



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

Langages Orientés Objet

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 [Classes internes & imbriquées](#)
 - 2.8 [Polymorphisme](#)
 - 2.9 [L'Héritage](#)
 - 2.10 [L'Introspection](#)
 - 2.11 [Classes abstraites](#)
 - 2.12 [Composition](#)
 - 2.13 [Généricité](#)
 - 2.14 [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*
 - Nécessité d'évolution
 - amortissement nécessaire
 - réutilisation et/ou adaptation d'un tel objet
 - Complexité
 - de spécification
 - d'implémentation
- 
- 100.000 lignes
⇒ 4 / 20 h/a
⇒ 120 / 600 K€
- *: efficaces

1.1 Le Logiciel

- Qualité logicielle

- Aspect externe

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

Utilisateur

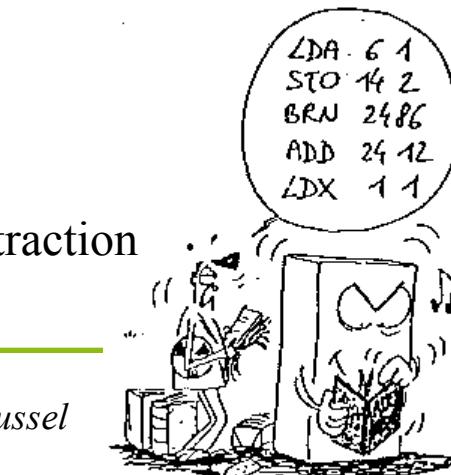
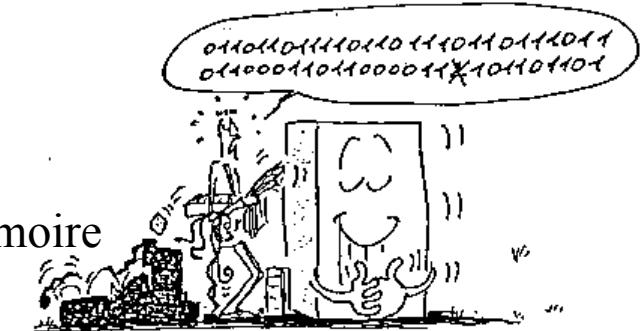
- Aspect interne

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

Programmeur

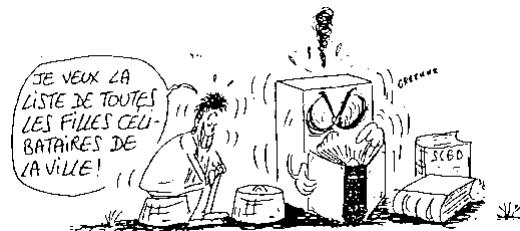
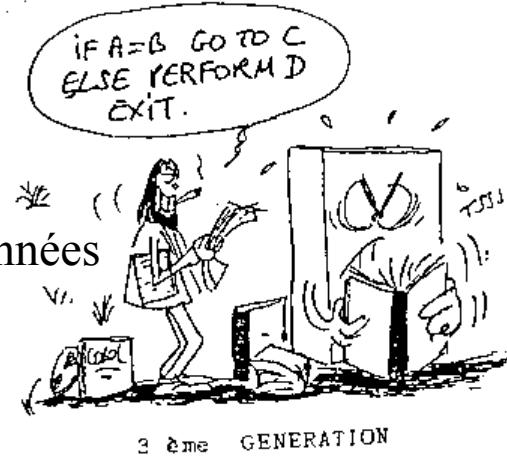
1.2. Historique de la conception (1/2)

- 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 ⇒ réécriture
 - 2^{ème} génération (1960)
 - Assembleur
 - Structure de données évoluée et globale
 - Sous programmes comme éléments d'abstraction
 - Modification ⇒ réécriture



