Introduction à la programmation orientée objets en Java

Philippe REITZ
reitz@lirmm.fr

CNAM – novembre 2008

Plan

- Contexte
- Idées générales
  - Objet, classe, instance, message, …
  - Encapsulation : les classes sans l'héritage
  - Héritage simple entre classes
  - Extensions
    - Héritage multiple
    - Multi-héritage

Paradigmes de programmation

Paradigme = ensemble de méthodes, théories ou techniques pour résoudre des problèmes

- Assembleur (machines à registre) [1950]
- Programmation impérative [1960]
  - Fortran, Cobol, PL/1, Basic, Pascal, C, …
- Programmation fonctionnelle [1960]
  - Lisp, APL, Scheme, ML, CaML, Haskell, …
- Programmation logique (par contraintes) [1970]
  - Prolog, Datalog, …

Paradigmes de programmation

- Programmation par règles [1970]
  - Gamma, MGS, OPS5, systèmes experts, …
- Programmation concurrente [1970]
  - Ada, Occam, Erlang, …
- Programmation objet [1970]

  sujet de cette présentation…
- Programmation agent [1980]
  - Actor, Agent-0, …

Programmation objet : le slogan

- Programme en cours d'exécution =

  ensemble d'**objets** (le *monde*) qui interagissent via des échanges de **messages**
  - un objet a un état évoluant dans le temps (processus)
  - le processus modifiant l'état d'un objet est déclenché suite à la réception d'un message par cet objet (passivité)
  - un objet ne peut modifier l'état d'un autre objet qu'en lui envoyant des messages (localité)
  - des objets apparaissent ou disparaissent du monde (dynamicité)
- Lancer un programme =

  envoyer un message à un objet particulier

Historique du paradigme objet

- Les types abstraits (apogée : années 70)
- Les premiers langages OO
  - Simula (années 1960)
  - Smalltalk (années 1970)
- Les langages OO industriels (années 1980)
  - C++ / Ada
  - Smalltalk, Objective-C, CLOS, Eiffel
  - Delphi / Java / C#

Les types abstraits de données

- Historique
  - apogée des travaux dans les années 1970
- Objectifs affichés
  - [*interface*] ne précise que :
    - les noms des types de données
    - les signatures des opérations qui manipulent ces données
      - signature d'une opération = son nom + ses paramètres
    - les propriétés algébriques de ces opérations
  - [*implémentation*] précise :
    - la réalisation effective d'une donnée
    - la réalisation des opérations qui la manipulent

Les types abstraits de données

Ex : décrit le type des listes composées d'éléments de type *a* (*a* quelconque)

- type :

  `liste<a>`

- opérations :

```
     vide : liste<a>
    ajout : a * liste<a> → liste<a>
     tête :     liste<a> → a
    reste :     liste<a> → liste<a>
est_vide? :     liste<a> → bool
```

- axiomes :

soient `e : a`
et `L : liste<a>`

(1) `tête( ajout(e, L) ) = e`
(2) `reste( ajout(e, L) ) = L`
(3) `est_vide?( vide ) = true`
(4) `est_vide?( ajout(e, L) ) = false`

- erreurs :

(1) `tête( vide )`
(2) `reste( vide )`

Les types abstraits de données

- Les principaux apports aux LOO
  - séparation interface/implémentation
    - concept d'**encapsulation**
  - notion de *module* ou *paquetage*
    - concept de **classe**
  - spécifications algébriques
    - programmation par **contrat** (Eiffel)
      - **invariant** de classe
      - **pré** et **post condition**

Notions premières

- Objet ⇒ se caractérise par :
  - identifiant
  - état local
  - comportement
- Message ⇒ se caractérise par :
  - identifiant objet émetteur
  - identifiant objet receveur
  - information portée
- Envoi de message
  - seul moyen d'obliger un objet à se comporter
  - comportement = comment réagir à la réception d'un message

Notion d'objet

- Un objet :
  - se distingue des autres par son **identifiant**
  - se caractérise par son **état** local
    - état = ensemble de **propriétés** (ex : du genre **struct** en C)

      propriété =
      - » un nom
      - » une valeur *(un identifiant est une valeur possible)*
      - » un type (si langage typé)
  - possède un **comportement** :
    - comment réagir lors de la réception d'un message :
      - modification possible de l'état
      - émission possible de messages
      - création / suppression possible d'objets
  - a une **durée de vie** :
    - naissance : un message permet de créer un nouvel objet
    - mort : un message permet de retirer l'objet receveur du monde

Notion d'objet : un exemple

Objet **#8763** ← identifiant

– propriétés :

- *nom* = « Dupont »
- *prénom* = « Jean »
- *âge* = 25
- *père* = **#56977**

} ← état

– comportement :

- **si**

  message reçu = *dit_bonjour()*

  alors

  envoyer le message *afficher(« Bonjour »)* à l'objet identifié *Système*

- **si** … *(suite de la description du comportement)*

Familles de langages OO

- Motivation
  - Constat : une propriété d'objet possède une valeur ; cette valeur est soit un identifiant d'objet, soit une valeur de base : entier, flottant, caractères, …
  - Question : une valeur de base (ex : un entier) est-il l'identifiant d'un objet ?
- Réponses
  - Oui, tout est objet
    - langages dits *purs* : Smalltalk, Eiffel (presque)
  - Non, tout n'est pas objet : distinction *valeur*/*objet*
    - langages dits *impurs* : la grande majorité des langages

      en particulier : C++, Ada, Delphi, Java, C#, …

Familles de langages OO

- Motivation
  - Constat : plusieurs objets peuvent être distincts (identifiants, états) mais se caractériser par un même comportement
  - Question : comment *factoriser* cette information commune ?
- Réponses
  - Les langages à classes
    - approche majoritairement suivie par les auteurs de langages
    - une classe est assimilée à un type de données
  - Les langages à prototypes
    - approche peu répandue (exemple : *JavaScript*)
      - la dichotomie classe/instance n'existe pas
      - seule celle de prototype existe
    - support de la formalisation des langages objets (sémantique)

