

ensiie

école nationale supérieure d'informatique
pour l'industrie et l'entreprise

Langages Orientés Objet

2008

Langages Orientés Objet - David Roussel

1

Langages orientés objet

Sommaire

- **1. Introduction**
 - 1.1. Le logiciel
 - 1.2. Historique de la conception
 - 1.3. Objectifs
 - 1.4. La réponse objet
- **2. Mécanismes orientés objet**
 - 2.1. Caractéristiques des langages exemples
 - 2.2. Caractéristiques des objets
 - 2.3. Classe & instance
 - 2.4. Accessibilité
 - 2.5. Constructeurs / Destructeurs
 - 2.6. Membres de classes
 - 2.7. Polymorphisme
 - 2.8. L'Héritage
 - 2.9. Composition
 - 2.10. Généricité
 - 2.11. Robustesse

1. Introduction

1.1 Le Logiciel

- Coût du logiciel
 - 20 / 100 lignes par jour
 - 100 / 500 lignes par semaine
 - 5000 / 25000 lignes par an

} 100.000 lignes
⇒ 4 / 20 h/a
⇒ 120 / 600 K€
- Nécessité d'évolution
 - amortissement nécessaire
 - réutilisation et/ou adaptation d'un tel objet
- Complexité
 - de spécification
 - d'implémentation

1.1 Le Logiciel

- Qualité logicielle

- Aspect externe

- Validité par rapport au cahier des charges
 - Usage :
 - Facilité d'emploi
 - Reflète le fonctionnement apparent du logiciel

Utilisateur

- Aspect interne

- Maintenance
 - Evolution
 - Portabilité
 - Réutilisabilité
 - Reflète le fonctionnement effectif du logiciel

Programmeur

1.2. Historique de la conception (1/2)

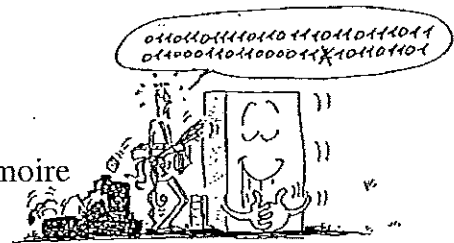
- 1.2.1. Clefs de classement

- Topologie des données
 - Topologie des traitements

- 1.2.2. Générations de langages

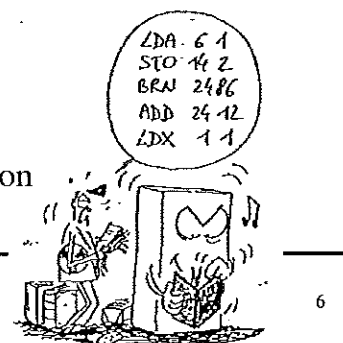
- 1^{ère} génération (1950)

- Code binaire dépendant du matériel
 - Structure de données plate
 - Sous programmes pour économiser la mémoire
 - Modification \Rightarrow réécriture



- 2^{ème} génération (1960)

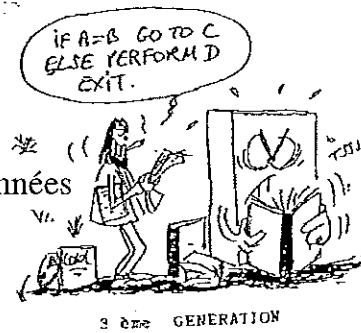
- Assembleur
 - Structure de données évoluée et globale
 - Sous programmes comme éléments d'abstraction
 - Modification \Rightarrow réécriture



1.2. Historique de la conception (2/2)

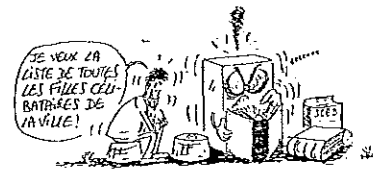
– 3^{ème} génération (1970)

- Langages procéduraux : C, Pascal, Cobol
- Structure de données évoluée et globale
- Organisation hiérarchique et en modules indépendants avec leur propres structures de données
 - Eléments d'abstraction
 - Modules de traitements
 - Modules utilitaires
- Modifications
 - Locales aux modules
 - De la structure \Rightarrow réécriture



– 4^{ème} génération (1985)

- Langages orientés "Tâches"
- Les langages objets se situent dans la 3^{ème} ou la 4^{ème} génération



– 5^{ème} génération

- Programmation en langage naturel ???
 - Exprimer le problème correspond souvent à une bonne partie de sa solution ...



1.3. Objectifs

- Améliorer la qualité des logiciels à travers :
 - Portabilité : isoler les détails de l'implémentation
 - Vrai pour tous les LOO
 - Compatibilité : avec les langages antérieurs :
 - Langage $C \subset C++$
 - Validité, vérifiabilité :
 - Développement de méthodologies et outils de GL
 - Intégrité : protection des données et du code interne :
 - Données privées
 - Traitements généralement publics
 - Fiabilité : Fonctionnement "dégradé" possible :
 - Non, mais on peut établir un "contrat" de fonctionnement
 - Maintenance, extensibilité : héritage
 - Caractéristique commune à tous les LOO
 - Réutilisabilité :
 - Généricité
 - Efficacité : Proche de la machine
 - Ergonomie :
 - Facilité d'utilisation et d'apprentissage

1.4. La réponse objet

- Structuration :
 - décomposition, regroupement
- Associer données et traitements :
 - types abstraits, classes
- Limiter les accès :
 - Masquage : Interface externe publique
- Typer, caractériser :
 - typage fort
- Gestion dynamique des éléments :
 - Création, destruction, polymorphisme
- Généraliser :
 - Héritage, Généricité

2. Les mécanismes orientés objet

Avant propos

Langages utilisés : Java, C++, SmallTalk

Attention : les exemples utilisés sont souvent incomplets et ne peuvent être compilés tels quels.

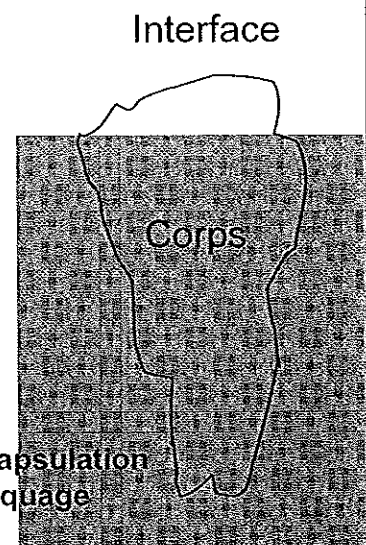
2.1. Caractéristiques des langages-exemples

- Java, C# *(NetBeans)*
 - Orienté objet *(NetBeans)*
 - Structures de contrôle semblables au C
 - Compilé
 - Nécessite une machine virtuelle
- C++
 - Orienté objet
 - Structures de contrôle du C : $C \subset C++$
 - Compilé
 - Ne nécessite pas de machine virtuelle
- SmallTalk
 - Objet
 - Les structures de contrôle sont des méthodes des objets
 - Compilé au vol (*Just In Time*)
 - Nécessite une machine virtuelle

méthode d'objet

2.2. Caractéristiques des objets

- Caractéristiques
 - Pas de structure de données globales (les données restent **dans** l'objet)
 - Communication par messages (méthodes des objets)
- Paradigme de l'iceberg
 - Interface externe accessible
 - des fonctions
 - une fonctionnalité
 - Implémentation interne (Corps) cachée
 - des données (accessibles uniquement par les fonctions ci-dessus)
 - des fonctions internes



2.3. Classe & instance

Classe & instance

Définition de la classe :

Données (informations passives)
+ *Traitements* liés à ces données (informations actives)
⇒ Type abstrait

Définition de l'instance :

La classe est un type
Les variables de ce type sont des *instances* de la classe

Deux étapes :

1. Définition de la classe ⇔ type
2. Déclaration et création des instances ⇔ variables
→ Au moment de leur utilisation

Types

- Une classe définit un « type » au sens des langages procéduraux (C, Pascal, ...).
- Java :
 - Les types simples existent : byte, short, int, char (entiers), float, double (flottants), boolean. Mais ils ne sont pas considérés comme des classes.
 - Les classes peuvent être vues comme des types.
- C++ :
 - Les types simples existent (ceux du C) et sont considérés comme des classes.
 - Les classes forment des types (compatibilité avec le C).
- SmallTalk : langage faiblement typé.
 - Les types simples n'existent pas.
 - Tout est objet, même les structures de contrôle.

Exemple

Objet : «*Compte bancaire*»

Données membres - Attributs (variables d'instance) :

numéro et *solde*

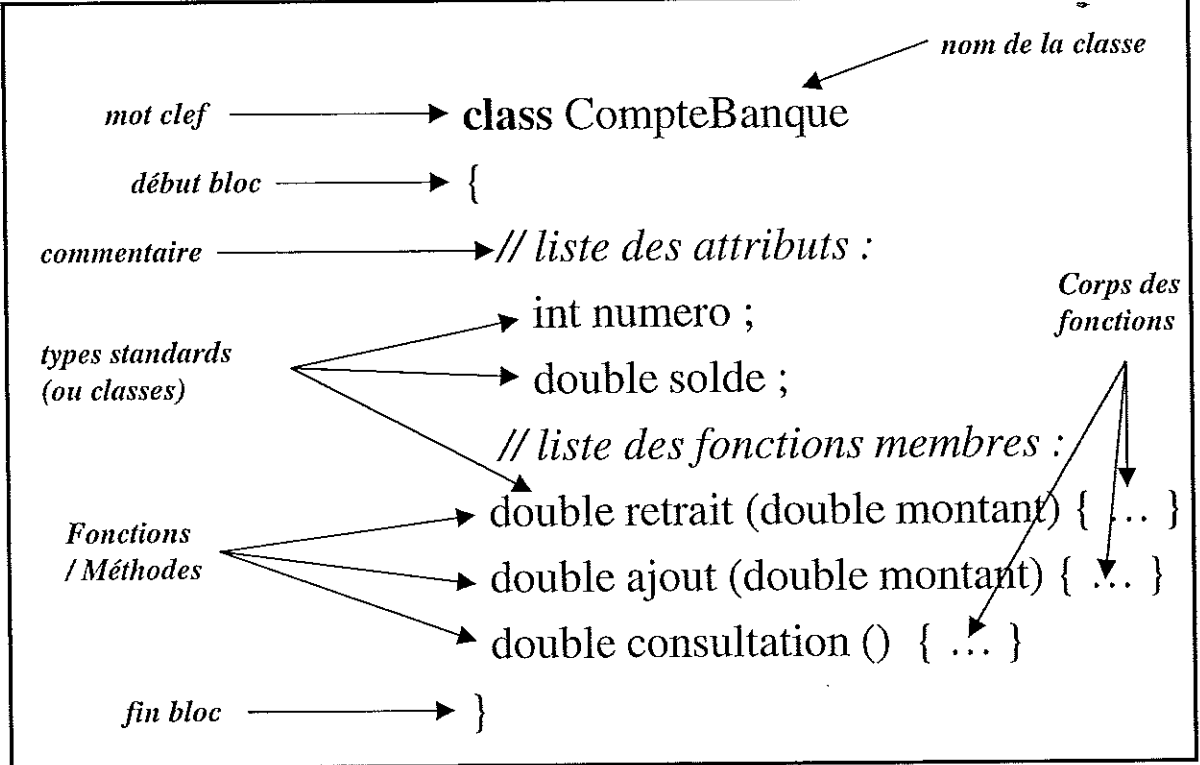
Fonctions membres - Méthodes (traitements) :