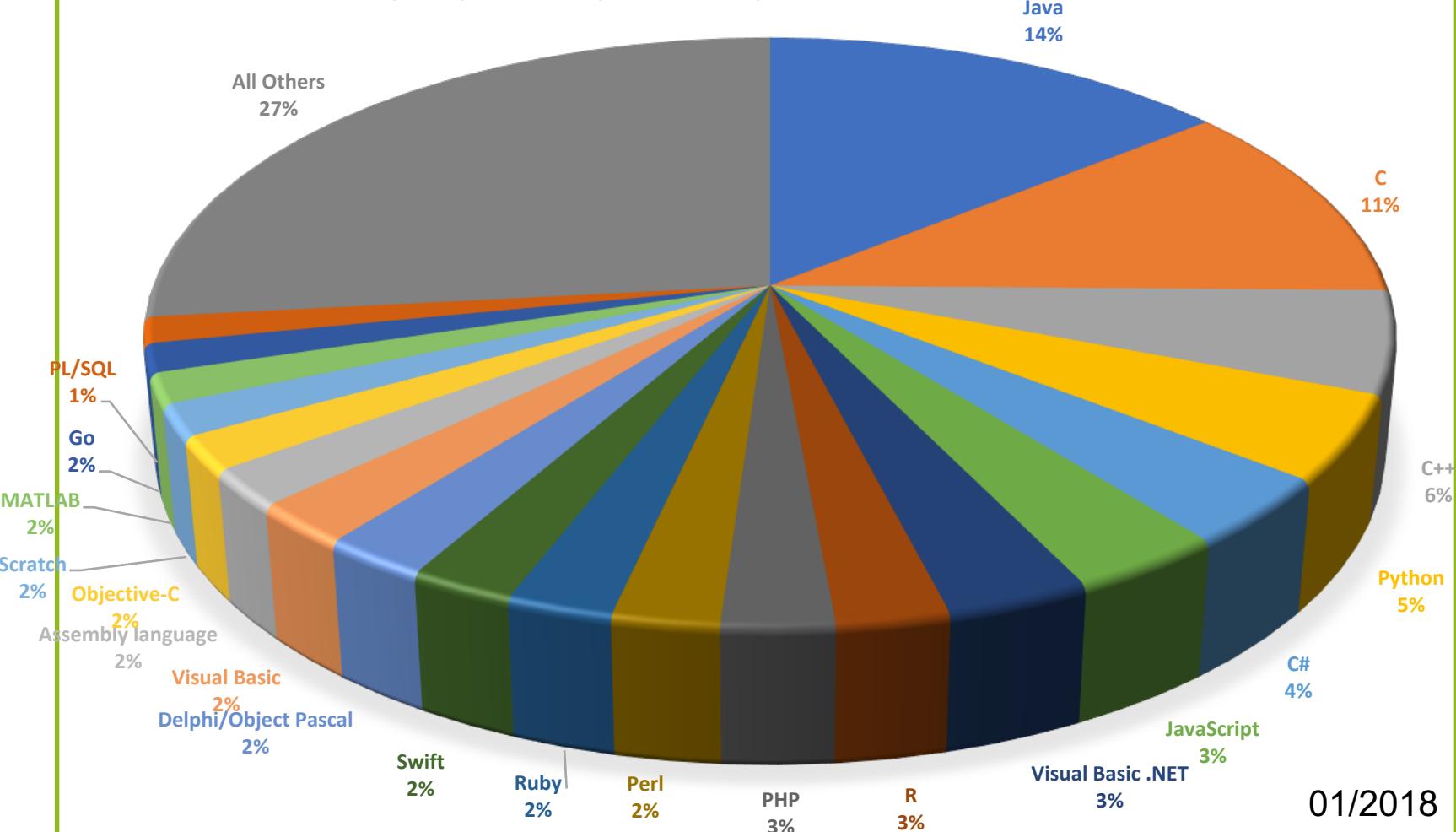
1.2. Historique de la conception (2/2)

- 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 ⇒ 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 ...