Les langages à classes

- un objet = une instance
  - chaque instance
    - porte un identifiant
    - est associée à une classe unique (relation d'instanciation)
    - possède un état
      - valeurs associées aux attributs définis par sa classe
  - chaque classe
    - décrit la structure de l'état de ses instances
      - un vecteur d'attributs (attribut = nom + type)
    - définit le catalogue des messages traitables par ses instances
      - signatures de méthodes
    - décrit le comportement à tenir par ses instances pour chaque message reçu
      - catalogue du code de méthodes

Les langages à classes : terminologie

- classe **instanciable**
  - classe pour laquelle il est possible de définir des instances
- classe **instanciée**
  - classe pour laquelle au moins une instance est définie
- classe **abstraite**
  - classe non instanciable : aucune instance ne peut lui être associée
- classe **concrète**
  - classe instanciable, à l'inverse d'une classe abstraite

Les classes : plan de présentation

- Les classes sans l'héritage
  - tous les concepts clés
  - l'encapsulation
- Les classes et l'héritage simple
- Les classes et l'héritage multiple
- Autres extensions
  - Réflexivité (ou introspection)
  - Multi-héritage et multi-méthodes
  - Concurrence et agents

Les classes sans héritage

- Propriétés
  - Attributs
  - Méthodes
- Envois de messages
  - Accès aux attributs (lecture ou écriture)
  - Appels de méthodes
- Protection des propriétés
- Méthodes
  - Définition de leur code
    - variable spéciale d'auto-référencement
  - Constructeurs
  - Destructeur

Propriété = attribut ou méthode

- propriété = nom + valeur + type
  - méthode ⇒ la valeur est *fonctionnelle* (code exécutable)
  - attribut ⇒ la valeur n'est pas fonctionnelle (ex : entier)
- une classe
  - décrit les attributs portés par ses instances (nom + type)
  - définit les méthodes applicables à ses instances (nom + type + valeur [*code*])
- une instance
  - définit les valeurs de ses attributs

Propriété = attribut ou méthode

- classe Point
  - attributs
    - `x`, `y` : réels
  - méthodes : leurs signatures
    - `lireX()`, `lireY()` : réel
    - `forcerX(`réel`)`, `forcerY(`réel`)` : void
    - `déplacer(`réel, réel`)` : void
  - méthodes : leurs codes
    - `lireX() { … }`
    - `forcerX(v) { … }`
    - …
- instance de Point **#8756**
    - `x` = 3
    - `y` = -2

Envois de messages

- accès aux attributs
  - lecture : $o.a$
    - valeur associée à l'attribut de nom *a* pour l'objet *o*
  - écriture : $o.a = v$
    - nouvelle valeur *v* associée à l'attribut de nom *a* pour l'objet *o*
- appels de méthodes
  - appel : $o.m(e_1, \ldots, e_n)$
    - exécute le code de la méthode *m* dans le contexte de l'objet *o* et des arguments e_i de l'appel

Envois de messages

- classe Point
  - attributs
    - x, y : réels
  - méthodes : les signatures
    - lireX(), lireY() : réel
    - forcerX(réel), forcerY(réel) : void
    - déplacer(réel, réel) : void
  - méthodes : les codes
    - lireX() { return x; }
    - forcerX(v) { x = v; }
    - déplacer(dx, dy) {
      x += dx; y += dy;
      }
    - …
- instance de Point **#8756**
  - x = 3
  - y = -2
- messages
  - lecture

    #8756.x
  - écriture

    #8756.x = 4
  - appel de méthode

    #8756.lireX()

    #8756.déplacer(-1, 2)

Protection des propriétés

- objectif :
  - autoriser ou non l'envoi d'un message selon le contexte dans lequel il est défini
- exemples (C++, Java, C#, Php5) :
  - protection **private**
    - une propriété *P* d'une classe *C* n'est manipulable que dans le code d'une méthode de cette même classe *C*
  - protection **public**
    - une propriété *P* d'une classe *C* est manipulable quelque soit le contexte de la manipulation
  - protection **protected**
    - une propriété *P* d'une classe *C* n'est manipulable que dans le code d'une méthode rattachée à une classe héritant de *C*

Encapsulation

les attributs d'une instance :

- ne doivent pas pouvoir être manipulés directement
  - propriétés non publiques
    - implémentation cachée de l'extérieur
    - objet = boite noire
- ne peuvent être manipulés que par des méthodes dédiées
  - propriétés publiques
    - *sélecteur* (ou *accesseur*, ou *observateur*) = méthode de lecture de la valeur d'un attribut
    - *modificateur* = méthode d'écriture d'une valeur dans un attribut