retrait, *ajout* et *consultation*

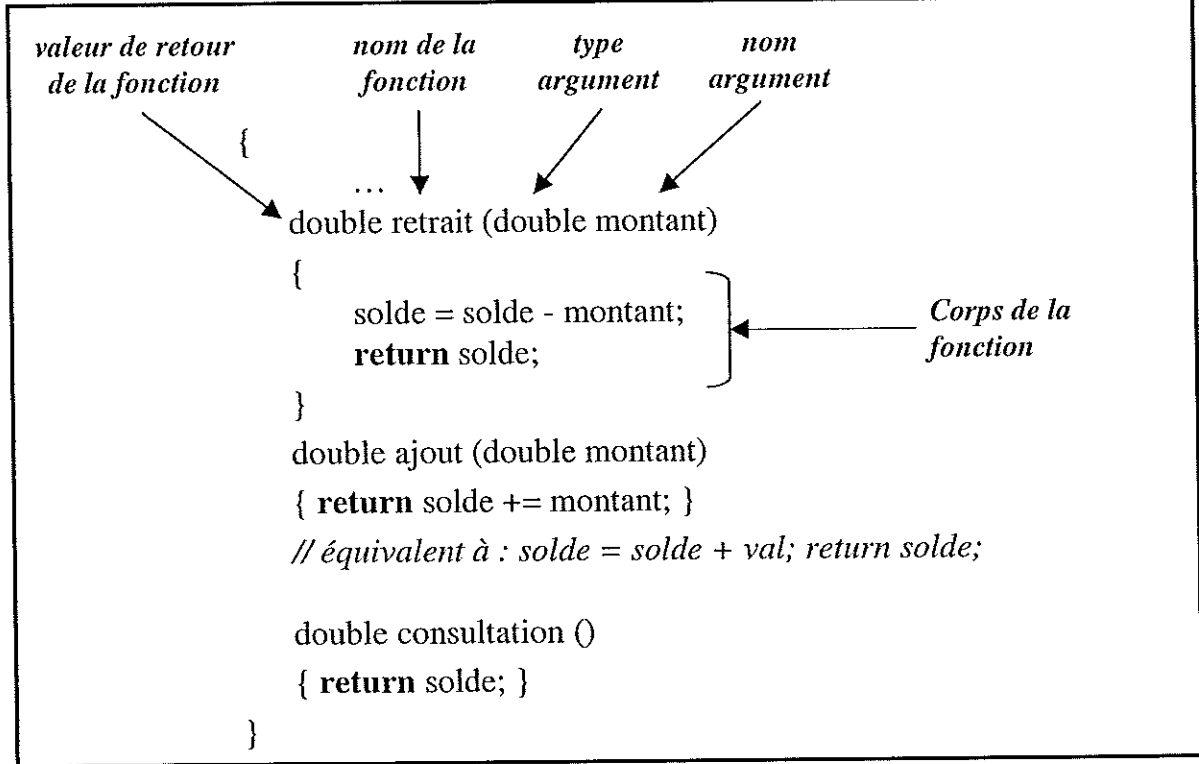
Définition de la classe :

- Liste des attributs et fonctions membres
 - *numéro* : entier
 - *solde* : double
 - *retrait* et *ajout* : renvoient une valeur double, 1 argument double
 - *consultation* : renvoie une valeur double, pas d'argument

Exemple - Déclaration



Exemple - implémentation



Exemple - *Instanciation*



Déclaration & création d'une instance :

nom de la classe (type) nom de l'instance (variable) déclaration de l'instance

Opérateur « new » CompteBanque monCompte ; création de l'instance

monCompte = new CompteBanque () ; * monCompte est une référence vers un CompteBanque

Autres cas :

- Constantes : «final»
 final Chaine NULLE = **new** Chaine("");
- Tableaux : «[]»
 Deux formes de déclaration {
 Point vecteur[] = **new** Point[10]; Déclaration/ Création de tableaux
 Point[][] matrice = **new** Point[10][100]; Création des cases de tableaux
 for (int i=0; i<5; i++) { vecteur[i] = **new** Point() ; }

Exemple - *Accès aux membres*



Accès :

Toujours par rapport à une structure de données existante, réelle, effective, c'est-à-dire une *instance*

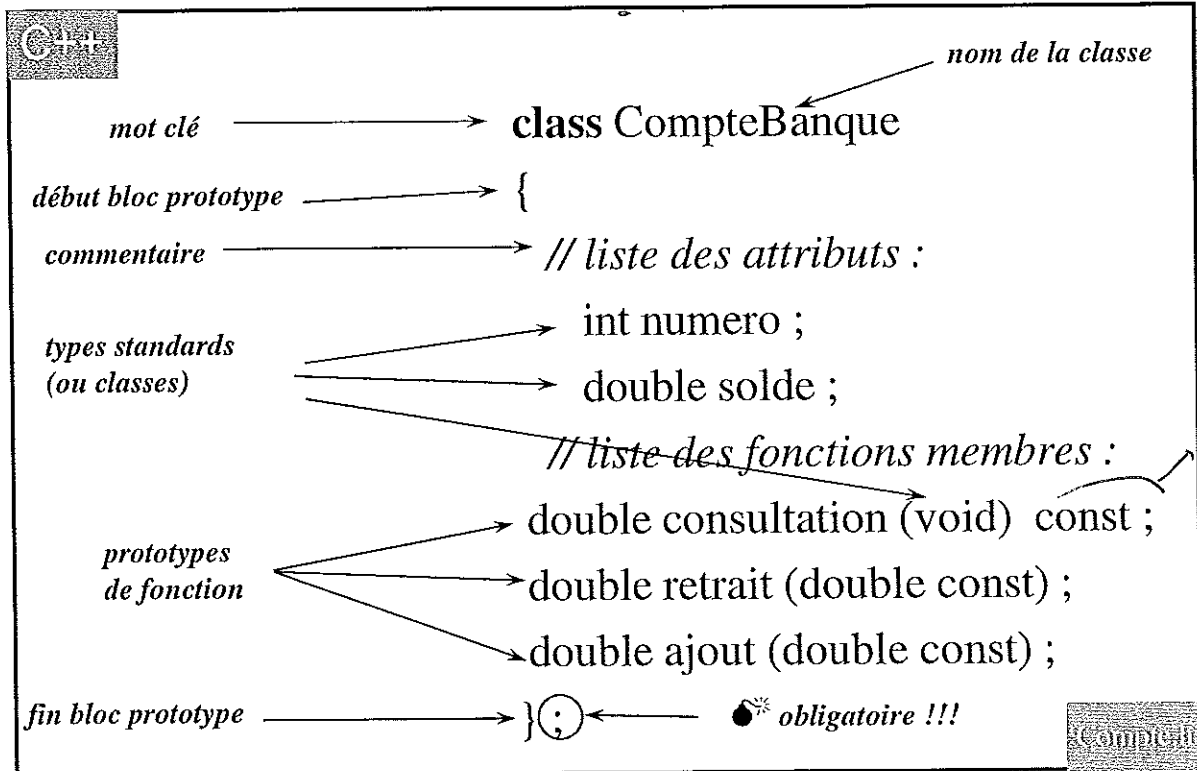
Syntaxe :

<u>Depuis :</u>	<u>Opérateur :</u>
• une instance	« . » (point)
• un élément d'un tableau	« . » (point)

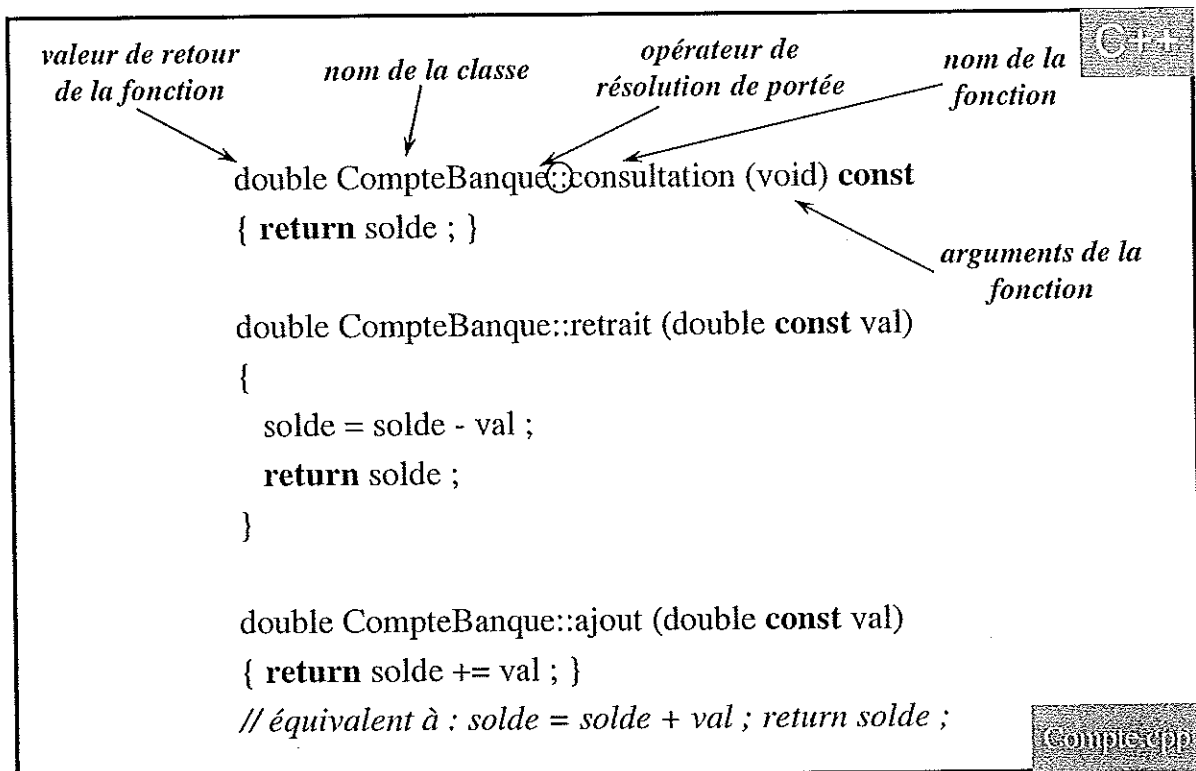
Exemples :

```
CompteBanque monCpt = new CompteBanque();  
CompteBanque[] tabCpt = new CompteBanque[3];  
for (int i=0; i<3; i++) tabCpt[i] = new CompteBanque();  
... monCpt.numero = ...  
... monCpt.ajout (12); ...  
... tabCpt[1].consultation () ...  
CompteBanque.solde  
CompteBanque.retrait (23)      Faux ! (CompteBanque est un type, pas une variable)
```

Exemple - Déclaration



Exemple - Implémentation



Exemple - Instanciation

Déclaration d'une instance :

nom de la classe
(type)

nom de l'instance
(variable)

CompteBanque monCompte ;

Autres cas :

- Constantes : «const»
Chaine const NULLE ("");
- Tableaux : «[]»
Point matrice [10][100];
- Pointeurs : «*»
Liste * ptr ; ... ; ptr = new Liste(...);

Différents types d'instances (1/2)

```
class MaClass {  
private:  
    int a;  
public:  
    MaClass(const int value);  
    ~MaClass(void);  
    int valeur(void) const;  
};
```

```
#include <iostream>  
#include "MaClass.h"  
using namespace std;  
MaClass::MaClass(const int value) : a(value) {  
    cout << "Création : " << a << endl;  
}  
MaClass::~MaClass(void) {  
    cout << "Destruction : "  
        << a << endl;}  
int MaClass::valeur(void) const {  
    return a;  
}
```

```
#include <iostream>  
#include "MaClass.h"  
using namespace std;  
MaClass * creeInstance(int valeur)  
{  
    // Instance dynamique  
    MaClass * nouvelle = new MaClass(valeur);  
    return nouvelle;  
    // A la fin de ce bloc nouvelle  
    // disparaît mais le pointeur  
    // qu'elle contient est renvoyé  
    // dans la pile et peut donc  
    // être transmis à un autre pointeur  
}
```

Différents types d'instances (2/2)

C++

```
...
int main (int argc, char** argv)
{
    for (int i=1; i < argc; i++)
    {
        int val;
        sscanf(argv[i], "%d", &val);
        // Instance statique
        //(n'a d'existence que dans ce bloc)
        MaClass c(val);
        cout << "Valeur = " << c.valeur() << endl;
        // A la fin de ce bloc c est détruit
        // à chaque tour de boucle.
    }

    MaClass * pa = creeInstance(7);

    cout << "Valeur de pa : " << pa->valeur() << endl;

    // Si on ne détruit pas explicitement pa,
    // la mémoire allouée dans la fonction
    // "creeInstance" ne sera jamais libérée.
    delete pa;

    return 0;
}
```

test(MaClass.cpp)

Exemple - Accès aux membres

C++

Accès :

Toujours par rapport à une structure de données existante, réelle, effective, c'est-à-dire une *instance*

Syntaxe :

Depuis :

- une instance
- un élément d'un tableau
- un pointeur

Opérateur :

- « . » (point)
- « . » (point)
- « -> » (moins supérieur)

Exemples :

```
CompteBanque monCpt, tabCpt [3], * unCpt, * unTabCpt ;
... monCpt.numero; ... monCpt.ajout (12); ...
... tabCpt [1].consultation (); ...
... unCpt = new CompteBanque; ... unCpt->ajout(10);
... unTabCpt = new CompteBanque [10]; ... unTabCpt[7].ajout(35);
CompteBanque solde
CompteBanque retrait (23) Faux ! (CompteBanque est un type, pas une variable)
```