TIOBE Index

Requêtes +"<language> programming" dans les moteurs de recherche

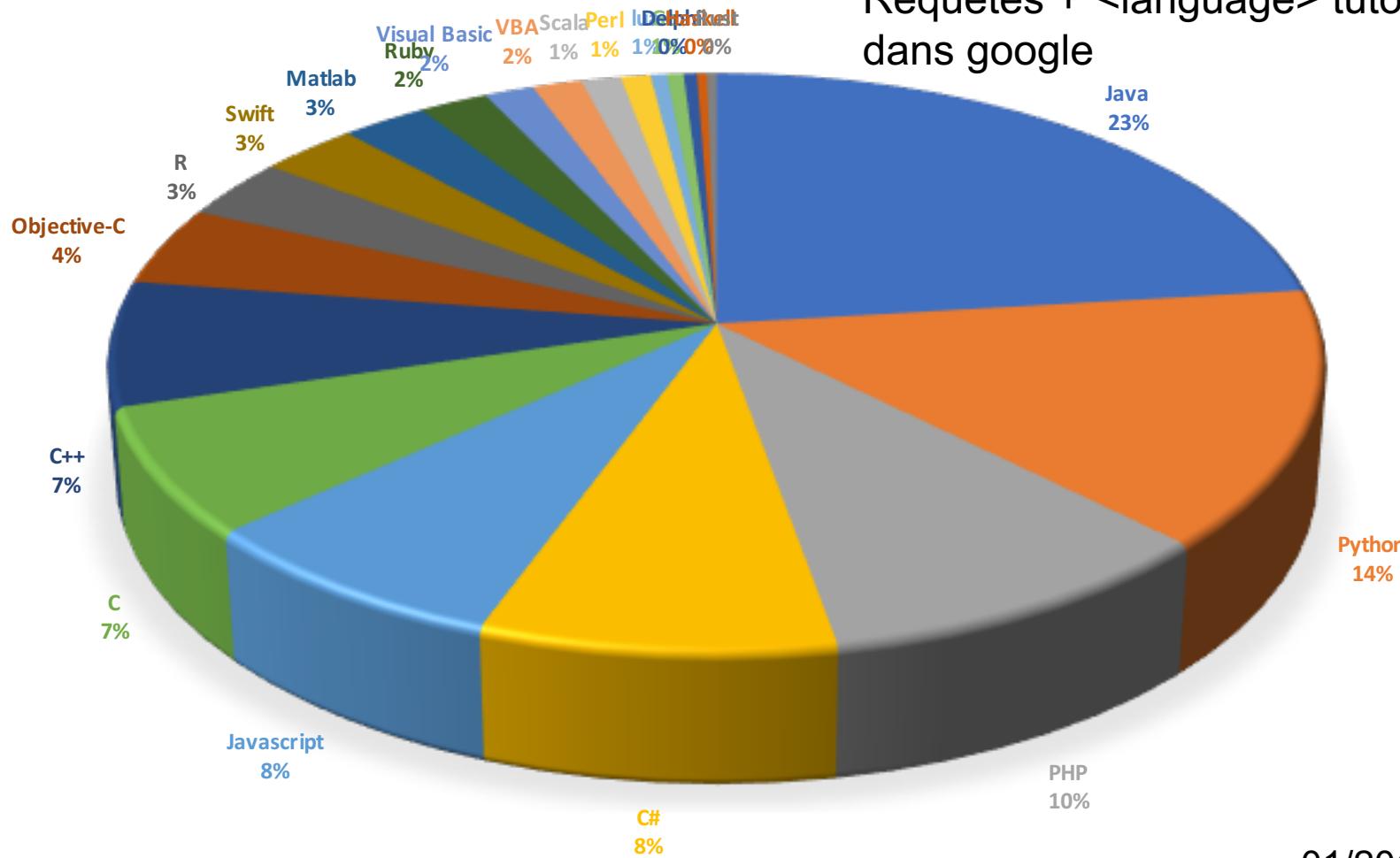


<http://www.tiobe.com/tiobe-index/>

Langages Orientés Objet - David Roussel

PopularitY of Programming Language (PYPL)

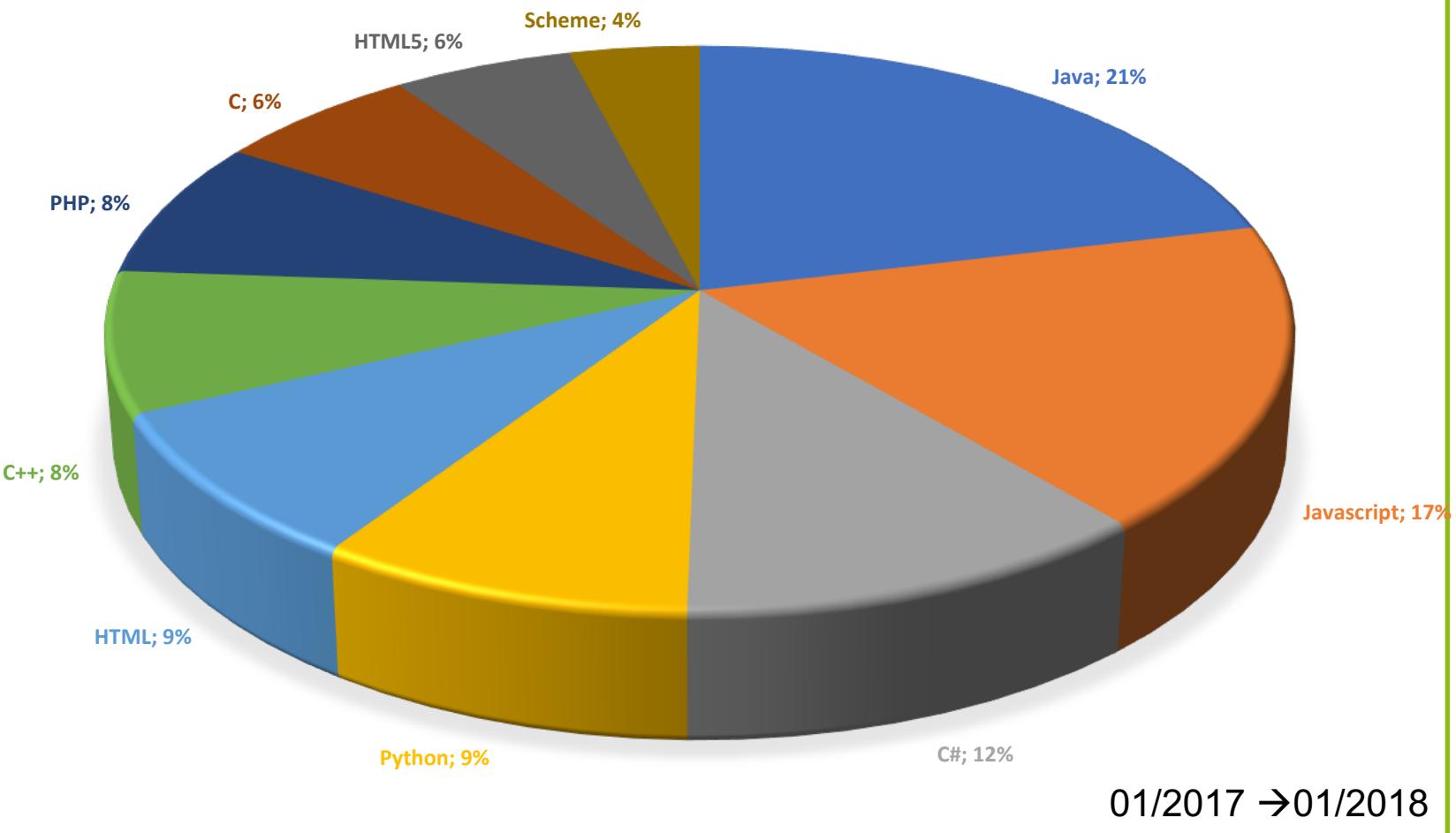
Requêtes +"<language> tutorial"
dans google



01/2017

Trendy Skills

Extracting Skills that employers seek in the IT industry



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 ⊂ 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++*, C#, SmallTalk

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

* : voir la fin du cours (p 111)