Les méthodes

- méthodes particulières
  - Constructeur
    = méthode appelée automatiquement à la création d'un objet
  - Destructeur
    = méthode appelée automatiquement à la suppression d'un objet
- codage d'une méthode
  - il y a toujours un paramètre implicite dans une méthode : celui portant l'identifiant de l'objet receveur de l'appel
    - variable d'auto-référencement
      - **`this`** en C++, Java ou Php
      - **`self`** en Smalltalk

Exemple en Java

Définition d'une classe :

```
class uneDate {
  // attributs --------------------
  int jour,   // entier de 1..31
      mois,   // entier de 1..12
      année;  // entier
 // constructeur -----------------
  uneDate(int j, int m, int a) { … }
 // méthodes ---------------------
  void affiche(String pré, String post) { … }
  void affiche() { … }
  void lendemain() { … }
  int dernierJourMois() { … }
  boolean annéeBissextile() { … }
}
```

Exemple en Java

- Les trois envois de messages de base :
  - Lecture de la valeur d'un attribut d'un objet

```
System.out.println("jour = " + d.jour);
```

  - Ecriture de la valeur d'un attribut d'un objet

```
d.mois = 4;
```

  - Appel d'une méthode
    - création d'une instance : opérateur **new**

```
d = new uneDate(28, 2, 2004);
```

    - appel d'une méthode

```
d.affiche()
```

Exemple en Java

- L'exécution de :

```
for (uneDate d : new uneDate[] {
      new uneDate(28, 2, 2004),  new uneDate(28, 2, 2001),
      new uneDate(28, 2, 1900),  new uneDate(31, 12, 2003),
      new uneDate(10, 12, 2004)
    }) {
    d.affiche("", " => ");
    d.lendemain();
    d.affiche();
  };
```

- provoque l'affichage de :

```
28/02/2004 => 29/02/2004
28/02/2001 => 01/03/2001
28/02/1900 => 01/03/1900
31/12/2003 => 01/01/2004
10/12/2004 => 11/12/2004
```

Exemple en Java

- Définition du code des méthodes
  - constructeur

```
uneDate(int j, int m, int a) {
  this.jour  = j;
  this.mois  = m;
  this.année = a;
}
```

*variable d'auto-référencement : **this** désigne toujours l'objet receveur du message dans le code d'une méthode*

  - méthode d'affichage

```
void affiche(String pré, String post) {
  System.out.print(pré+
          this.jour+"/"+
          this.mois +"/"+
          this.année+post);
}
```

Exemple en Java

- Définition du code des méthodes (suite)
  - modificateur

```
void lendemain() {
  // augmente la date d'un jour
  if (this.jour == this.dernierJourMois()) {
    this.jour = 1;
    if (this.mois == 12) {
      this.mois = 1;
      this.année += 1;
    } else
      this.mois += 1;
  } else
    this.jour += 1;
}
```

Exemple en Java

- Définition du code des méthodes (fin)
  - méthode utilitaire

```
int dernierJourMois() {
 // retourne le dernier jour du mois de la date
  switch (this.mois) {
    case 2:
      return this.annéeBissextile() ? 29 : 28;
    case 4: case 6: case 9: case 11:
      return 30;
    default:
      return 31;
} }
```

  - méthode utilitaire

```
boolean annéeBissextile() {
 // retourne vrai ou faux selon que l'année de la date est bissextile ou non
  int a = this.année;
  return a%400==0 || a%100!=0 && a%4==0;
}
```

Même exemple en C++

```
class uneDate {
  public: // types locaux
    typedef unsigned short int unJour;
    enum unMoisS {Jan=1, Fev, Mar, …, Sep, Oct, Nov, Dec};
    typedef unsigned short int unMoisN;
    typedef unsigned int       uneAnnee;
  private: // attributs
    unJour   le_jour;
    unMoisN  le_mois;
    uneAnnee l_annee;
  public: // constructeurs
    uneDate(unJour, unMoisN, uneAnnee);
    uneDate(unJour, unMoisS, uneAnnee);
    uneDate(char*); // format jj/mm/aa ou jj/mm/aaaa
    uneDate(unJour, char*, uneAnnee);
  public: // méthodes publiques
    void lendemain();
    ostream& affiche(ostream& = cout) const;
    bool operator<(const uneDate&) const;
  private: // méthodes à usage interne
    unJour dernier_jour_mois() const;
    bool bissextile() const;
}; // le code des méthodes est défini ailleurs, non présenté ici…
```

L’héritage

- objectif :
  - outil permettant de définir de nouvelles classes en modifiant ou étendant la définition de classes existantes
- principe :
  - une classe *C* définit des propriétés *P*
  - une classe *S* est définie comme héritant de *C*
    - elle reprend par défaut toutes les propriétés *P* de *C* (héritage de propriétés)
      - elle peut altérer la définition de quelques-unes de ces propriétés héritées (redéfinition de propriétés)
      - plus rare : elle peut préciser des propriétés à ne pas hériter (Eiffel)
    - elle peut ajouter de nouvelles propriétés *Q*