Accès à soi même

- Comment faire référence à soi même à l'intérieur d'une classe ?
 - Java & C++ : mot clé « this »
 - en C++ : this est un pointeur sur l'instance courante
 - en Java : this est une référence à l'instance courante
 - SmallTalk : mot clé « self »
- Utilité : résolution de conflits de noms

```
void init(double solde,  
          int numero)  
{  
    this.solde = solde;  
    this.numero = numero;  
}
```

Java

```
void CompteBanque::init  
(double solde, int numero)  
{  
    this->solde = solde;  
    this->numero = numero;  
}
```

C++

2.4. Accessibilité

Accessibilité

- But : implémenter le paradigme de l'iceberg
 - Masquer au client* le corps de la classe (partie privée)
 - Le client est limité à l'interface (partie publique)
- } Encapsulation
- Utilité
 - Modularité & cohérence
 - Les modifications du corps n'entraînent aucune modification dans l'interface présentée au client
- ⇒ Développement
- 1/ Définition des interfaces
 - 2/ Implémentation du corps des classes en parallèle !
- * : utilisateur de la classe

Notions d'accessibilité

- Domaine d'application de l'accessibilité
 - A l'intérieur d'une classe tous les membres sont accessibles
 - Les règles d'accessibilité s'appliquent à l'extérieur de la classe :
⇒ au client.
- En SmallTalk (accessibilité figée)
 - Les attributs sont privés
 - Les méthodes sont publiques ⇒ on n'accède aux attributs qu'à travers les méthodes
- En Java et en C++ (accessibilité à la carte)
 - Trois modes d'accès applicables aux membres de classes (attributs & méthodes) :
 - public
 - protégé
 - privé

Modes d'accès (1/2)

Modes d'accès - 3 cas possibles définis par des « modificateurs » d'accès :

- **Public «public»**

Tout le monde peut utiliser un tel membre

Emploi du nom du membre autorisé à l'extérieur de la classe

Exemple C++ :

```
int main (void)
{
    CompteBanque monCpt ;
    monCpt.retrait (100) ; // autorisé si retrait possède le mode public
    return 0 ;
}
```

- **Protégé «protected»**

Seules les instances de la classe ou de ses héritières peuvent l'utiliser

Emploi du nom du membre interdit en dehors de la classe, sauf héritières

Exemple Java :

```
...
CompteBanque monCpt = new CompteBanque() ;
monCpt.solde =100 ; // refusé si solde possède le mode protected
...
```

• en Java, un membre protégé est malgré tout accessible en dehors de la classe, pour les classes appartenant au même « Package »

Modes d'accès (2/2)

- **Privé «private»**

Seules les instances de la classe peuvent l'utiliser

Emploi du nom du membre interdit en dehors de la classe

- **Accès interne**

- Les méthodes ont accès à tous les membres (attributs, méthodes) de leur classe

- **Modes par défaut**

- Lorsque l'on ne spécifie aucun mode d'accès
- en Java : le mode implicite est le mode « package » équivalent au mode « public », mais limité aux classes d'un même package
- en C++ : le mode par défaut est le mode « private »
- en Smalltalk (les modificateurs n'existent pas)
 - Les méthodes sont toutes publiques
 - Les attributs sont tous protégés

} Encapsulation forte

Choix du mode d'accès

Il faut classer les membres :

- «Internes» : protégés ou privés
 - typiquement les attributs et les méthodes utilitaires
- «Externes» : publiques
 - méthodes d'accès aux attributs (ex : consultation)
 - opérations de la classe (ex : retrait, ajout)

Encapsulation forte

→ Nouvelle définition de la classe CompteBanque :

```
class CompteBanque
{
    public :
        double consultation(void) const ;
        double retrait (double const) ;
        double ajout (double const) ;
    protected :
        int numero ;
        double solde ;
};
```

```
class CompteBanque
{
    public double consultation(){...}
    public double retrait(double montant){...}
    public double ajout (double montant){...}

    protected int numero ;
    protected double solde ;
};
```

Java

Accès aux classes

Java

- Dans un Package Java
 - on peut spécifier la visibilité en dehors du package avec le modificateur « public » appliqué cette fois à une classe
 - Mode par défaut : « package »
 - Exemple

```
package MonPackage;

public class Visible
{
    ...
}

class Cachee
{
    ...
}
```

MonPackage.java

```
import MonPackage.*;

class maClasse
{
    Visible m = new Visible( ... )
    Cachee mu = new Cachee( ... )

    // interdit car Cachee
    // n'est pas accessible en
    // dehors du Package MonPackage
    ...
}
```

MonAppl.java

Fonctions amies : « friend »



- Les fonctions amies ont accès aux membres protégés et privés de la classes dans laquelle elles sont déclarées.
- Utilité : surcharge des opérateurs d'entrée (<<) et sortie (>>) standards.
- Exemple :

```
class CompteBanque
{
public :
...
friend ostream & operator << (ostream & out, Compte const & cpt);
friend istream & operator >> (istream & in, Compte & cpt);
protected :
int numero ;
double solde ;
};
ostream & CompteBanque::operator << (ostream & output, Compte const & cpt)
{
output << " Compte n° : " << cpt.numero ...
}
```

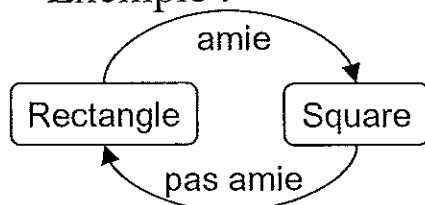
operator << n'est pas membre

Accès à l'attribut protégé « numero »

Classes amies : « friend »



- Les classes amies ont accès aux membres privés et protégés de la classe dans laquelle elles sont déclarées amies.
- Exemple :



```
// forward declaration
class Square;

class Rectangle {
int width, height;
public:
int area (void);
void convert (Square a);
};
```

```
class Square {
private:
int side;
public:
void set_side (int a);
friend class Rectangle;
};

void Square::set_side(int a) {
side=a;
}

void Rectangle::convert (CSquare a) {
width = a.side;
height = a.side;
}
```

Accès au membre privé

Initialisations

Comment initialiser les membres protégés ou privés ?

Avec une méthode d'initialisation (spécifique ou standard → constructeurs)

Elle doit être publique pour pouvoir être appelée de l'extérieur

→ Nouvelle définition de la classe CompteBanque :

```
class CompteBanque
{
    public :
        double consultation (void) const;
        double retrait (double const);
        double ajout (double const);
        void init (int const, double const);
    protected :
        int numero;
        double solde;
};
...
void CompteBanque::init(int const n,
                        double const s)
{ numero = n; solde = s; }
```



```
class CompteBanque
{
    public double consultation(){...}
    public double retrait(double montant){...}
    public double ajout (double montant){...}
    public void init (int n, double s)
    {
        numero = n;
        solde = s;
    }

    protected int numero;
    protected double solde;
};
```



2.5. Constructeurs / Destructeurs

Constructeurs / Destructeurs

- Méthodes particulières utilisées lors de la création et la destruction d'instances
- Les Constructeurs remplacent avantageusement les méthodes d'initialisation
- Implémentation dans les langages
 - C++
 - Constructeurs
 - Destructeurs
 - Java
 - Constructeurs
 - Garbage collecting pour la destruction & méthodes de terminaison (finalize) avant le garbage collecting
 - SmallTalk
 - Constructeur : Méthode new (ou tout autre méthode)
 - Garbage collecting pour la destruction

Constructeurs (1/2)

Rôle : membres prédéfinis pour l'initialisation des instances

Nom : celui de la classe

Surchargeable, comme toute méthode (plusieurs définitions possibles)

Exemple :

```
// déclaration de la classe
class CompteBanque
{
public CompteBanque ()
{ numero = 0 ; solde = 0.0 ;}

public CompteBanque (int n,
double s)
{ numero = n ; solde = s ;}

public CompteBanque (int n) ;
{ numero = n ; solde = 0.0 ;}
...
}
```

Java

```
// Prototype :
class CompteBanque
{
public :
CompteBanque (void) ;
CompteBanque (int const n,
double const s) ;
CompteBanque (int const n) ;
...
}
```

C++

```
// Implémentation :
CompteBanque::CompteBanque (void)
{ numero = 0 ; solde = 0.0 ;}

CompteBanque::CompteBanque (int const n,
double const s)
{ numero = n ; solde = s ;}

CompteBanque::CompteBanque (int const n)
{ numero = n ; solde = 0.0 ;}
```

CompteBanque.h

CompteBanque.java

CompteBanque.cpp

Constructeurs (2/2)

Retour :

Renvoie une instance de la classe (déclaration du type de retour implicite)

Appel :

Uniquement lors de la création d'une instance (1 seule fois par instance, a priori), au moment de la déclaration (C++), lors d'une allocation dynamique de mémoire ou d'une recopie dans la pile (C++ : argument ou valeur de retour)

Exemple : déclarations d'instances

C++
`CompteBanque c1 (1, 100.0), c2 (2), c3 ; ...`

Java
`CompteBanque c1 = new CompteBanque(1, 100.0); ...`

❖* `c1.CompteBanque (1, 100.0)` est **interdit** !

En C++ généralisé aux types standards. Exemple : `int i(3) ;`

Appels implicites aux constructeurs

- Java :

- Aucun appel implicite, toutes les instances sont créées à travers l'opérateur **new** qui fait appel aux constructeurs.

- Passage d'arguments : par référence uniquement pour les objets, il n'y a donc qu'une recopie de la référence dans la pile d'appel.

- Exemple : `int maMethode (MonObjet o, int valeur)` → Valeur
 → Référence

- C++ :

- Appels implicites:

- Déclaration d'une instance statique : `MonObjet o ;`

- Déclaration suivie d'une affectation: `MonObjet o; MonObjet o2 = o1;`

- Passage d'arguments : lors de passage d'arguments par valeurs :

- `int maMethode (MonObjet o, int valeur)` → Valeurs

Recopie dans la pile

- Valeur de retour d'une méthode :

- `MonObjet maMethode(void) { MonObjet o; ... ; return o; }`

Constructeur par défaut (1/2)

- Lorsque aucun constructeur n'est défini :
 - ⇒ Il existe un constructeur par défaut (sans arguments).
- S'il existe au moins un constructeur :
 - Quel qu'il soit : avec ou sans arguments.
 - ⇒ Le constructeur par défaut n'existe plus.
- Lorsqu'il existe un ou plusieurs constructeurs dans une classe :
 - ⇒ on continue d'appeler « Constructeur par défaut » tout constructeur défini sans arguments.

Constructeur par défaut (2/2)

Constructeur sans arguments

Exemple :

```
class CompteBanque
{
  public :
    CompteBanque (void) ;
  ...
} ;
```

```
CompteBanque::CompteBanque (void)
{ numero = 0 ; solde = 0 ;}

int main ()
{
  CompteBanque cpt ; ← Appel implicite
  ...
  return 0 ;
}
```

```
class CompteBanque
{
  public CompteBanque ()
  { numero = 0 ; solde = 0 ;}
  ...
}
...
class mainapp
{
  public static void main (String
args[])
  {
    CompteBanque cpt =
      new CompteBanque() ; ← Appel explicite
    ...
    return 0 ;
  }
}
```

Si **aucun** constructeur n'est défini : il existe « par défaut » un constructeur par défaut

Initialisation des attributs

C++

Si opération complexe → affectation explicite dans le corps du constructeur

Si opération simple → liste d'initialisation des attributs (qui contient la liste des constructeurs des attributs)

Exemple :

```
class CompteBanque
{
public :
    CompteBanque (int const n, double const val, String const & nom) ;
protected :
    int numero ;
    String titulaire ;
    double solde ;
};
```

```
CompteBanque::CompteBanque (int const n, double const val,
String const & nom)
```

```
: numero (n), titulaire (nom), solde (0)
```

Liste
d'initialisation*

```
{
    if (val >= 500)
        { solde = val ; }
    else
        { cout << "Versement initial trop faible" << endl ; }
}
```

Corps du
constructeur

* : dans l'ordre des déclaration des attributs

Constructeur de copie

C++

Syntaxe : `nomClasse (nomClasse const &, ...)` ;

Si ce constructeur n'est pas défini : il y a recopie bit à bit

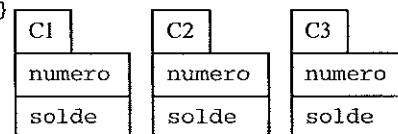
Deux types d'appels :

- Appel explicite lors d'une initialisation par copie
- Appel implicite lors de la recopie dans la pile d'un argument par valeur ou valeur de retour

Exemple :

```
class CompteBanque { ... } ;
void AfficheCompte (CompteBanque cpt) { ... }

int main ()
{
    CompteBanque c1 ;
    CompteBanque c2 (c1) ; // appel explicite , initialisation par copie
    CompteBanque c3 = c1 ; // appel explicite, autre syntaxe,
                          // initialisation par copie,
                          // différent de l'affectation c3 = c1
    ...
    AfficheCompte (c1) ; // appel implicite, par recopie d'un argument
                       // (passage par valeur)
}
```



Constructeur de copie

Java

Syntaxe : `nomClasse (nomClasse objet, ...)` ;

Si ce constructeur n'est pas défini : il faut donc l'écrire !!!