2.1. Caractéristiques des langages-exemples

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

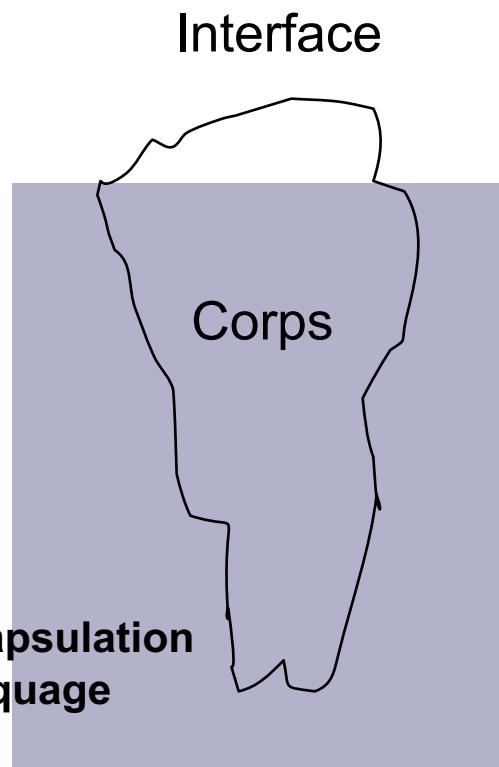


Modification

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.
 - Typage dynamique (d'après les expressions).
 - Tout est objet



Modification

Exemple

Objet : «*Compte bancaire*»

Données membres - **Attributs** (variables d’instance) :
numéro et *solde*

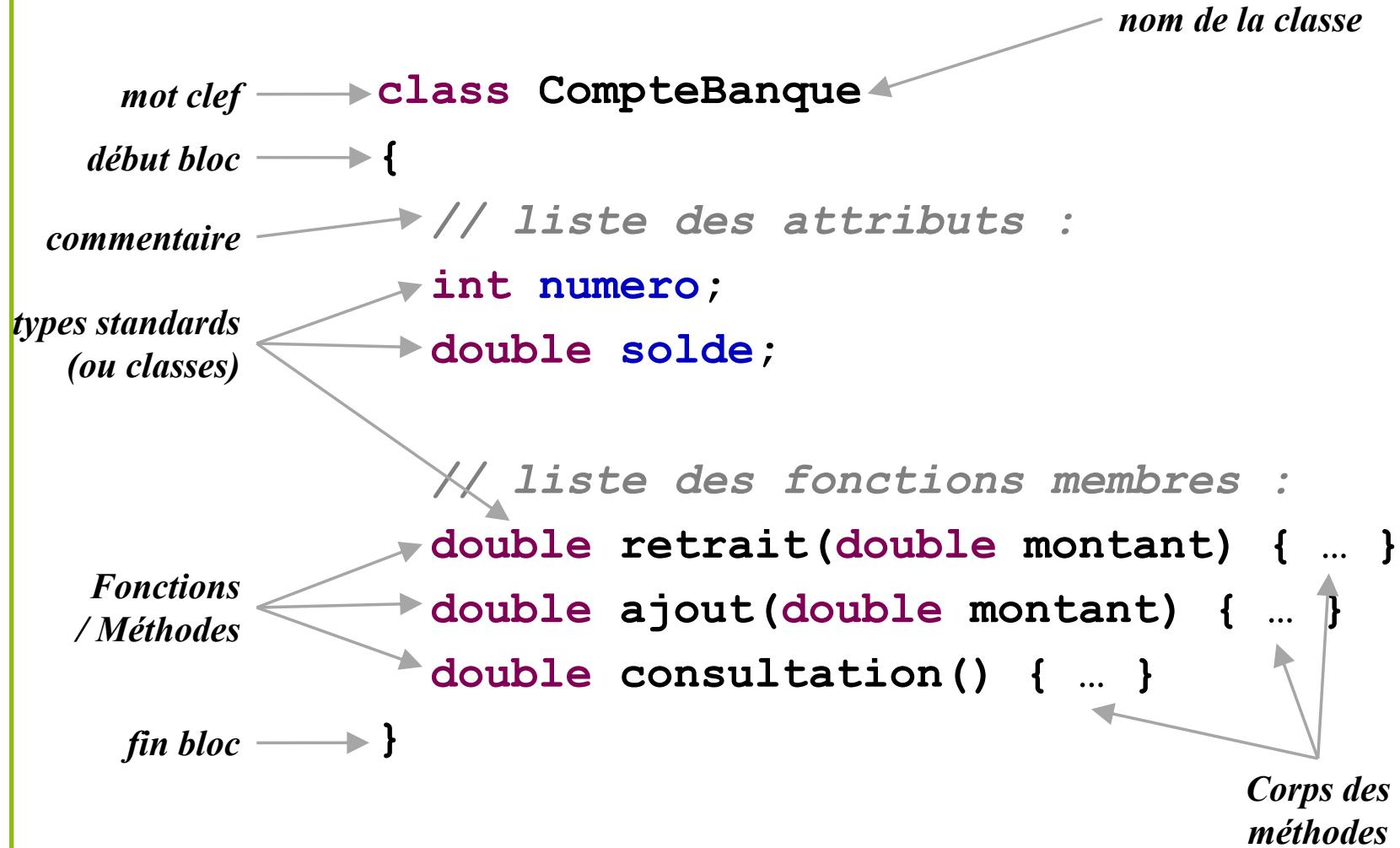
Fonctions membres - **Méthodes** (traitements) :
retrait, *ajout* et *consultation*

Définition de la classe :

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

Exemple - Déclaration

Java



Exemple - *implémentation*

Java

The diagram shows a Java code snippet with annotations pointing to specific parts:

