java

- En Java, chaque objet possède un certain type lors de sa création, **qu'il conserve pendant toute sa durée de vie** (contrairement à Pascal, ML, C, C++).
- **instanceof** teste l'appartenance d'un objet à une classe :

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

- (PointC)p équivaut à :

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

Sous-classe et sous-typage

Conversion implicite :

$t <: t'$ signifie $x : t \Rightarrow x : t'$ pour tout x

On dit que t est un **sous-type** de t' .

En Java :

`byte <: short <: int <: long`

`float <: double`

`char <: int`

Une classe est un type ; une sous-classe un sous-type. On propage la notation :

$C <: C'$ si C est une sous-classe de C'

Ex. PointC <: Point.

Racine de la hiérarchie des classes

Object est la classe la plus générale.
(ancêtre de toutes les classes)

$$\forall C \quad C <: \text{Object}$$

(*C* est une classe ou un tableau quelconque).

- La hiérarchie des classes est une simple arborescence.
- Les méthodes de **Object** sont **clone**, **finalize**, **getClass**, **hashCode**, **notify**, **notifyAll**, **wait**, **equals**, **toString**.
- Seules **clone**, **hashCode**, **equals**, **toString**, **finalize** peuvent être redéfinies.
- **Tous** les objets contiennent ces méthodes.
- On convertit les **scalaires** **int**, **float**, **char**, ... en objets avec un « conteneur » :

```
Integer x = 3;
```

Attention

Utilisez et redéfinissez les méthodes comme `equals`

Évitez le test `s1 == s2` pour les `String`.

Pourquoi ?

mais plutôt : `s1.equals(s2)`

Similaire pour `hashCode`

Retour aux points

```
class Point{
    double x, y;
    static void translation(Point p,
                            double dx, double dy){
        p.x += dx; p.y += dy;
    }
}
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 pc = new PointC(3, 4, JAUNE);
        translation(pc, 1, 2);
    }
}
```

Ce qui ne peut marcher

```
class Point{
    ...
    static Point translation(Point p,
                             double dx, double dy){
        Point pt = new Point();
        pt.x = p.x + dx; pt.y = p.y + dy;
        return pt;
    }
}
class PointC extends Point{
    ...
    public static void main(String[] args){
        PointC pc = new PointC(3, 4, JAUNE);
        pc = (PointC)translation(pc, 1, 2);
    }
}
```

Pas de conversion père → fils. La compilation marche (avec le cast), mais pas l'exécution (ClassCastException).

Résumé du cours

- Attention à la programmation objet, elle recèle de nombreux pièges.
- Vous pouvez programmer à l'ancienne, ou bien vous lancer dans la modernité échevelée. Nous resterons en général conservateur dans le domaine.

Prochains rendez-vous : TD cet après-midi (nommez les délégués qui manquent) ; amphi 03 mercredi prochain 10 février.