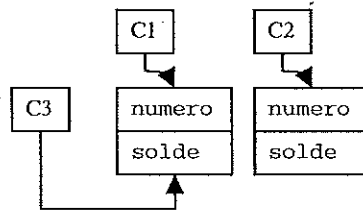
- ⇒ les deux objets pointeront donc vers une **même** zone mémoire
- ⇒ Il ne s'agit donc **pas** de deux instances différentes

Un seul type d'appel :

- Appel explicite lors d'une initialisation par copie
- Une déclaration suivie d'une affectation ne fait **pas** appel au constructeur de copie.

Exemple :

```
class CompteBanque
{ ...
void AfficheAutreCompte (CompteBanque cpt) { cpt.consultation() }
public static void main (String args[]) {
    CompteBanque c1 = new CompteBanque(1, 1000.0);
    CompteBanque c2 = new CompteBanque(c1) ;// appel explicite,
                                           // initialisation par copie
    CompteBanque c3 = c1 ; // copie de la référence de c1 dans c3
    c2.AfficheAutreCompte (c1) ; // passage de la référence de c1
}
```



Gestion mémoire

- Gestion implicite de la mémoire : Langages Objets purs (Java, SmallTalk)
 - Création d'instances : **new**
 - `MaClasse monObjet = new MaClasse (...)` ;
 - Destruction d'instances :
 - Automatique, dès qu'il n'y a plus aucune référence vers ces instances : **Garbage Collecting**
 - On peut néanmoins, implémenter la méthode « finalize » en Java
 - Nettoyage des instances avant Garbage collecting.
 - `protected void finalize() throws Throwable { ... }`
- Gestion explicite de la mémoire : C++
 - Création d'instances
 - Statiques : déclaration → `MaClasse monObjet (...)` ;
 - Dynamiques : **new** → `MaClasse * monObjet = new MaClasse (...)` ;
 - Destruction d'instances dynamiques
 - `delete` : appel au **Destructeur** → `delete monObjet;`

Rôle :

Fonction membre dédiée au «démontage» des instances

Syntaxe :

~NomClasse (void) ;

Appel :

Appel implicite automatique :

lorsque l'instance n'est plus définie

(sort de sa zone de portée, qui est le bloc où elle a été déclarée)

Appelé aussi pour chacun des attributs de la classe

Appel explicite avec le mot clé **delete** pour une instance créée dynamiquement

Non surchargeables (prototype figé)

Destructeurs - Exemple

```
class Toto
{
    public :
        Toto () ; // constructeur par défaut
        ~Toto () ; // destructeur
};

Toto::Toto () { cout << "construction" << endl ; }
Toto::~Toto () { cout << "destruction" << endl ; }

void f () { Toto t ; cout << "appel de f" << endl ; }

int main ()
{
    f () ;
    Toto t ;
    return 0 ;
}

// résultat de l'exécution :
construction }
appel de f    } f () ;
destruction  }
construction Toto t ;
destruction  return 0 ;
```

Constructeurs / Destructeurs

- C++
 - Ne sont pas appelés pour les attributs de type pointeur
⇒ Nécessite un appel explicite avec l'opérateur **new**
 - Pour les tableaux :
 - Le constructeur par défaut est appelé pour chacun des éléments
 - idem pour le destructeur
- Java
 - Les constructeurs doivent être appelés explicitement avec le mot clé **new**.
 - Pour les tableaux :
 - Les constructeurs doivent là aussi être appelés explicitement pour chacun des éléments. (voir Exemple - Accès *aux membres* en Java page 20)

2.6. Membres de classe

Membres de classes

- Éléments globaux à une classe : indépendants des instances
- Attributs de classes
 - Une seule valeur, partagée par toutes les instances de la classe
 - Mêmes limitations d'accès que pour les autres membres
 - variable de classe \neq variable d'instance (attribut ordinaire)
 - Les variables de classes sont stockées dans la classe elle même
 - Alors que les variables d'instances sont stockées dans les objets de cette classe (les instances).
- Méthodes de classes
 - Les méthodes de classe ne peuvent accéder qu'aux **attributs** statiques de la classe.
 - On utilise les méthodes de classes
 - Pour manipuler les variables de classes
 - Pour les opérations concernant plusieurs instances de la classe
 - Exemple : distance entre deux Points (voir exemple suivant)

Membres de classe - C++

- mot clé `static`
- Initialisé à l'extérieur du prototype de la classe quel que soit le mode d'accès.
- Exemple : modification d'une variable de classe

```
class A
{
    public : static int st ;
};

int A::st = 10 ;// initialisation à l'extérieur du prototype de la
               classe
int main ()
{
    A a1, a2 ;
    cout << "a2.st = " << a2.st << endl ;// affichage : a2.st = 10 ;
    a1.st = 33 ;
    cout << "a2.st = " << a2.st << endl ;// affichage : a2.st = 33 ;
    return 0 ;
}
```

Membres de classe - Java

- mot clé static
- Exemple : nombre d'instances & opération sur 2 instances

```
class Point2D
{
    protected double x, y;
    protected static int nbPoints = 0;
    ...
    public Point2D (double x,
                    double y)
    {
        this.x = x; this.y = y;
        nbPoints++;
    }
    public static double distance
    (Point2D p1, Point2D p2)
    {
        double dx = p2.x - p1.x;
        double dy = p2.y - p1.y;
        return(Math.sqrt( (dx*dx) +
                          (dy*dy) ));
    }
}

class testPoint2D
{
    // test de la classe Point2D (méthode main)
    public static void main (String args[])
    {
        Point2D p1, p2;
        p1 = new Point2D(0.0, 0.0);
        p2 = new Point2D(1.1, 2.0);
        double dist1 = p1.distance(p1, p2);
        double dist2 = Point2D.distance(p1, p2);
        System.out.println(
            "Nombre de points : " +
            Point2D.nbPoints);
    }
}
```

Pas de sens

Membres de classe - SmallTalk

- Metaclass
 - Chaque classe est une instance de sa métaclasse
 - Les variables de classes sont des variables d'instance de la métaclasse.
 - Les membres de classe sont donc des membres d'instance de la métaclasse.
- Exemple :

2.7. Polymorphisme

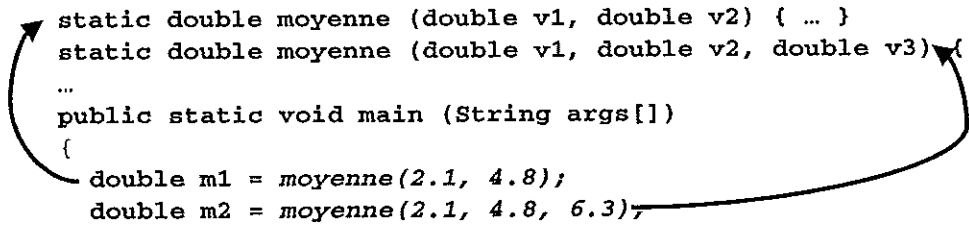
Polymorphisme - *La surcharge*

- Capacité de donner le même nom à des méthodes «similaires» ayant des listes d'arguments distinctes par le **nombre** ou le **type** des arguments.

1. Identification par le **nombre** des arguments

Exemple Java : *moyenne de 2 ou 3 valeurs*

```
...
static double moyenne (double v1, double v2) { ... }
static double moyenne (double v1, double v2, double v3) { ... }
...
public static void main (String args[])
{
    double m1 = moyenne(2.1, 4.8);
    double m2 = moyenne(2.1, 4.8, 6.3);
}
```



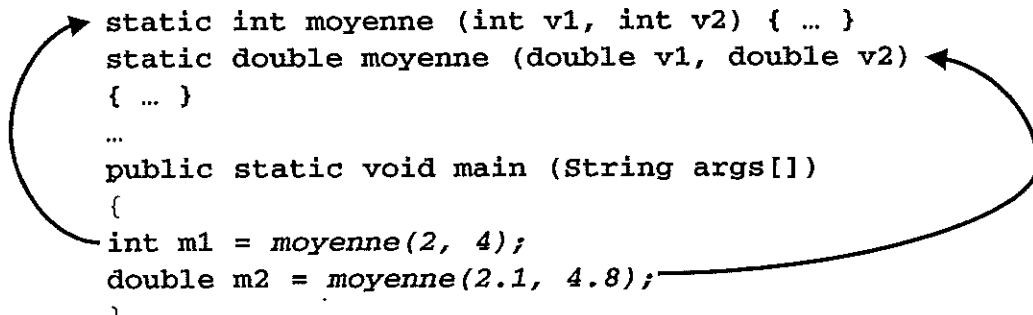
Valable pour Java, C++, Smalltalk, ...

Polymorphisme - *La surcharge*

2. Identification par le **type** des arguments

Exemple Java : *moyenne de 2 valeurs entières ou réelles*

```
...
static int moyenne (int v1, int v2) { ... }
static double moyenne (double v1, double v2)
{ ... }
...
public static void main (String args[])
{
int m1 = moyenne(2, 4);
double m2 = moyenne(2.1, 4.8);
}
```



Valable pour Java, C++
Smalltalk est non typé

Polymorphisme - *La surcharge*

- **Pas** de prise en compte du type de la valeur de retour pour l'identification de l'appel
 - Valable pour Java, C++, Smalltalk

- Exemple C++: *moyenne de 2 valeurs entières ou réelles*

```
...
double moyenne (double const v1, double const v2) ;
int moyenne (int const v1, int const v2) ;
double moyenne (int const v1, int const v2) ;
// Erreur de compilation
...
```

- Respecter autant que possible la sémantique du nom de la fonction

Polymorphisme - Conversion de types

Attention aux conversions de types implicites

Exemple C++: *moyenne de 2 valeurs entières ou réelles*

```
double moyenne (double const v1, double const v2) ;
int moyenne (int const v1, int const v2) ;

int main ()
{
    float f (2) ; // Conversion automatique de l'entier en réel
                // Dans ce sens, il n'y a aucun risque de perte
                // d'information

    char c ('x') ;
    moyenne (f, f) ; // Appel de double moyenne (double const v1,
                    // double const v2) ;
    moyenne (c, c) ; // Appel de int moyenne (int const v1, int const v2)
    moyenne (f, 2) ; // Erreur de compilation :
                    // Ambiguïté, ne sait pas lequel appeler

    return 0;
}
```

Valable pour C++ & Java
Smalltalk est non typé

Polymorphisme - Les constructeurs

- Les constructeurs sont soumis aux mêmes règles que les autres méthodes pour la surcharge
- Comment appeler un constructeur d'une même classe dans un autre constructeur ?
 - Java
 - avec le mot clé `this`
 - `Compte(String titulaire) { this(); ...}`
Rq : l'appel à l'autre constructeur doit alors être la première instruction du constructeur
 - C++
 - impossible pour une même classe (risque de boucles)
 - solution : partager une méthode d'initialisation entre constructeurs.
 - Smalltalk
 - Il n'y a pas vraiment de constructeur, seulement la réimplémentation de la méthode `new`.