- valeur de retour de la fonction**: Points to the opening brace of the first method definition.
- nom de la fonction**: Points to the name of the first method, `retrait`.
- type argument**: Points to the return type of the first method, `double`.
- nom argument**: Points to the parameter of the first method, `montant`.
- Corps de la méthode**: A bracket on the right side groups the body of the first method (`{ soldé = soldé - montant; return soldé;`) and the bodies of the other two methods (`ajout` and `consultation`).

```
// ...
double retrait (double montant)
{
    soldé = soldé - montant;
    return soldé;
}

double ajout (double montant)
{
    return soldé += montant;
}

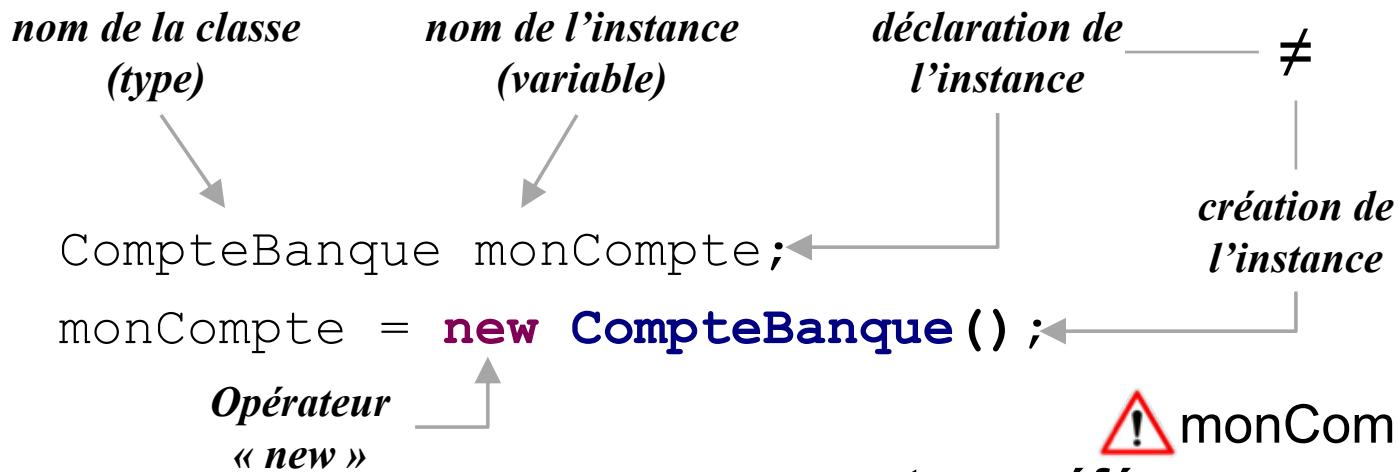
// équivalent à : soldé = soldé + val;
return soldé;

double consultation ()
{
    return soldé;
}
```

Exemple - Instanciation

Java

Déclaration & création d'une instance :



Autres cas :

- Constantes : «final»

```
final Chaine VIDE = new Chaine("");
```

Déclaration/
Création de
tableaux

- Tableaux : «[]»

```
Deux  
formes de  
déclaration  
Point vecteur[] = new Point[10];  
Point[][] matrice = new Point[10][100];  
for (int i=0;i<5;i++) {vecteur[i]= new Point();}
```

Création des
cases du
tableau

Exemple - Accès aux membres

Java

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

Opérateur :

« . » (point)

« . » (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.sold  
CompteBanque.retrait (23)
```

Faux ! (CompteBanque est un type, pas une variable)

Accès à soi même

Java

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

```
void init(double solde, int numer0)  
{  
    this.solde = solde;  
    this.numer0 = numer0;  
}
```

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)
- 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 !

Encapsulation

* : 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, C++, 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 Java :

```
...
CompteBanque monCompte = new CompteBanque(...);
monCompte.retrait(100); // autorisé si retrait est déclaré public
```

- **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 monCompte = new CompteBanque(...);
monCompte.solde = 100; // refusé si solde est déclaré 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 \forall leur mode d'accès.
- **Modes par défaut**
 - Lorsque l'on ne spécifie aucun mode d'accès
 - **Java** : le mode implicite est le mode « package » équivalent au mode « public », mais limité aux classes d'un même package
 - **C++ & C#** : le mode par défaut est le mode « private »
 - **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» : publics
 - méthodes d'accès aux attributs (ex : consultation)
 - opérations de la classe (ex : retrait, ajout)

} Encapsulation forte

→ Nouvelle définition de la classe CompteBanque :

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

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

Java

Accès dans les Packages

Java

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

```
package monpackage;
public class Visible
{
    protected Cachee cachee;
    public Visible()
    {
        cachee = new Cachee();
    }
}
```

monpackage/Visible.java

```
package monpackage;
class Cachee
{
    ...
    public Cachee() {...}
}
```

monpackage/Cachee.java

```
import monpackage.*;

public class TestPackageAccess
{
    public static void main(String[] args)
    {
        Visible vis = new Visible();
        Cachee cas = new Cachee(); //erreur
        // interdit car "Cachee" n'est pas
        // accessible en dehors du package
    }
}
```

TestPackageAccess.java

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.
 - Lors de la déclaration des attributs

→ Nouvelle définition de la classe CompteBanque :