L'héritage – exemple

- classe *Point du plan*
  - Attributs :
    - *x* et *y* : réels
  - Méthodes :
    - *dessiner*(zoneGraphique) { … }
- classe *Point coloré*
  - Hérite de :
    - Point du plan
  - Attributs :
    - *couleur* : { rouge, vert, bleu, blanc, … }
  - Méthodes :
    - *dessiner*(zoneGraphique) { … }
- classe *Point coloré à coordonnées entières*
  - Hérite de :
    - Point coloré
  - Attributs :
    - *x* et *y* : entiers

ajout d'un nouvel attribut

redéfinition du code d'une méthode

redéfinition de types d'attributs

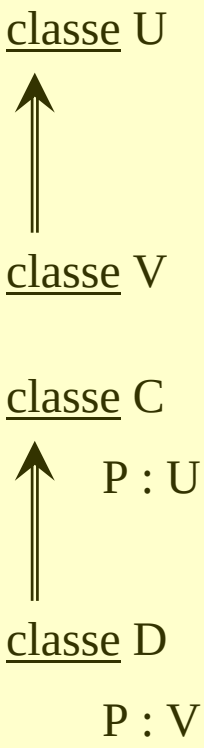
L’héritage : terminologie

- si S est une classe qui hérite d’une classe C
  - C est une **super-classe** (ou sur-classe) de S
  - S est une **sous-classe** de C
- pour toute classe C
  - **descendance** = ensemble des classes qui héritent de C (directement ou indirectement), y compris C
  - **ascendance** = ensemble des classes dont C hérite (directement ou indirectement), y compris C
- **graphe d’héritage**
  - graphe orienté sans circuit :
    - les sommets sont les classes
    - si S hérite de C, alors il existe un arc de S vers C
  - forme canonique : graphe de Hasse
  - héritage simple ou multiple
    - héritage simple : toute classe n’a au mieux qu’une super-classe
      - le graphe se réduit à un arbre (ou une forêt d’arbres)
    - héritage multiple : au moins une classe a au moins deux super-classes

L'héritage

rappel : propriété = nom + valeur + type

- altérer une propriété héritée, c'est :
  - redéfinir son nom (Eiffel)
  - redéfinir sa valeur
    - exemple : changer le code d'une méthode
  - redéfinir le type de sa valeur (*si langage typé*)
    - en général, ce type est restreint :
      - soit *V* une sous-classe d'une classe *U*
      - soit *P* une propriété d'une classe *C* dont le type de la valeur est *U*
      - une sous-classe *D* de *C* peut redéfinir le type de *P* en le remplaçant par le type *V*
  - redéfinir sa protection (*si langage supportant la notion*)



Un exemple en Java

```
class uneFigure {
  double x, y; // abscisse et ordonnée du point de référence
  uneFigure(double x, double y) {
    this.x = x; this.y = y; }
  uneFigure() { this(0, 0); }
  double taille() { return -1; } // taille indéfinie
  void decrire() { …  }
}

class uneFigurePlane extends uneFigure {
● uneFigurePlane(double x, double y) {
    super(x, y); }
● double taille() { return this.surface(); }
● double surface() { return -2; } // surface indéfinie
}

class unCarré extends uneFigurePlane {
● double c; // coté
● unCarré(double x, double y, double c) {
    super(x, y);
    this.c = c; }
● double surface() { return this.c * this.c; }
}
```

● = *nouvelle propriété ajoutée*

● = *propriété héritée redéfinie*

Un exemple en Java

- l'exécution de :

```
uneFigure c;
c = new uneFigure(3, 4);
System.out.println(c.taille()) ;

c = new uneFigurePlane(3, 4);
System.out.println(c.taille()) ;

c = new unCarre(3, 4, 2);
System.out.println(c.taille()) ;
```

- provoque l'affichage de :

```
-1 (taille indéfinie)
-2 (surface indéfinie)
4
```

L’héritage

L’**héritage** est un outil pour construire de nouvelles classes.

Deux notions sont associées à l’héritage, sans lesquelles il n’aurait guère d’intérêt :

- le **polymorphisme** des variables
  - défini pour les langages typés
- la **liaison dynamique** du code