Fonctions et opérateurs membres sont surchargeables

Surcharge d'opérateurs :

- L'arité et la priorité sont imposés

Exemples :

opérateur ~ : arité 1 et priorité 14

opérateur / : arité 2 et priorité 13

- Se définit comme pour une fonction dont le nom est :

operator *nomOperateur* (liste arguments)

Exemple de prototype d'un opérateur membre de classe :

NomClasse NomClasse::operator + (type mode arg) ;

- L'argument gauche doit toujours être une instance.
- Pour un opérateur membre d'une classe, il s'agit de l'instance courante, qui est alors implicite



```
class Compte
{
    public :
        double solde () ;
        double operator + (double const) ;
        double operator - (double const) ;
    protected :
        double val_solde ;
} ;

double Compte::operator + (double const v)
{ return val_solde += v ; }

double Compte::operator - (double const v)
{ return val_solde -= v ; }

int main ()
{
    Compte cpt ;
    cpt + 10. ; // équivalent à cpt.operator + (10.) ;
    cpt - 2. ;
    return 0 ;
}
```

instance courante

- Permet de donner une valeur par défaut aux arguments d'une méthode
 - Permet donc implicitement de présenter plusieurs surcharges d'une même méthode simultanément.
- Exemple :

Définition	Utilisation
<pre> class Forme { private: int x_pos; int y_pos; int color; public: Forme (int x, int y, int c); void deplace(int dx, int dy); }; Forme::Forme(int x, int y, int c = 1) : x_pos(x), y_pos(y), color(c) {} void Forme::deplace(int dx = 0, int dy = 0) {x_pos+=dx; y_pos+=dy;} </pre>	<pre> int main(void) { Forme maForme2(10,10); Forme maForme3(10,10,2); maForme2.deplace(); maForme2.deplace(3); maForme3.deplace(3,4); return 0; } </pre>

Valeurs par défaut

2.8. L'Héritage

Concept et réalisation

Concept

Définir une classe par spécialisation et/ou extension d'une autre classe

Attribution de manière automatique des propriétés (spécifications) d'une classe dite «*mère*» à une classe dite «*héritière*»

Caractéristique des vrais langages orientés objet

Réalisation

Tous les membres d'une classe *mère* sont dans la classe *héritière*

Des spécialisations sont possibles :

- sur la réalisation des traitements : redéfinition, surcharge
- sur les propriétés d'accessibilité

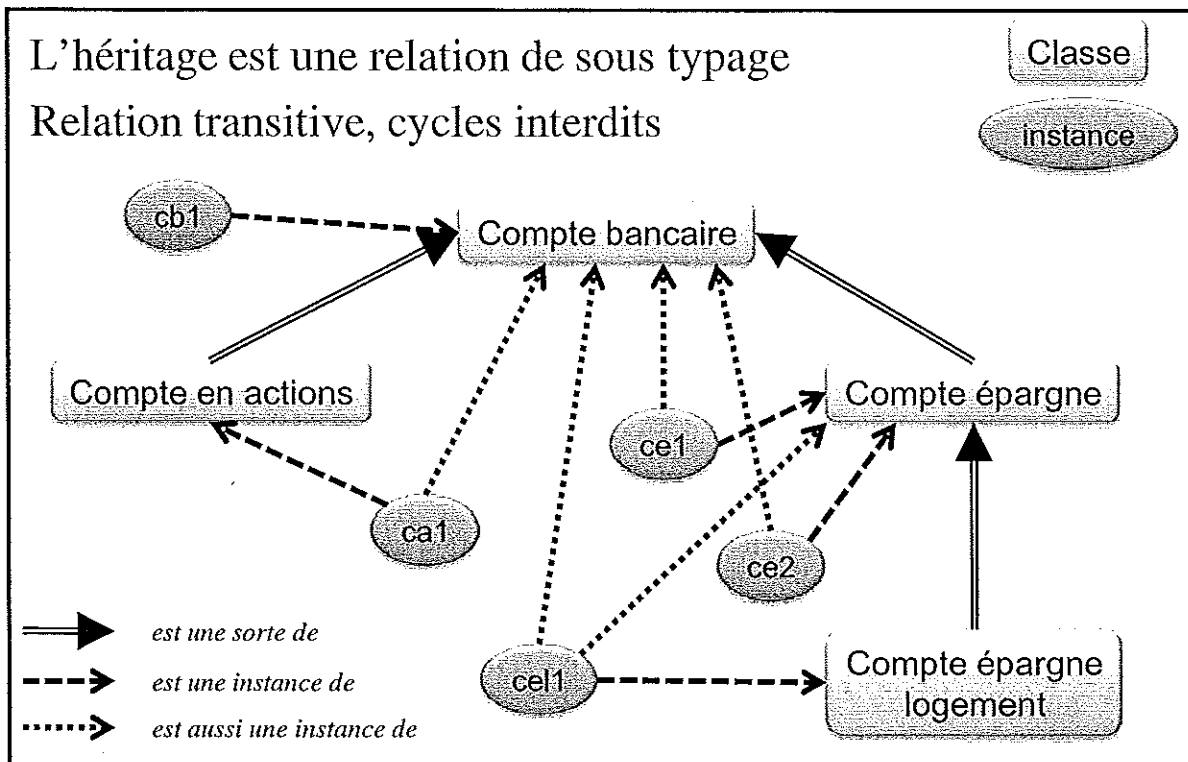
Des extensions peuvent se faire

- par ajout de nouveaux membres

Abstraction : sous typage

L'héritage est une relation de sous typage

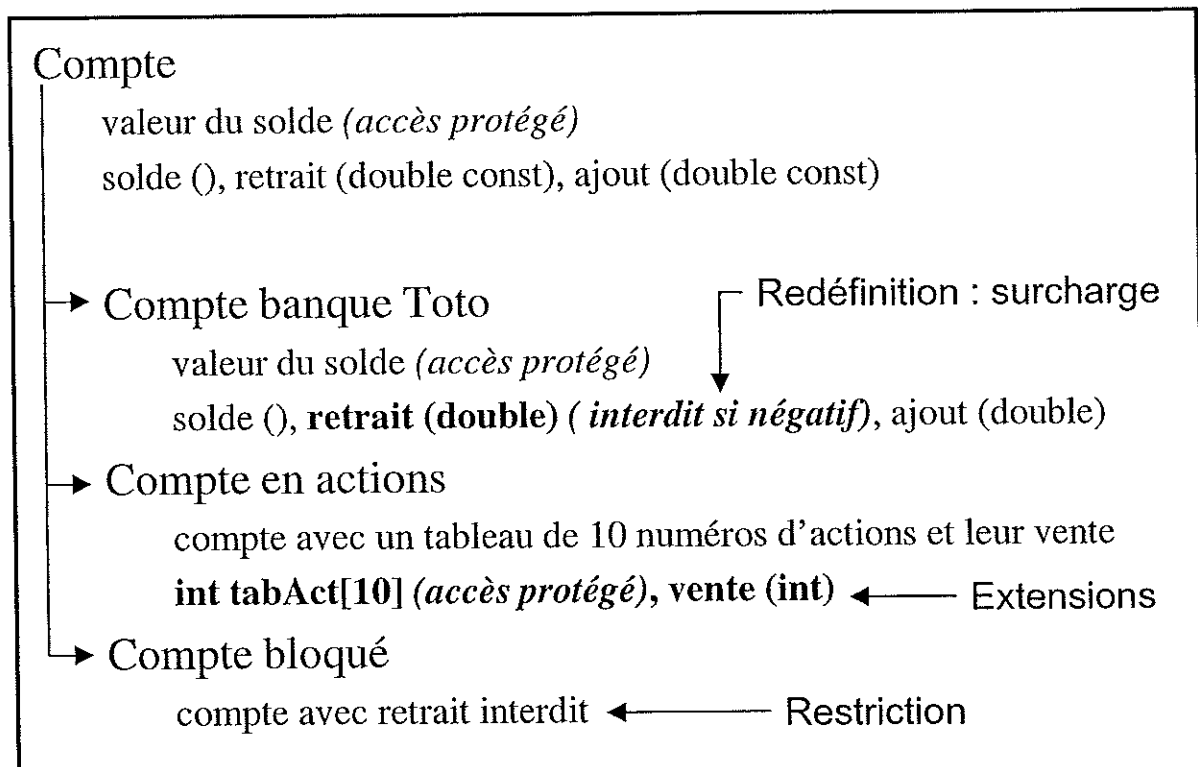
Relation transitive, cycles interdits



Terminologie

- Héritage (1 niveau ou plus) :
 - «mère», «ancêtre», «parente», «super-classe»
 - «héritière», «fille», «dérivée», «sous-classe»
- Arbre d'héritage :
 - Toutes les classes descendent implicitement d'une superclasse « Object » (Java, Smalltalk)
 - Sans arbre d'héritage : Une classe peut ne pas avoir d'ancêtre (C++)
- Héritage simple : (Java, Smalltalk)
 - Une et une seule classe mère
 - Java : Une classe hérite d'une seule autre mais peut implémenter plusieurs interfaces (voir plus loin)
- Héritage multiple : (C++)
 - Une classe peut avoir plusieurs classes mères
 - Difficultés pour l'héritage «répété» de membres :
 - Membres de même nom
 - Même classe héritée plusieurs fois

Exemples d'héritage



Accessibilité

- Accès entre classes (rappel)
 - Par défaut :
 - Java : public mais limité au package (mode implicite « package »)
 - C++ : private par défaut ⇒ accès à l'intérieur de la même classe à l'exclusion des descendantes
 - SmallTalk : méthodes publiques mais limitées au DomainName
- Accès entre classes (lors de l'héritage)
 - Java, SmallTalk : inchangée ⇒ héritage public
 - Héritage à la carte
 - C++ : dépend du mode d'héritage (public, protected, private)
 - Java : On peut changer les propriétés d'accès d'une méthode en la redéfinissant dans une classe héritière, mais ces propriétés ne peuvent **pas** être **plus** restrictives que dans les classe ancêtres.

Accès entre classes

Instances d'une même classe

Accès à tous les membres (publiques, protégés, privés)

Membres d'une classe définie par héritage

Accès à tous les nouveaux membres et aux membres **publics** ou **protégés** qui sont hérités

```
class Mere
{
    public int mpub ;
    protected int mprot ;
    private int mpriv ;
};
class Heritiere extends Mere
{
    public int hpub ;
    protected int hprot ;
    private int hpriv ;
    public void testMere (Mere mere)
    {
        mere.mpub = 11;           // OK
        mere.mprot = 22;         // devrait être interdit mais on est
                                // implicitement dans le même package
        mere.mpriv = 33;         // interdit
    }
}

public void test (Heritiere fille)
{
    fille.hpub = 11;           // OK
    fille.hprot = 22 ;        // OK
    fille.hpriv = 33 ;        // OK
    fille.mpub = 11;          // OK
    fille.mprot = 22 ;        // OK
    fille.mpriv = 33 ;        // interdit
    mpub = 11;                // OK
    mprot = 22 ;              // OK
    mpriv = 33 ;              // interdit
}
```

Héritage à carte



Restriction d'accès uniquement

Héritage global de l'accessibilité :

Tous les membres hérités prennent «au moins» la protection du mode

Exemples :

```
class CompteToto : public Compte { ... } ;  
class CompteBloque : protected Compte { ... } ;
```

Trois modes d'héritage : public, protected, private

Public - L'accessibilité ne change pas :

membres privés : pas d'accès externe

membres protégés : pas d'accès externe, accès dans les héritières

membres publiques : accès externe

Protected :

Tous les membres publiques hérités deviennent protégés,
les autres ne changent pas

Private : (mode par défaut)

Tous les membres hérités deviennent privés

Héritage à la carte - Exemple



Héritage à la carte :

Seuls certains membres sont hérités sans changements

Exemples :

```
class CompteBloque1 : protected Compte  
{  
    public : Compte::solde, Compte::ajout ;  
}; // seul retrait est modifié et devient protégé
```

Mode pour la classe

```
class CompteBloque2 : private Compte  
{  
    public : Compte::solde, Compte::ajout ;  
    protected : double valSolde ;  
}; // seul retrait est modifié et devient privé
```

Mode spécifique pour ces membres

Mode pour la classe

Modes spécifique pour ces membres

- Mot clé « final » : permet de figer l'implémentation d'une classe, d'une méthode, ou d'une variable
- De classe
 - Il ne pourra pas y avoir d'héritières à cette classe
 - exemple : `public final class AFinalClass { ... }`
- De méthode
 - Une telle méthode ne pourra pas être redéfinie dans les classes héritières (protection du code)
 - exemple : `public final static void main (String args[])`
- De variable
 - Une telle variable est une constante (on l'avait déjà vu !)
 - exemple : `final float PI = 3.1415927`

Surcharge lors de l'héritage

- Recouvrement et/ou remplacement
 - C++ : La redéfinition recouvre toutes les surcharges héritées
 - Java : remplacement uniquement des méthode possédant des signatures identiques.