```
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 = 0;
    protected double solde = 0.0;
}
```

Java

Possible
mais pas
recommandé

Constructeurs

2.5. Constructeurs / Destructeur

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 (Ctors)
 - Destructeur (Dtor)
 - Java
 - Constructeurs
 - Pas de destructeur : Garbage collecting pour la destruction & méthode de terminaison (`finalize`) déclenchée par le Garbage Collector.
 - C#
 - Constructeurs
 - Le Garbage Collector déclenche les destructeurs qui eux même déclenchent implicitement les méthodes `Finalize`.
 - SmallTalk
 - Constructeur : Méthode `new` (ou tout autre méthode)
 - Garbage collecting pour la destruction

Constructeurs (1/3)

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
public 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

CompteBanque.java

Constructeurs (2/3)

- Vocabulaire
 - Constructeur **par défaut** : constructeur sans arguments initialisant les attributs avec des valeurs par défaut.
 - On verra quelques subtilités supplémentaires à ce sujet.
 - Constructeur **valué** : constructeur possédant un ou plusieurs arguments destinés à initialiser tout ou partie des attributs.
 - Constructeur **de copie** : constructeur possédant un seul argument du type même de la classe et permettant de copier tout ou partie des attributs d'une instance dans une autre.

Constructeurs (3/3)

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).

Exemple : déclarations d'instances

Java

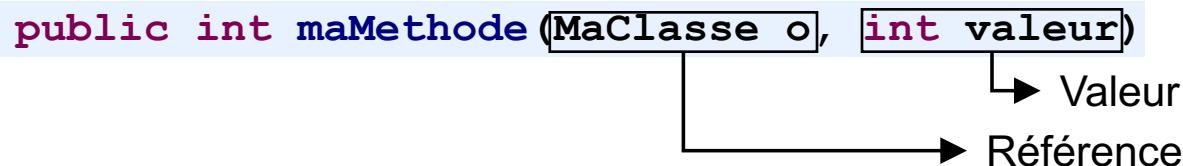
```
CompteBanque c1 = new CompteBanque(1, 100.0);  
CompteBanque c2 = new CompteBanque(2);  
CompteBanque c3 = new CompteBanque();
```

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

Appels implicites aux constructeurs

Java

- Que se passe t'il lors du passage d'un élément en argument d'une méthode ?
 - Type objet : la **référence** de l'objet est recopiée dans la pile.
 - Type simple : la **valeur** est recopiée dans la pile.
- Aucun appel implicite aux constructeurs en Java :
 - Toutes les instances sont créées à travers l'opérateur **new** qui fait appel aux constructeurs.
 - Exemple :



Constructeur(s) par défaut (1/2)

- Lorsque aucun constructeur n'est défini :
 - en Java : Il \exists un constructeur par défaut (sans argument)
 - en C++ : Il \exists un ctor par défaut (sans argument), un ctor de copie, un ctor de déplacement par défaut, des opérateurs de copie/déplacement par défaut et un dtor par défaut.
- S'il existe au moins un constructeur (Quel qu'il soit : avec ou sans arguments) :
 - en Java : le constructeur par défaut fourni automatiquement n'existe plus (il faut donc l'écrire si on en a besoin).
- Lorsqu'il existe un ou plusieurs constructeurs dans une classe :
⇒ on continue d'appeler abusivement « Constructeur par défaut » tout constructeur défini sans argument.

Constructeur(s) par défaut (2/2)

Constructeur sans arguments

Exemple :

```
class CompteBanque
{
    public CompteBanque ()
    {
        numero = 0 ; solde = 0 ;
    }
    /*...*/
}
/*...*/
class MainApp
{
    public static void main (String args[])
    {
        CompteBanque cpt = new CompteBanque () ;
        /*...*/
    }
}
```

Java

CompteBanque.java

Appel
explicite

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

Constructeur de copie

Java

Syntaxe : NomClasse (NomClasse objet);

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

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.

≠ Copie :

⇒ les deux objets pointeront vers une **même** zone mémoire

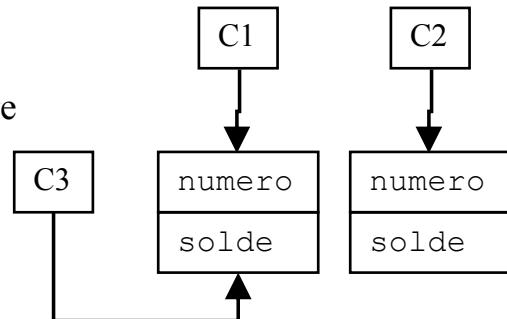
⇒ Il ne s'agit donc **pas** de deux instances différentes

Exemple :