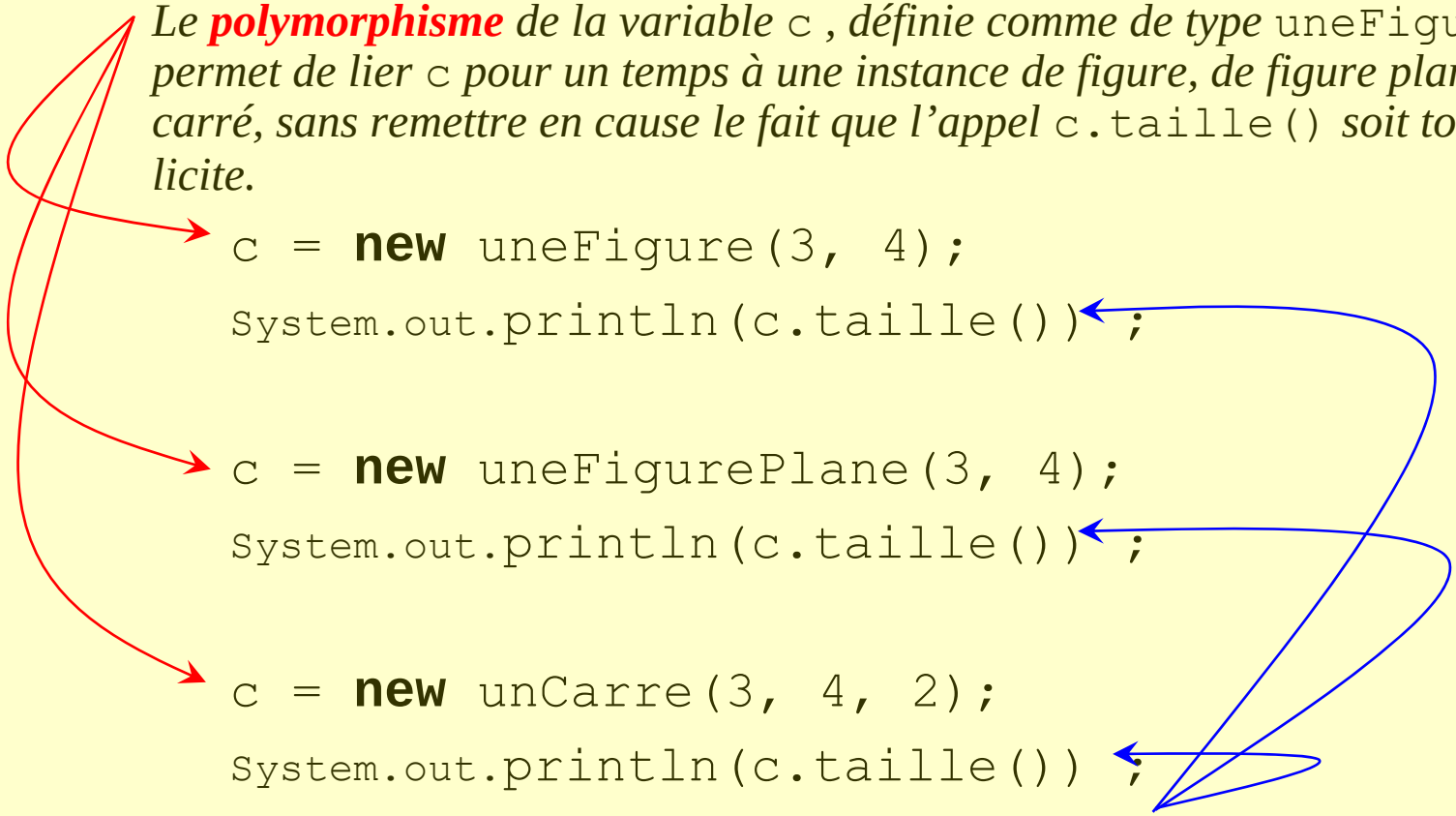
Polymorphisme et liaison dynamique sur un exemple

*Le **polymorphisme** de la variable* `c` *, définie comme de type* `uneFigure`*, permet de lier* `c` *pour un temps à une instance de figure, de figure plane ou de carré, sans remettre en cause le fait que l'appel* `c.taille()` *soit toujours licite.*

```
c = new uneFigure(3, 4);
System.out.println(c.taille());

c = new uneFigurePlane(3, 4);
System.out.println(c.taille());

c = new unCarre(3, 4, 2);
System.out.println(c.taille());
```



*La **liaison dynamique** garantit que le code exécuté à l'appel* `c.taille()` *est toujours celui le mieux adapté à l'objet lié à* `c`*.*

Le polymorphisme

- Définition :
  - hypothèse :
    - soit une classe *C*
  - le **polymorphisme** d'une variable :
    - si *V* est une variable de type *C*
    - alors *V* peut contenir l'identifiant d'une instance dont la classe *S* appartient à la descendance de *C*
- Intérêt :
  - garantit que si *V* exploite une propriété *P* de *C*, cette propriété existera pour n'importe quelle instance d'une classe quelconque de sa descendance et liée à *V*.
- Conséquence :
  - Le lien d'héritage est presque toujours exploité pour représenter la relation abstraite de *généralisation / spécialisation*

Héritage – un bon exemple

```
static void afficheTaille(uneFigure fig) {
   System.out.println("taille de la figure = "+fig.taille());
}
```

polymorphisme de `fig` *: cette variable peut être liée à une instance de figure, figure plane ou carré ; il y aura toujours une taille à afficher.*

```
class uneFigure {
  double x, y; // abscisse et ordonnée du point de référence
  uneFigure(double x, double y) {
    this.x = x; this.y = y; }
  double taille() { return -1; }
  void décrire() { …  }
}

class uneFigurePlane extends uneFigure {
  uneFigurePlane(double x, double y) {
    super(x, y); }
  double taille() { return this.surface(); }
  double surface() { return -2; }
}

class unCarré extends uneFigurePlane {
  double c; // coté
  unCarré(double x, double y, double c) {
    super(x, y);
    this.c = c; }
  double surface() { return this.c * this.c; }
}
```

Une figure plane est un cas particulier de figure : toute figure plane doit au moins se comporter comme n'importe quelle figure.

*La notion de figure généralise celle de figure plane (**généralisation**).*

*Celle de figure plane spécialise celle de figure (**spécialisation**).*

un carré est un cas particulier de figure plane

Héritage – un mauvais exemple

```
class unPoint {
  double x, y; // abscisse et ordonnée du point
  unPoint(double x, double y) {
    this.x = x; this.y = y; }
  void déplacer(double dx, double dy) {
    this.x += dx; this.y += dy; }
}

class unRectangle extends unPoint {
  double H, L; // hauteur, largeur
  unRectangle(double x, double y, double L, double H) {
    super(x, y); this.H = H; this.L = L; }
}

class unCercle {
  unPoint c; // centre
  double R; // rayon
  unCercle(double x, double y, double R) {
    this.c = new unPoint(x, y);
    this.R = R; }
}
```

Héritage exploité pour récupérer les propriétés des points : le rectangle est défini entre-autre par son coin inférieur gauche.

Le bénéfice semble grand : il n'y a rien à écrire de plus pour le coin du rectangle…

Pourquoi est-ce mauvais ?