	C++		Java
<pre>class Compte { public : void ouvrir (void) ; void ouvrir (double const) // surcharge OK protected : double solde ; }; class CompteToto : public Compte { public : void ouvrir (double const, double const) ; // redéfinition : remplace les 2 formes héritées ! protected : double decouvert ; };</pre>		<pre>void CompteToto::ouvrir (double const dec, double const sol) { Compte::ouvrir (sld) ; // OK ouvrir (sld) ; // Erreur de compilation : non défini }</pre>	<pre>class Compte { protected double solde ; public void ouvrir () { solde = 0.0; } // surcharge OK public void ouvrir (double montant) { solde = montant; } }; class CompteToto extends Compte { double decouvert ; public void ouvrir (double montant, double decouvert) { ouvrir(); // OK forme héritée ouvrir(0.0); // OK forme héritée super.ouvrir(montant); // OK appel explicite this.decouvert = decouvert; } };</pre>

Héritage des constructeurs

- Constructeurs : Ancêtres → Descendants
- Destructeurs : Descendants → Ancêtres
- Les constructeurs ne sont pas hérités sauf le constructeur par défaut lorsqu'il n'existe aucun constructeur dans la classe héritée.
 - Exemple 1 : On peut malgré tout utiliser les constructeurs des classes ancêtres dans les constructeurs des classes héritières.
 - Java : Dans le corps des constructeurs (1^{ère} instruction)
 - C++ : Dans la liste d'initialisation
 - Exemple 2 : S'il existe un ou des constructeurs valués mais pas de constructeurs par défaut dans les classes ancêtres, les constructeurs valués devront être appelés explicitement dans les classes héritières.

Héritage des constructeurs - Exemple 1

Java	C++
<pre>class A { private int a; public A(int val) { System.out.println("Création A(" + val + ")"); a = val; } } class B extends A { private int b; public B(int val) { super(2); System.out.println("Création B(" + val + ")"); b = val; } } final class testHeritageConstruct { static void main (String args[]) { A a = new A(1); B b = new B(3); } }</pre>	<pre>class A { private : int a; public : A(int val); }; class B : public A { private : int b; public : B(int val); }; A::A(int val) : a(val) { cout << "Création A(" << val << ")" << endl; } B::B(int val) : A(2), b(val) { cout << "Création B(" << val << ")" << endl; } int main(void) { A a(1); B b(3); return 0; }</pre>

Héritage des constructeurs - Exemple 2

- Exemple Java

```
class Compte
{
    protected double solde;
    protected int numero;

    // la classe compte ne possède QU'UN constructeur valué
    public Compte(double solde, int numero)
    {
        this.solde = solde;
        this.numero = numero;
        System.out.println("Compte (valued constructor)");
    }

    public double retrait (double montant)
    {
        return solde -= montant;
    }

    public double ajout (double montant)
    {
        return solde += montant;
    }
}
```

```
class CompteDecouvert extends Compte
{
    double decouvert;

    CompteDecouvert()
    {
        // erreur tentative de déclenchement de Compte()
        decouvert = 0;
        System.out.println("CompteDecouvert (default constructor)");
    }
}

class testComptes
{
    public static void main (String args[])
    {
        Compte c1 = new Compte(); // erreur ce constructeur n'existe pas
        Compte c2 = new Compte(1000.0, 1);

        CompteDecouvert c3 = new CompteDecouvert();
        CompteDecouvert c4 = new CompteDecouvert(1000.0, 3);
        // erreur ce constructeur n'existe pas :
        // => On n'hérite pas de constructeurs !
    }
}
```

Polymorphisme d'héritage

- Une instance héritière peut toujours être vue comme (restreinte à) une instance d'une de ses classes ancêtre
- Exemple Java :

```
class Compte
{
    public Compte()
    { numero = 0; solde = 0.0; }
    public double retrait (double montant)
    { return solde -= montant; }
    public double ajout (double montant)
    { return solde += montant; }
    public double virer (Compte c, double montant)
    {
        retrait(montant);
        c.ajout(montant);
        return montant;
    }
    protected double solde ;
    protected int numero;
};
```

```
class CompteToto extends Compte
{ };
class testComptes
{
    public static void affiche (Compte c)
    { System.out.println("Solde Compte : " + c.virer (c, 0)); }
    public static void main (String args[])
    {
        Compte c;
        CompteToto ct1, ct2 ;
        c = new Compte();
        ct1 = new CompteToto();
        ct2 = new CompteToto();
        c.virer(ct1, 10.0); // ct1 est converti
        ct1.virer(c, 10.0); // ct1 est converti
        ct1.virer (ct2, 10.0); // ct1 et ct2 sont convertis implicitement
        affiche (ct1); // ct1 est converti implicitement
    }
}
```


Lien dynamique

- Problème
 - Choix dynamique de la réalisation des fonctions membres exécutées
≠ surcharge : choix à la compilation, statique
 - Résolution statique : détermination lors de la compilation
 - Résolution dynamique : détermination lors de l'exécution, ce qui revient à toujours exécuter la méthode de l'objet effectivement instancié en mémoire.
- Dans les langages
 - C++ :
 - Par défaut : Résolution statique
 - Résolution dynamique si la méthode concernée est déclarée virtuelle (mot clé **virtual**)
 - Java : Résolution dynamique
 - Smalltalk : Résolution dynamique

Lien dynamique - Exemple

C++

```
#include <iostream.h> // pour le cout

// Classe Compte de base
class Compte
{
protected :
    double solde;
    int numero;
    virtual void affiche(void);
public :
    ...
    void printInfo(char * const);
}
void Compte::printInfo (char * const
    textInfo)
{
    cout << textInfo;
    affiche();
}
void Compte::affiche (void)
{
    cout << "numero : " << numero;
    cout << " solde : " << solde <<
        endl;
}

class CompteDecouvert : public Compte
{
protected :
    double decouvert;
    void affiche(void);
}

void CompteDecouvert::affiche(void)
{
    cout << "numero : " << numero <<
        " solde : " << solde << " decouvert : "
        << decouvert << endl;
}

int main (void)
{
    Compte c1(0);
    CompteDecouvert c2(1) ;
    c1.printInfo("Compte 1: ");
    c2.printInfo("Compte 2: ");
}
```

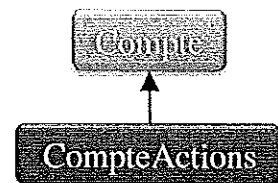
Quelle méthode
« affiche » sera
utilisée dans
« printInfo » ?

Introspection (1/2)

Java

- Capacité que possède une classe à se décrire elle-même.
 - Introspection au travers de la classe « Class » dont les instances représentent une « signature » propre à chaque classe.
 - Utilisation du polymorphisme d'héritage : toute classe descend de la classe Object qui possède la méthode d'introspection.
 - Class getClass() qui permet de déterminer le type (effectif) d'une instance.
 - Les méthodes de la classe « Class » permettent :
 - D'interroger la classe sur son contenu (constructeurs, méthodes, attributs, etc.)
 - Mais aussi sur son ascendance : superclasse, interfaces, et polymorphisme d'héritage.

```
Compte c = new Compte(...);  
CompteActions ca = new CompteActions(...);  
ca.getClass().isInstance(c); → false  
c.getClass().isInstance(ca); → true
```



Introspection (2/2)

Java

- Class literal : **.class** appliqué aux types

```
A a = new A();  
if (a.getClass() == A.class) ⇒ true  
{...}
```

- Opérateur de comparaison de types : **instanceof**

```
System.out.print("a[" +  
                a.getClass().getSimpleName() +  
                "]" );  
if (a instanceof A)  
{  
    System.out.print(" est bien une instance de A");  
}
```

Classes abstraites

Définition : Classes non entièrement implémentées

- Fonctions membres non réalisées : méthodes dites «pures»
- Ne peut pas avoir d'instances
- On peut en parler par référence ou par pointeur
- On peut en hériter
- Ce sont les héritières finales qui fournissent la réalisation des méthodes pures de la classe abstraite

C++ : fonctions/méthodes pures

- Syntaxe : «=0» après le prototype de ces fonctions

Java :

- **Abstract** : Classes et méthodes
- **Interfaces** : Classes entièrement non implémentées,

Smalltalk : méthodes abstraites → Subclass responsibility

Séparer la conception de l'implémentation !

Méthodes pures



- On dit qu'une classe C++ est « abstraite » si elle contient au moins une méthode pure.

Exemple

```
class Compte
{
    public : virtual double virer (Compte &, double const) = 0 ;
    protected : double solde ;
};

class CompteToto : public Compte
{ public : double virer (Compte &, double const) ; } ;
// Implémentation dans la classe fille

double CompteToto::virer (Compte & c, double const v)
{ solde -= 5 ; return Compte::virer (c, v) ; }

void affiche1 (Compte c) // Erreur de compilation : instantiation
{ cout << "Solde : " << c.virer (c, 0) ; }

void affiche2 (Compte & c)
{ cout << "Solde : " << c.virer (c, 0) ; }

int main ()
{
    Compte c ; // Erreur de compilation : instantiation
    CompteToto ct ;
    Compte & ref = ct ; // OK : reference
    return 0 ;
}
```

↑
Méthode
Pure virtuelle

Classes abstraites



- On peut regrouper dans une classe abstraite des membres communs aux sous classes sans connaître encore le détail de l'implémentation.
- Exemple

```
//classe abstraite
abstract class MyFirstAbstractClass
{
    protected int anInstanceVariable;
    // méthode abstraite
    public abstract int
subclassesImplementsMe();
    // une méthode ordinaire
    public void doSomething()
    {anInstanceVariable = 0; }
}

// classe concrète
class AConcreteClass extends
MyFirstAbstractClass
{
    public int subclassesImplementsMe()
    {return anInstanceVariable++;}
}

class testAbstractClass
{
    protected static int aNumber;

    // ok car il s'agit d'une référence
    public static void
    uneMethode(MyFirstAbstractClass ac)
    {aNumber = ac.subclassesImplementsMe(); }

    public static void main (String args[])
    {
        MyFirstAbstractClass a =
            new MyFirstAbstractClass();// illégal
        AConcreteClass b = new AConcreteClass();// Ok
        MyFirstAbstractClass ab = b;// Ok
        uneMethode(b);
    }
}
```

Classes interfaces



- Interfaces : Classes non implémentées, ne contiennent que des :
 - Méthodes abstraites : *public abstract*
 - Constantes de classe : *public static final*
 - Les Interfaces représentent des « comportements »
- On peut considérer la hiérarchie des interfaces comme // à la hiérarchie des classes.
- Une classe concrète « implémente » une interface

- Exemple :

```
public class Neko extends java.applet.Applet
    implements Runnable {...}
```

- Neko hérite de Applet c'est donc une sorte d'applet
- Neko implémente aussi Runnable, elle **doit** donc implémenter un comportement qui lui permet de s'exécuter comme un programme Java ordinaire.
- une classe Java peut « implémenter » plusieurs interfaces
 - Héritage multiple de « comportements »

- Exemple

```

interface Figure
{
    public abstract void dessine();
    public abstract void echelle(double
        facteur);
    // public abstract est déjà le mode par défaut
    // pour les méthodes dans une interface
}

class Point
{ ... }

abstract class FormeGeometrique
{
    protected static int nbPoints;
    protected Point[] points;

    public abstract double aire();
}

class Triangle extends FormeGeometrique
implements Figure
{
    // constructeur : propre à Triangle
    Triangle()
    {
        nbPoints = 3;
        for (int i=1; i < nbPoints; i++)
            {points[i] = new Point(); }
    }
    // dessin : comportement de figure
    public void dessine() {...}
    // echelle : comportement de figure
    public void echelle(double facteur)
    { ... }

    //aire : implémentation méthode abstraite
    forme...
    public double aire()
    { ... }
}
    
```

L'héritage multiple (1/2)

- Déclaration : Liste de plusieurs classes mères
- Construction : Liste des différents constructeurs à appeler pour chaque classe
- Tous les destructeurs sont appelés
- Problèmes de conflits :
 - Héritage de membres de même nom

Préciser lequel on veut en ajoutant le préfixe de la classe ancêtre
NomClasse::...