```
class CompteBanque {  
    ...  
    public CompteBanque(CompteBanque c) { ... } // constructeur de copie  
    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 au  
                                         // constructeur de copie  
        CompteBanque c3 = c1; // copie de la référence de c1 dans c3  
        c2.AfficheAutreCompte(c1); // passage de la référence de c1  
    }  
}
```

Deep copy

Shallow copy



Gestion mémoire

Java

- Gestion implicite de la mémoire : Langages Objets purs (Java, SmallTalk, C#)
 - 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
 - Appelée par le Garbage Collector avant le nettoyage des instances.
 - **protected void finalize() throws Throwable { ... }**
- Gestion explicite de la mémoire : C++
 - Toute instance dynamiquement allouée par l'utilisateur doit être explicitement détruite par l'utilisateur (pas de Garbage Collecting).

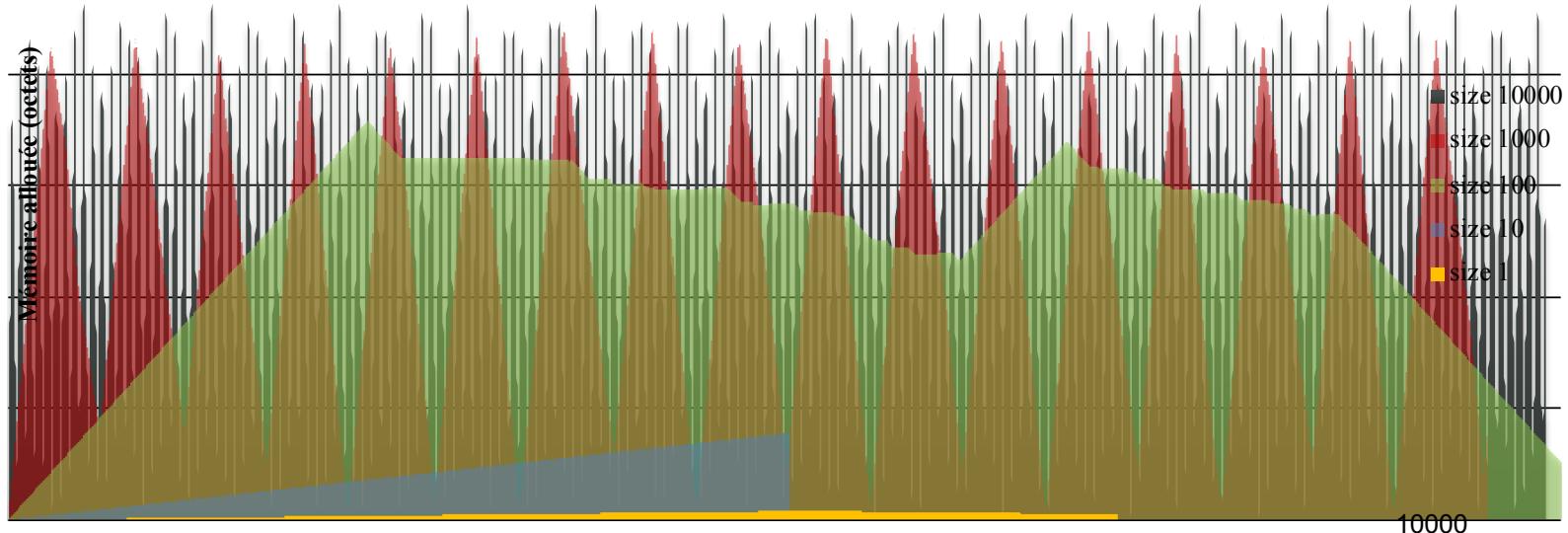
Garbage Collector

Java

- Tâche : allocation de n doubles m fois sans conserver de références :

```
while (loopCount < maxLoop)
{
    double[] dt = new double[size];
    loopCount++;
}
```

Mémoire allouée pour des instances de taille variables



Construction / Destruction

Java

- Construction
 - Les constructeurs doivent être appelés explicitement avec le mot clé **new**.
 - Pour les tableaux :
 - Les constructeurs doivent être appelés explicitement pour chacun des éléments. (voir Exemple - *Instanciation* en Java page 28)
- Destruction
 - Par le Garbage Collector dès qu'une instance n'est plus référencée.
 - Le Garbage Collector peut éventuellement déclencher la méthode **protected void finalize() throws Throwable** de l'instance à détruire.

variable ≠ instance

Java

- Une variable référence une instance (ou pas)
- Une variable n'existe que dans le bloc dans lequel elle est déclarée.
- Les instances non référencées sont destructibles (GB)

```
A a0;
{
    /*...
A a1;
A a2;
a1 = new A();
a2 = new A();
a0 = a2;
}
/*
 * à la fin du bloc les variables a1 et a2 disparaissent : l'instance
 * référencée par a1 n'est plus référencée, mais l'instance référencée
 * par a2 est encore référencée par a0. Seules les instances non
 * référencées sont destructibles par le Gargabe Collector
*/
```

** à la fin du bloc les variables a1 et a2 disparaissent : l'instance
* référencée par a1 n'est plus référencée, mais l'instance référencée
* par a2 est encore référencée par a0. Seules les instances non
* référencées sont destructibles par le Gargabe Collector*

2.6. Membres de classe

Membres de classes

- Eléments 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 instances de cette classe (les objets).
- Méthodes de classes
 - Les méthodes de classe ne peuvent accéder qu'aux **attributs** de classe 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 - 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
mais
syntaxiquement
correct

Membres de classe – *SmallTalk / Python*

- Metaclasse
 - Chaque classe est une instance de sa métaclassse
 - Les variables de classes sont des variables d’instance de la metaclasse.
 - Les membres de classe sont donc des membres d’instance de la metaclasse.
 - Alors que les membres ordinaires sont membres d’instance de la classe.

2.7 Classes Internes et Imbriquées

Inner / Nested / Local classes

- Classes déclarées à l'intérieur d'une classe
- Java
 - Classes internes
 - Les classes internes forment une « clôture » (closure) dans la mesure où elles « capturent » l'environnement dans lequel elles sont définies :
 - Une classe interne est un attribut comme les autres, elle a donc accès aux membres de la classe dans laquelle elle est déclarée.
 - Classes imbriquées
 - Une classe imbriquée n'a pas accès aux membres de la classe dans laquelle elle est définie.
 - Déclaration avec le mot clé « static »
 - Classes locales (anonymes)
 - Classes déclarées dans un bloc (i.e. un méthode par exemple), forment une clôture comme les classes internes : capturent l'environnement de la classe dans laquelle elles sont déclarées.
- C++
 - Classes imbriquées seulement
 - Déclaration sans mot clé « static »