Le centre d'un cercle est un point.

Le polymorphisme des variables implique que le centre peut être une instance d'une classe héritant de point.

Le centre d'un cercle peut donc être un rectangle ! Absurde géométriquement…

La liaison dynamique

- Définition :
  - Hypothèses :
    - soit *C* une classe définissant une propriété *P*
    - soit *S* une sous-classe de *C* redéfinissant la propriété *P*
    - soit *V* une variable de type *C*, liée à une instance de type *S*
  - La **liaison dynamique** :
    - si *V* exploite la propriété *P*, la liaison dynamique garantit que c'est sa version redéfinie dans *S* qui sera considérée
  - Notion dérivée : le **super-envoi** (appel à la super-méthode)
    - défini lorsque l'instance de type *S* souhaite exploiter sa propriété *P* avec la définition portée par *C*

exemple Java

```
class A {
  void meth() {
   System.out.println("A.meth");
  }
};

class B extends A {
  // redéfinition du code de meth
  void meth() {
   System.out.println("B.meth");
  }
};
```

```
static void fonc(A obj) {
  obj.meth(); // liaison dynamique
};

main() {
    A V = new A();
    fonc(V);
    // affiche A.meth
    V = new B(); // polymorphisme
    fonc(V);
    // affiche B.meth
}
```

Exemple en Java

```
class A {
  void meth() {
   System.out.println("A.meth");
  }
};

class B extends A {
  void autre() {
   this.meth(); }
};

class C extends B {
  void meth() {
   super.meth();
   System.out.println("C.meth");
  }
};
```

```
A v;
v = new A();
v.meth();        // affiche A.meth



v = new B();
v.meth();        // affiche A.meth
v.autre();       // affiche A.meth


v = new C();
v.autre();       // affiche A.meth
                 // puis    C.meth
```

Appel à la super-méthode : appelle le code de `meth()` *défini pour la classe dont* `C` *hérite, soit donc celui de* `B.meth()`*, lequel est celui de* `A.meth()`*, puisque hérité.*

Quelques extensions utiles

- Classes abstraites, concrètes
- Contrôler les redéfinitions
- Héritage multiple
- Redéfinir le type des propriétés héritées
- mais aussi :
  - Multi-héritage
  - Réification
  - Réflexivité

Classes abstraites / concrètes

- classe abstraite
  - classe non instanciable
  - intérêt : permet de définir un premier ensemble de propriétés, mais incomplet ⇒ les sous-classes devront le compléter
    - exemple : notre classe `uneFigure` devrait être abstraite, puisque la méthode `taille()` ne peut être complètement définie (absence de code) ; idem pour la classe `uneFigurePlane`, à cause de la méthode `surface()`.
- classe concrète
  - classe non abstraite… donc instanciable
    - exemple : notre classe `unCercle`.
- concrétisation dans quelques langages :
  - en C++ : notion de méthode *pure* (= à code indéfini)
  - en Java ou Php5 : une classe ou une méthode peut explicitement être déclarée abstraite :

```
abstract class uneFigure { …
   abstract double taille(); …
}
```

Interdire la redéfinition

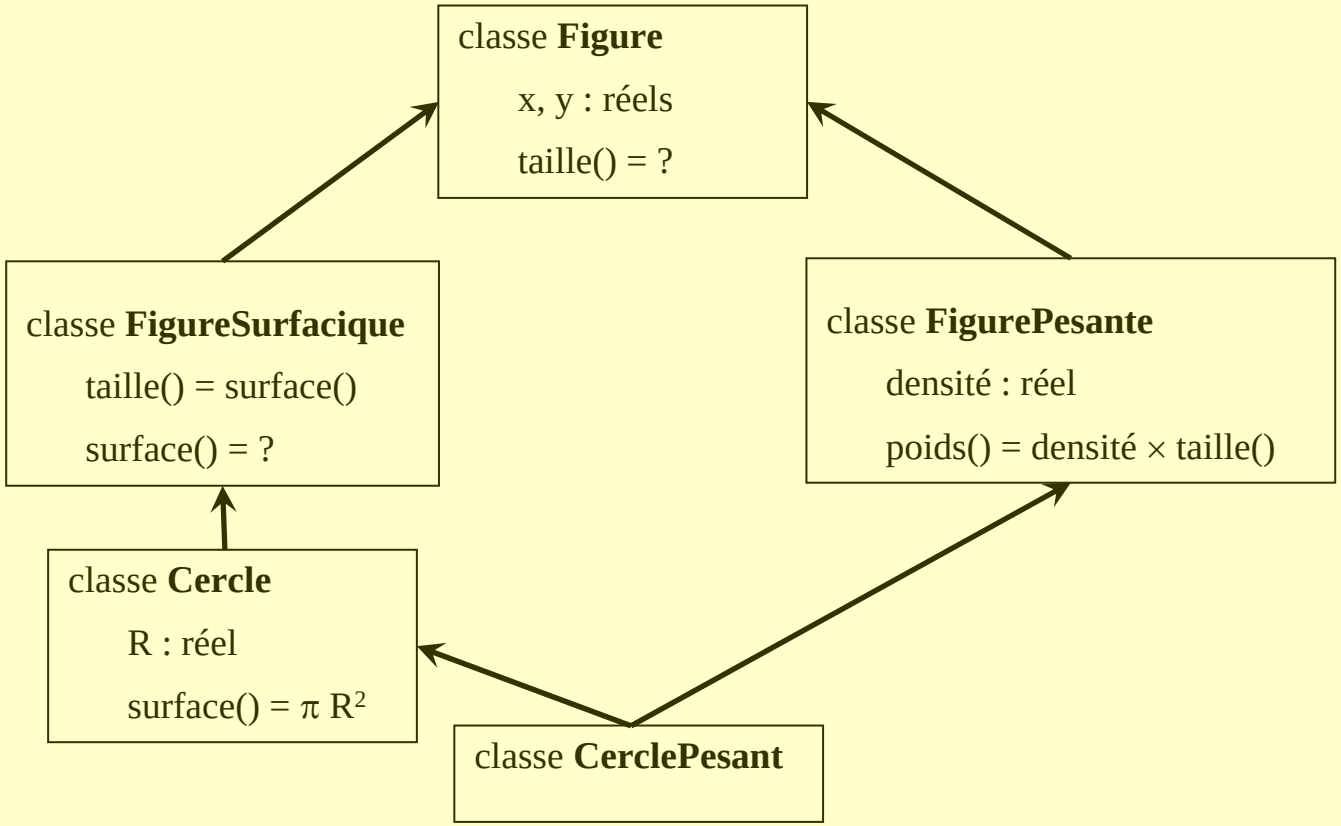
- motivation :