Exemple :

```

class A { public : int i ; };
class B { public : int i ; };
class C : public A, public B { public : int i ; void f () ; };
void C::f () { i=13 ; A::i=1 ; B::i=7 ; }
    
```

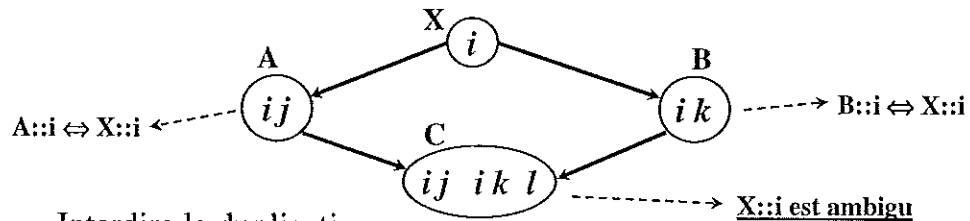
L'héritage multiple (2/2)

2. Héritage répété de membres

Préciser lors de l'accès *NomClasse::*

Exemple :

```
class X { public : int i ; } ;
class A : public X { public : int j ; } ;
class B : public X { public : int k ; } ;
class C : public A, public B { public : int l ; } ;
```

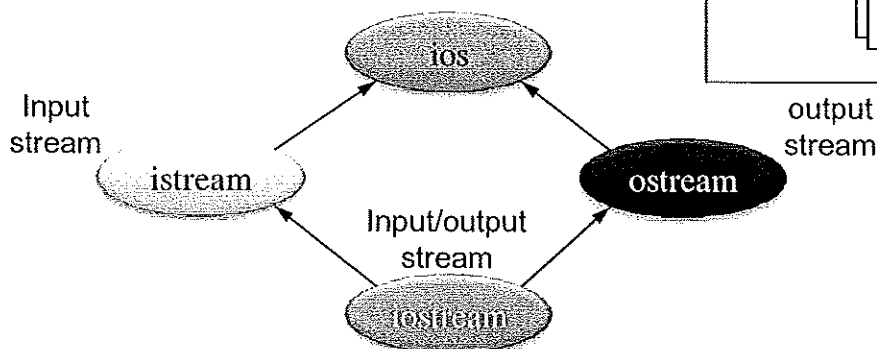
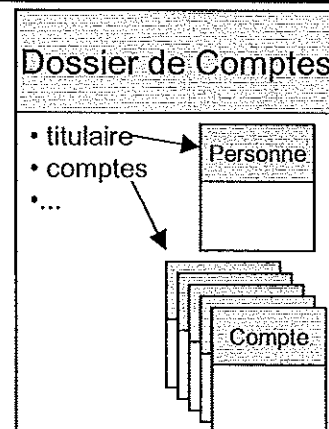


Interdire la duplication

```
class C : virtual public A, virtual public B {
    public : int l ; } ;
```

2.9. Composition

- Composition : Est composé de
 - Exemple
 - Dossier de comptes
- Composition par héritage
 - Nécessite l'héritage multiple
 - Exemple : Les flux d'entrée sortie en C++



2.10. Généricité

- Définition
 - Capacité d'une classe à manipuler des entités non-(encore)-typées
 - Exemples : Tableau de ..., Listes de ...
- Implémentation dans les différents langages
 - Smalltalk : implicite grâce au typage faible du langage.
 - Java :
 - En quelque sorte (implémentation sur des " Object ", Arbre d'héritage & utilisation du polymorphisme d'héritage)
 - Exemples : Array et Vector de « Object » etc ...
 - Ou bien utilisation d'un paramètre de type avec « Java Generics »
 - Exemples : Array<T>, Vector<T> etc...
 - C++ : patrons de classes : mot clé « template »
 - Mécanismes :
 - Instanciation / proto-instanciation
 - Exemple : instanciation d'un tableau d'entiers à partir d'un tableau générique :
Tableau<int> tab; instanciation de la classe générique Tableau<T> avec des int

Généricité - Patrons de classe



- Mot clé : **template**
- Patrons de **fonctions** et de **classes**

1. Patrons de fonctions :

Les types doivent apparaître au minimum dans les types des arguments

```
template <class C1, class C2> TypeRetour NomFonction (C1 arg1, C2 arg2, ...)  
{ ... }
```

2. Patrons de classes :

```
template <class C1, class C2> class C  
{  
    C1 m1 ;  
    void f (C2) ;  
    ...  
}; ...  
template <class C1, class C2> void C::f(C2 arg) { ... }
```

3. Instanciation du patron avec des types connus :

```
C <int, double> monInstaneDeC; ←  
C <Compte, float> autreInstaneDeC; ← Deux types différents
```

Attention, pas de conversion de type implicite, sauf entre classes héritières et classes ancêtres

→ Le lien dynamique d'héritage est préservé

Patrons de classe – Exemple (1/6)



```

template <class UnType> // ou bien template <typename UnType>
class Tableau
{
public :
    // -- CONSTRUCTEURS -----
    // constructeur par défaut
    Tableau (void) ;
    // constructeur de copie
    Tableau (Tableau const &) ;
    // constructeur valué
    Tableau (int const, UnType const []) ;
    // destructeur
    ~Tableau (void) ;

    // -- ACCESSEURS -----
    int LireTaille (void) const ;
    UnType * LireTab (void) const ;
    UnType getElement (int const) const ;
    void setElement(int const, UnType const &) ;

    // -- OPÉRATEURS -----
    // opérateur d'affectation
    Tableau & operator = (Tableau const &) ;
    // opérateur crochets
    UnType operator[] (unsigned int const) const ;
    // opérateur de sortie standard défini en interne
    friend ostream & operator << >> (ostream &, Tableau<UnType> const
);
protected :
    int Taille ;
    UnType * Tab ;

    void MemoirePleine (void) const ;
    void Debordement (void) const ;
};
// opérateur de sortie standard défini en externe
template <class UnType> ostream & operator << (ostream &, Tableau<UnType> const
&);
    
```

Patrons de classe – Exemple (2/6)



```

// constructeur par défaut
template <class UnType> Tableau<UnType>::Tableau (void) : Taille (0), Tab (NULL)
{
}

// constructeur de copie
template <class UnType>
Tableau<UnType>::Tableau (Tableau<UnType> const & t) :
    Taille (t.LireTaille ()), Tab (NULL)
{
    set_new_handler(&MemoirePleine);
    Tab = new UnType [Taille];
    for (int i=0; i<Taille; i++)
    {
        Tab[i]=t.Tab[i];
    }
}

// constructeur de copie
template <class UnType> Tableau<UnType>::Tableau (int const n, UnType const t [])
: Taille (n), Tab (NULL)
{
    set_new_handler(&MemoirePleine);
    Tab = new UnType [Taille];
    for (int i=0; i<Taille; i++)
    {
        Tab[i]=t[i];
    }
}

// Destructeur
template <class UnType> Tableau<UnType>::~~Tableau (void)
{
    if (Tab) delete [] Tab;
}
    
```


Patrons de classe – Exemple (3/6)

```
// Lecture des attributs -----
template <class UnType> int Tableau<UnType>::LireTaille (void) const
{
    return Taille ;
}

template <class UnType> UnType * Tableau<UnType>::LireTab (void) const
{
    return Tab ;
}

template <class UnType> UnType Tableau<UnType>::getElement (int const n) const
{
    if (n > Taille-1)
    {
        Debordement () ;
    }
    return Tab[n] ;
}

// Ecriture des attributs -----
template <class UnType> void Tableau<UnType>::setElement(int const n, UnType const & e1)
{
    if (n > Taille-1)
    {
        Debordement () ;
    }

    Tab[n] = e1 ;
}
```

Patrons de classe – Exemple (4/6)

```
// Operateur d'affectation -----
template <class UnType> Tableau<UnType> & Tableau<UnType>::operator =
    (Tableau<UnType> const & t)
{
    Taille = t.Taille;
    delete [] Tab;
    set_new_handler(&MemoirePleine);
    Tab = new UnType [Taille];
    for (int i=0; i<Taille; i++)
    {
        Tab[i]=t.Tab[i];
    }
    return *this;
}

template <class UnType> UnType Tableau<UnType>::operator[] (unsigned int const i)
    const
{
    return getElement(i);
}

// Methodes internes a la classe -----
template <class UnType> void Tableau<UnType>::MemoirePleine() const
{
    cerr << "Probleme d'allocation dans la classe Tableau !" << endl;
    cerr << "Programme interrompu !" << endl;
    exit(1);
}

template <class UnType> void Tableau<UnType>::Debordement() const
{
    cerr << "Probleme de debordement dans la classe Tableau !" << endl;
    cerr << "Programme interrompu !" << endl;
    exit(1);
}
```

Patrons de classe – Exemple (5/6)

```
// Operateur de sortie vers un flux -----
template <class UnType>
ostream & operator << (ostream & sortie, const Tableau<UnType> & tab)
{
    for (int i=0; i<tab.LireTaille(); i++)
    {
        sortie << tab.getElement(i) << ' ' ;
    }
    sortie << endl ;
    return sortie;
}

```

```
// =====
// TableauInt.cc : Proto-instanciation d'un Tableau d'entiers
// à partir de la classe générique Tableau
// =====

#include "Tableau.h"
#include "Tableau.cpp"

// instanciation du template avec des entiers
template class Tableau<int>;

// instanciation de l'opérateur de sortie avec des entiers
template ostream & operator << (ostream &, const Tableau<int> &);
// template istream & operator >> (istream &, Tableau<int> &);

```

2008

Langages Orientés Objet - David Roussel

99

Patrons de classe – Exemple (6/6)

```
#include "Tableau.h" // pour la classe Tableau
#include <iostream> // pour cout et endl;
using namespace std;

// ===== programme principal =====

int main (void)
{
    int t1 [] = {1, 2, 3, 4, 5}, t2 [] = {6, 7, 8, 9} ;
    float f1, f2 ;

    Tableau<int> tab1 (5, t1);
    Tableau<int> tab2 (4, t2);

    cout << "Tab1 : " << tab1 << endl ;
    cout << "Tab2 : " << tab2 << endl ;
    cout << endl ;
    tab1 = tab2 ;
    cout << "Tab1 apres tab1 = tab2 : " << tab1 << endl ;

    // acces par opérateur []
    cout << "Tab1[1] = " << tab1[1] << endl;
    cout << endl ;

    // opération impossible (pour l'instant)
    tab1[2] = 45; // erreur !!! l'opérateur [] n'a pas été définie pour cela

    cout << "4 ieme element de tab2 : " << tab2.getElement (3) << endl ;
    cout << "5 ieme element de tab2 : " << tab2.getElement (4) << endl ;
    cout << endl ;
}

```

2008

Langages Orientés Objet - David Roussel

100

- Problème : L'utilisation du polymorphisme d'héritage dans les collections (comme Vector) ne garantit pas l'unicité des types dans une collection :

```
Vector soupeAuxChoux = new Vector();
soupeAuxChoux.addElement(new Chou());
soupeAuxChoux.addElement(new Carotte()); // PB ceci n'est pas un chou
soupeAuxChoux.addElement(new Chaise()); // Hum, ...
```

- Solution : JDK 1.5 a introduit la notion de classes génériques (*generics*) en rajoutant des **paramètres de type** aux classes dites génériques.

```
Vector<Chou> soupeAuChou = new Vector<Chou>();
soupeAuChou.addElement(new Chou());
soupeAuChou.addElement(new Carotte()); // erreur de compilation
```

- Apport principal : Meilleur contrôle du typage dans la généricité

Polymorphisme d'héritage vs Java Generics

- Exemple sur une collection
 - Généricité par Polymorphisme d'héritage

```
public class MaCollection {
    ...
    public boolean add(Object element){...}
    public Iterator iterator(){...}
}
```

Collection de « Object »

```
MaCollection col = new MaCollection();
col.add(new Integer());
Integer x = (Integer)col.iterator().next();
```

Cast explicite

- Généricité avec Java Generics

```
public class MaCollection<E>{
    ...
    public boolean add(E element){...}
    public Iterator<E> iterator(){...}
}
```

Paramètre de type

Variable de type

Collection de « E »

Collection de « Integer »

```
MaCollection<Integer> col = new MaCollection<Integer>();
col.add(new Integer());
Integer x = col.iterator().next();
```

Généricité : Sous typage

Java

- Sous typage

- Exemple

```
List<String> ls = new ArrayList<String>();  
List<Object> lo = ls;
```

- String hérite de Object :

- Une liste de String est elle une sorte de liste de Object ?

```
lo.add(new Object());  
String s = ls.get(0);
```

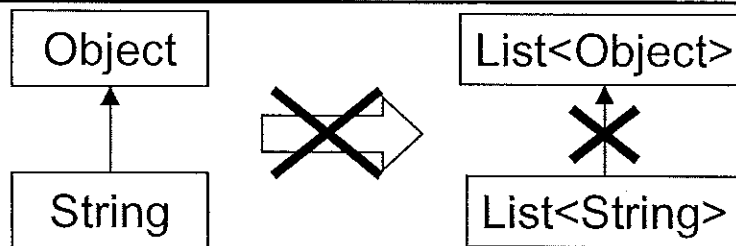
Tentative d'affectation
d'un « Object »
dans un « String »

illégal

Erreur de
compilation

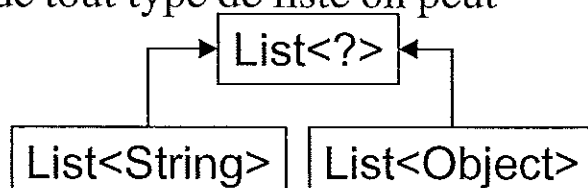
Généricité : Wildcards

Java



- Sous typage : une liste de String n'est PAS une sorte de liste d'Object.
- On a néanmoins besoin de définir la notion de liste de « n'importe quoi » pour assurer l'ancienne forme de généricité.
- Pour désigner le supertype de tout type de liste on peut utiliser le « wildcard » : ?