  empêcher un programmeur de redéfinir des parties de code critiques : mécanismes d'authentification, accès à des données protégées, …
- pouvoir écrire qu'une classe ne peut admettre de sous-classe
  - la classe n'aura donc pas de descendance
    - en C++ : n'existe pas
    - en Java : notion de classe *finale* (**`final class`** …)
- pouvoir écrire qu'une propriété ne peut pas être altérée lorsqu'elle est héritée
    - en C++ : possible à condition d'exploiter habilement les protections d'accès (**`private`**)
    - en Java : notion d'attribut ou de méthode *finale* (**`final int`** `meth(...)`)

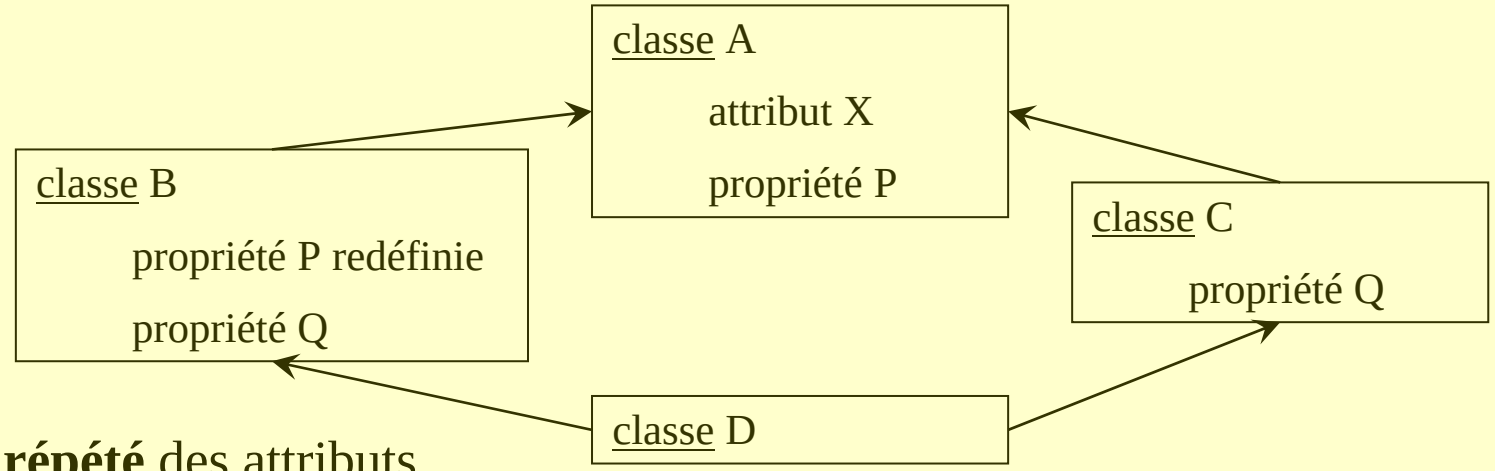
L’héritage multiple

- Extension de l’héritage simple :

  une classe peut hériter de plusieurs classes



Héritage multiple : les problèmes



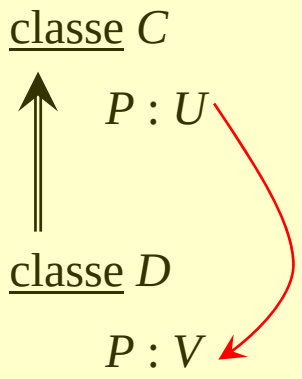
- **héritage répété** des attributs
  - toute instance de classe B porte un attribut X, idem pour toute instance de classe C
  - toute instance de classe D
    - peut se faire passer pour un objet de classe B ⇒ porte donc un attribut X (notons-le B•X)
    - peut se faire passer pour un objet de classe C ⇒ porte donc un attribut X (notons-le C•X)
    - question : B•X et C•X sont-ils deux attributs distincts ou un même attribut ?
- **conflit de masquage** d'une propriété
  - pour une instance de classe D, quelle définition de P considérer :
    - B•P, celle redéfinie dans la classe B
    - C•P, celle héritée de la classe C, égale à A•P
- **conflit de multiplicité** d'une propriété
  - pour une instance de classe D, quelle définition de Q considérer : B•Q ou C•Q