– List<?>



Généricité : utilisation de ?

- Exemple : écrire une méthode suffisamment générique pour afficher les éléments de listes de tout type

- Ancienne version

```
void printList(List l) {  
    for (Iterator i = l.iterator();  
         i.hasNext();) {  
        System.out.println(i.next());  
    }  
}
```

- Nouvelle version

```
void printList(List <Object> <?> l) {  
    for (Object e : l) {  
        System.out.println(e);  
    }  
}
```

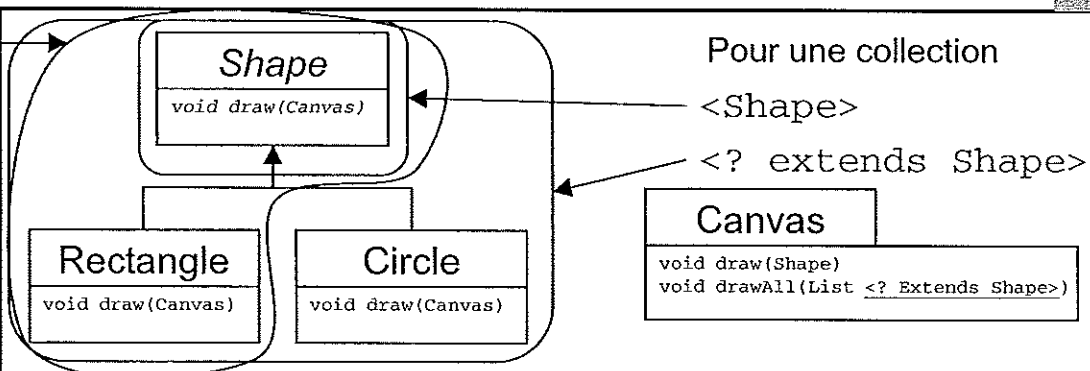
???
Sûr car \forall le type des éléments de la liste, ils dérivent de la classe Object: Arbre d'héritage.

- Ecriture dans une Liste Générique

```
List<?> l = new ArrayList<String>();  
l.add(new Object());
```

Erreur de Compilation :
Le compilateur ne peut pas inférer le type de <?>

Généricité : Wildcards bornés



- Dessiner une liste de « Shape » peut se faire avec <Shape> mais dessiner une liste de Circle ou de Rectangle nécessite le type générique <? extends Shape>
 - « Shape » est ici la borne supérieure du wildcard « ? »
 - ☠ La non inférence du type de <? extends Shape> par le compilateur subsiste \Rightarrow pas d'accès en écriture
- Il existe aussi des bornes inférieures : <? super Rectangle>

Tableaux génériques

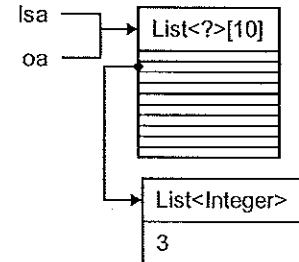
Java

- Le type des éléments d'un tableau effectif ne doit pas être une variable de type ou un paramètre de type, sauf un wildcard non borné.

```
T[] t = new T[10] // non autorisé si T est un paramètre de
                  // type ou une variable de type
new List<String>[10] // Non autorisé
new List<?>[10] // Autorisé
```

- Toutefois, on peut déclarer des tableaux dont le type est une variable de type ou un type paramétrisé.

```
List<String>[] lsa = (List<String>[]) new List<?>[10];
// Type safety warning
Object[] oa = lsa;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[1] = li; // correct
String s = lsa[1].get(0); // Run time error
// Mais on a été prévenu !
```



Méthodes génériques

Java

- Classe générique : Classe possédant un ou plusieurs paramètres de type : p. ex. Map <K,V>
- Méthode Générique : Méthode possédant un ou plusieurs paramètres de type.

- Exemple : Méthode permettant de copier les éléments d'un tableau dans une collection.

- Mauvais exemple :

```
static void fromArrayToCollection(Object[] array, Collection<?> col) {
    for (Object o : array) {
        c.add(o);
    }
}
```

Erreur de Compilation :
Le compilateur ne peut pas
inférer le type de <?>

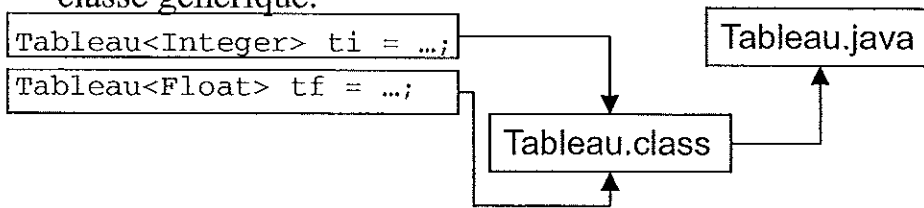
- Méthode Générique :

```
static <T> void fromArrayToCollection(T[] array, Collection<T> col) {
    for (T o : array) {
        c.add(o);
    }
}
```

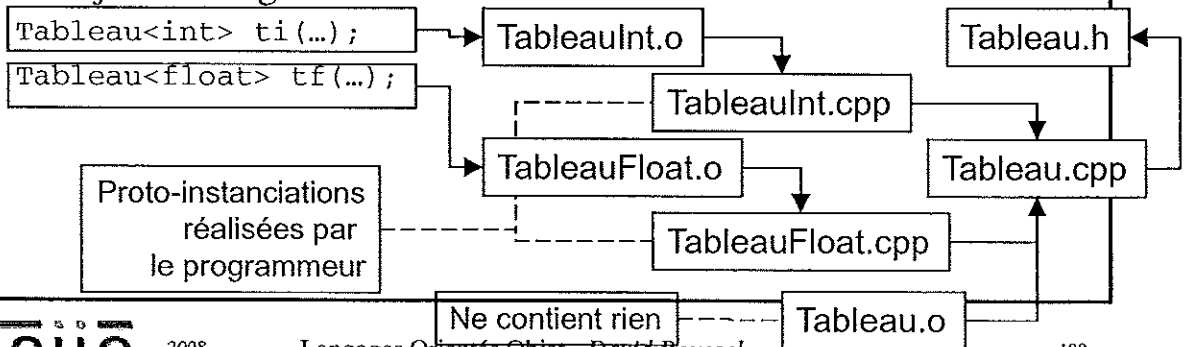
Paramètre de type

Généricité : Java vs C++

- Différence entre Java Generics et les Templates C++
 - Java : une seule classe est générée lors de la compilation d'une classe générique.



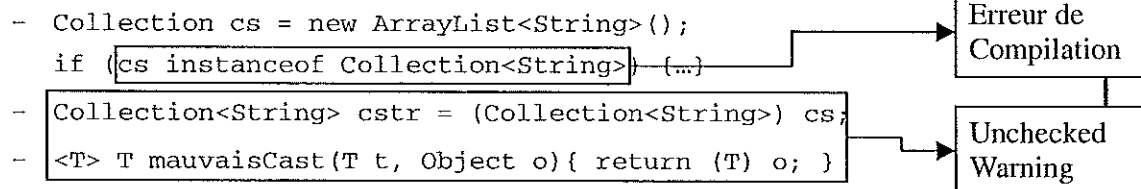
- C++ : Chaque proto-instanciation doit générer son propre fichier objet afin de générer du code exécutable.



Java Generics : introspection



- Une classe générique est partagée par toute ses instances
- Cela n'a donc pas de sens d'interroger une classe générique sur la nature de ses instances :



- Les variables de type n'existent pas lors de l'exécution, elles doivent être résolues à la compilation.
 - On ne peut donc pas les « introspecter » ou les caster.

2.11. Robustesse

- Définition
 - Garantir le bon fonctionnement quel que soit l'état du programme
- Notion de contrat
 - Préconditions : partie à remplir par le client avant l'appel d'un traitement.
 - Postconditions (axiomes) : partie à remplir par le fournisseur à la fin d'un traitement.
 - ⇒ Uniquement défini dans le langage Eiffel
- Gestions des erreurs
 - Exceptions : anomalies susceptibles de se produire dans un programme
 - Lorsqu'un problème survient, une exception est levée : « thrown »
 - On traite le problème en capturant l'exception : « catch »
 - Problème : Qui lève, et qui capture les exceptions ???

Exceptions

- Caractéristiques
 - Les exceptions sont des objets
- Soulèvement d'une exception :
 - clause « throw »
 - les méthodes susceptibles de lever une exception doivent le signaler : clause « throws » suivant l'entête de la méthode.
- Capture d'une exception : bloc « try ... catch »

```
try { bloc d'instructions pouvant soulever une ou plusieurs exceptions }
catch type d'exception { bloc de traitement de l'exception }
catch autre type d'exception { bloc de traitement de l'exception }
etc ...
```
- Règles d'utilisation
 - Une clause throw implique la présence d'une clause catch : une exception **doit** être traitée
 - Ne pas utiliser les exceptions sur des erreurs prévisibles !

Exceptions - Exemple

Java

```
class DivideByZeroException extends
Exception
{
    public DivideByZeroException () {}
    public DivideByZeroException (String
msg)
    {
        super(msg);
    }
}

class Fraction {
    protected double numerator;
    protected double denominator;

    Fraction()
    { numerator = 0.0; denominator = 0.0;}

    Fraction(double numerator, double
denominator)
    {
        this.numerator = numerator;
        this.denominator = denominator;
    }

    double divide() throws
DivideByZeroException
    {
        if (denominator == 0.0)
            throw new DivideByZeroException("It
happend, I can't believe it!");
        else
            return numerator / denominator;
    }
}
```

```
class testDivision
{
    public static double
tryToDivide(Fraction frac)
    {
        double result = Double.MAX_VALUE;
        try
        {
            result = frac.divide();
        }
        catch (DivideByZeroException dze)
        {
            System.out.println("Sorry: " +
dze.toString());
        }
        return result;
    }

    public static void main (String args[])
    {
        Fraction frac1 = new Fraction();
        Fraction frac2 = new Fraction(1.0,
2.0);

        // essaye de calculer les fraction
        System.out.println("Dividing frac1");
        tryToDivide(frac1);
        System.out.println("Dividing frac2");
        tryToDivide(frac2);
    }
}
```

ensiie

2008

Langages Orientés Objet - David Roussel

113

Exceptions - Exemple

C++

```
template <class T> class Matrix
{
    public:
    ...
    Matrix<T> & operator * (const Matrix<T> & m) const throw (MatrixException);
    ...
};
```

Matrix.h

```
template <class T>
Matrix<T> & Matrix<T>::operator * (const Matrix<T> & m) const throw (MatrixException)
{
    if (!compatibleSize(m)) {
        throw MatrixException(MatrixException::NOT_MULTIPLICABLE);
    }
    else {
        Matrix<T> * newMat = new Matrix<T>(nbLines,m.nbCols);

        // building new matrix values
        for (unsigned int i=0; i < newMat->nbLines; i++) {
            for (unsigned int j=0; j < newMat->nbCols; j++) {
                newMat->array[i][j] = nullElement();

                for (unsigned int k=0; k < nbCols; k++) {
                    newMat->array[i][j] += array[i][k] * m.array[k][j];
                }
            }
        }
        return *newMat;
    }
}
```

```
Matrix<double> m33(3,3);
m33.value(33);

Matrix<double> m45(4,5);
m45.value(45);

try {
    cout << "m33 * m45 = "
    << m33 * m45 << endl;
}
catch (MatrixException e) {
    cerr << e;
    exit(1);
}
```

test.cpp

Matrix.cpp

ensiie

2008

Langages Orientés Objet - David Roussel

114

3. Remerciements

- Merci à François Terrier et Annelies Braffort pour leur cours de C++ dont on peut retrouver ici les éléments.
- Merci à Ivan Augé pour la première version de ce cours
- Merci à Régine Laleau pour ses conseils en Java
- Merci à Xavier Briffault et Brigitte Grau pour leurs conseils en Smalltalk