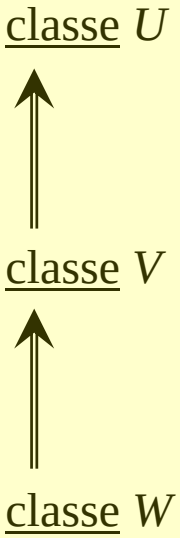
Quelques réponses

- Smalltalk
  - Héritage simple seulement
- Langage C++
  - Héritage multiple autorisé
  - Héritage répété ou non : les classes virtuelles
  - Conflits (masquage et multiplicité) à résoudre explicitement
- Langage Java
  - Deux notions de classes : les *classes* et les *interfaces*
    - Classes : peuvent être instanciées
      - Héritage simple seulement entre classes
      - Une classe peut hériter de (*implémenter*) plusieurs interfaces
    - Interfaces : non instanciables
      - Pas de définition d'attributs
      - Pas de définition de code pour les méthodes
      - Héritage entre interfaces seulement, éventuellement multiple

Héritage : redéfinir le type de propriétés héritées

- principe général :
  - altérer le type d'une propriété *P* héritée, c'est le spécialiser





- attention pour les méthodes (fonctions) :
  - redéfinition **covariante**
    - le type de l'entrée ou du résultat sont spécialisés :

$C \bullet P : U \to V$

se spécialise en

$D \bullet P : V \to W$

  - redéfinition **contravariante**
    - le type de l'entrée est spécialisé, ou celui du résultat généralisé

$C \bullet P : U \to V$

se spécialise en

$D \bullet P : V \to U$

Le multi-héritage

- constat
  - un appel de méthode de la forme
    $o.m(a_1, \ldots, a_n)$
    fait jouer à l'objet receveur o un rôle majeur
    - la liaison dynamique ne tient compte que du type de o
- question
  - cet appel ne pourrait-il pas s'écrire plus classiquement
    $m(o, a_1, \ldots, a_n)$
    sans rôle particulier de o comparé à celui des a_i
- réponse
  - la liaison dynamique doit tenir compte des types de tous les paramètres de m, pas seulement celui de o
    - exemple d'un tel langage : CLOS
- application concrète
  - la méthode testant l'égalité de deux objets

Réification

slogan : tout est objet

- toute valeur est un (*identifiant d'*) objet
  - donc y compris les entiers, les booléens, etc.
- une classe est un objet
  - instance d'une méta-classe
- un message est un objet
- une propriété est un objet
  - un attribut
  - une méthode
- une description de propriété est un objet

Réflexivité - introspection

- capacité du langage permettant aux objets de manipuler leurs propriétés
  - liste de leurs propriétés
    - leurs noms, leurs valeurs, leurs types
  - envois de messages exploitant ces informations
- intérêts
  - les objets peuvent découvrir dynamiquement ce que les autres savent faire
    - principe des *Beans* en Java
  - tout le système peut être recompilé à la volée
    - les objets peuvent changer dynamiquement leur codage en mémoire
    - le compilateur fait partie intégrante du monde des objets

Quelques travaux à étudier…

- modélisation objet en général
  - UML (méthodologie et langage de conception objet)
  - les motifs (ou patrons) de conception (*design patterns*)
    - solutions objets toutes faites, à adapter à son problème
- modélisation de systèmes physiques
  - Modelica
- langages issus de la recherche universitaire française
  - Mering, Ptitloo (J. Ferber)
  - Yafool (R. Ducournau)
  - Shirka (F. Rechenmann)
  - OCaML (M. Mauny, X. Leroy)
- programmation orientée agents
  - extension naturelle de la programmation orientée objets
    - un agent = un objet + un processus gérant son comportement

Annexe A : glossaire

- appel
- attribut
- classe
  - super-classe
  - sous-classe
- conflit
  - de masquage
  - de multiplicité
- encapsulation
- envoi
  - appel de méthode
  - appel de la super-méthode
- héritage
  - simple
  - multiple
- instance
- liaison dynamique
- message
- méthode
  - accesseur
  - constructeur
  - destructeur
  - modificateur
  - observateur
  - sélecteur
- objet
- polymorphisme
- propriété
- protection
  - privé
  - protégée
  - publique
- redéfinition

Annexe B : relations entre objets

- Nous connaissons déjà :
  - la relation de **généralisation** ou de **spécialisation** entre classes
  - la relation d'**instanciation** qui lie une instance à sa classe
- Mais l'on rencontre aussi :
  - la relation d'**association**
    - deux objets sont liés, mais peuvent exister l'un sans l'autre
      - exemple : un *étudiant* et *l'université* où il est inscrit
  - la relation d'**agrégation**
    - deux objets agrégés ne peuvent être séparés sans que l'un disparaisse
      - exemple : un objet *document* et son objet *vue graphique* associé (le document peut ne pas avoir de vue, mais une vue est nécessairement associée à un document)
  - la relation de **composition**
    - forme la plus commune d'agrégation ; relation tout - partie
  - la relation de **point de vue**
    - un objet est une représentation d'un autre objet, mais il peut en exister d'autres représentations alternatives
      - exemple : un objet *point du plan*, lequel est décrit par un objet donnant ses coordonnées cartésiennes, et par un autre objet donnant ses coordonnées polaires.
  - …