```
return new Tabliterator<E>(container);
```

```
return (innerIndex < nextElementIndex);
```

EnsembleTab <E>

```
# container: E[] [0..*]  
# nextElementIndex: int  
  
+ iterator(): Iterator<E>
```

Tabliterator <F>

```
# innerIndex: int  
# table: F[] [0..*]  
  
+ Tabliterator(F[])  
+ hasNext(): boolean  
+ next(): F  
+ remove()
```

Classes anonymes

Java

- Classes implémentant une interface directement dans une variable ou un argument nécessitant cette interface:
 - Exemple : `EventQueue.invokeLater` attend une instance d'une classe implémentant l'interface `Runnable`

```
// Création d'une fenêtre
final ClientFrame frame = new ClientFrame(...);

// Insertion de la frame dans la file des évènements GUI : Runnable anonyme
EventQueue.invokeLater(new Runnable()
{
    public void run()
    {
        try
        {
            frame.pack();
            frame.setVisible(true);
        }
        catch (Exception e)
        {
            ...
        }
    }
});
```

- Les classes anonymes (au même titre que les classes internes) ont accès aux membres de la classe dans laquelle elles sont déclarées.
- Toutefois, les classes anonymes ne peuvent accéder qu'aux variables « final » de leur environnement (scope)

} Note : Runnable ne définit qu'une seule méthode abstraite « run », c'est donc une « functional interface »

2.8. Polymorphisme

Polymorphisme - La surcharge

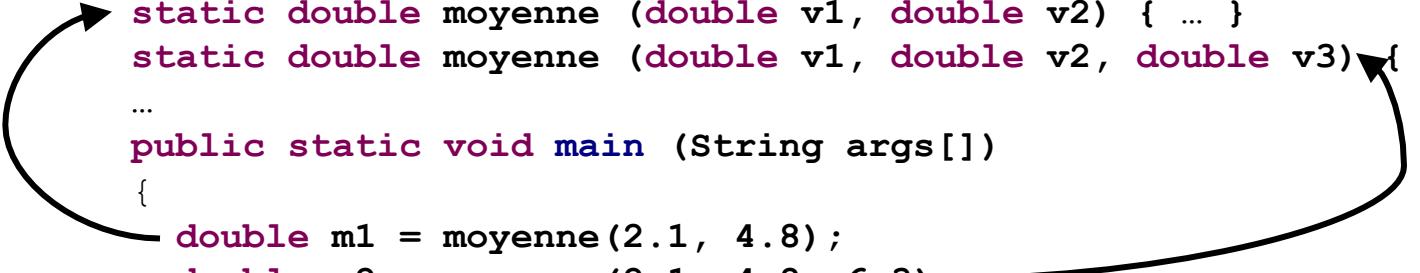
- Capacité de donner le même nom à des méthodes «similaires» ayant des listes d'arguments distinctes par le **nombre** et/ou le **type** des arguments.

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 Java : *moyenne de 2 valeurs entières ou réelles*

```
...
static double moyenne (double v1, double v2) {...}
static int moyenne (int v1, int v2) {...}
static double moyenne (int v1, int v2) {...}
// Erreur de compilation
...
```

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

Polymorphisme - Conversion de types

```
public class Stats {  
    public double moyenne(double v1, double v2) {...}  
    public double moyenne(int v1, int v2) {...}  
    public double moyenne(double v1, double v2, double v3) {...}  
    public double moyenne(double... values) {...}  
    // Erreur de compilation : duplicate method moyenne(double, double)  
    public int moyenne(double v1, double v2) {...}  
}
```

Stats.java

```
...  
public static void main(String[] args){  
    Stats stats = new Stats();  
    double r1 = stats.moyenne(2, 5);           // moyenne(int, int)  
    double r2 = stats.moyenne(r1, 5);          // moyenne(double, double)  
    double r3 = stats.moyenne(5, r1);          // moyenne(double, double)  
    double r4 = stats.moyenne(5, (int)r1); // moyenne(int, int)  
    double r5 = stats.moyenne(5, r2, r1); // moyenne(double, double, double)  
    double r6 = stats.moyenne(r1, r2, r3, r4, r5, 5); // moyenne(double... )  
}
```

TestStats.java

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++ (11+ uniquement)
 - grâce aux constructeurs délégués (utilisation d'un constructeur dans un autre constructeur) :
 - Smalltalk: Il n'y a pas vraiment de constructeurs, seulement la réimplémentation de la méthode `new`.

Appel du constructeur par défaut dans un constructeur valué



Polymorphisme – liste d'arguments variable

Java

- Nombre variable d'arguments de même type dans une méthode.

```
public class VarArgs
{
    public static void vaTest(String msg, int... v)
    {
        System.out.print(msg + " (" + v.length + "): ");
        for (int x : v)      // Java Generics foreach
            System.out.print(x + " ");
        System.out.println();
    }

    public static void main(String[] args)
    {
        vaTest("One vararg", 10);
        vaTest("Three varargs", 1, 2, 3);
        vaTest("No varargs");
    }
}
```

sortie

```
One vararg (1): 10
Three varargs (3): 1 2 3
No varargs (0):
```

2.9. 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

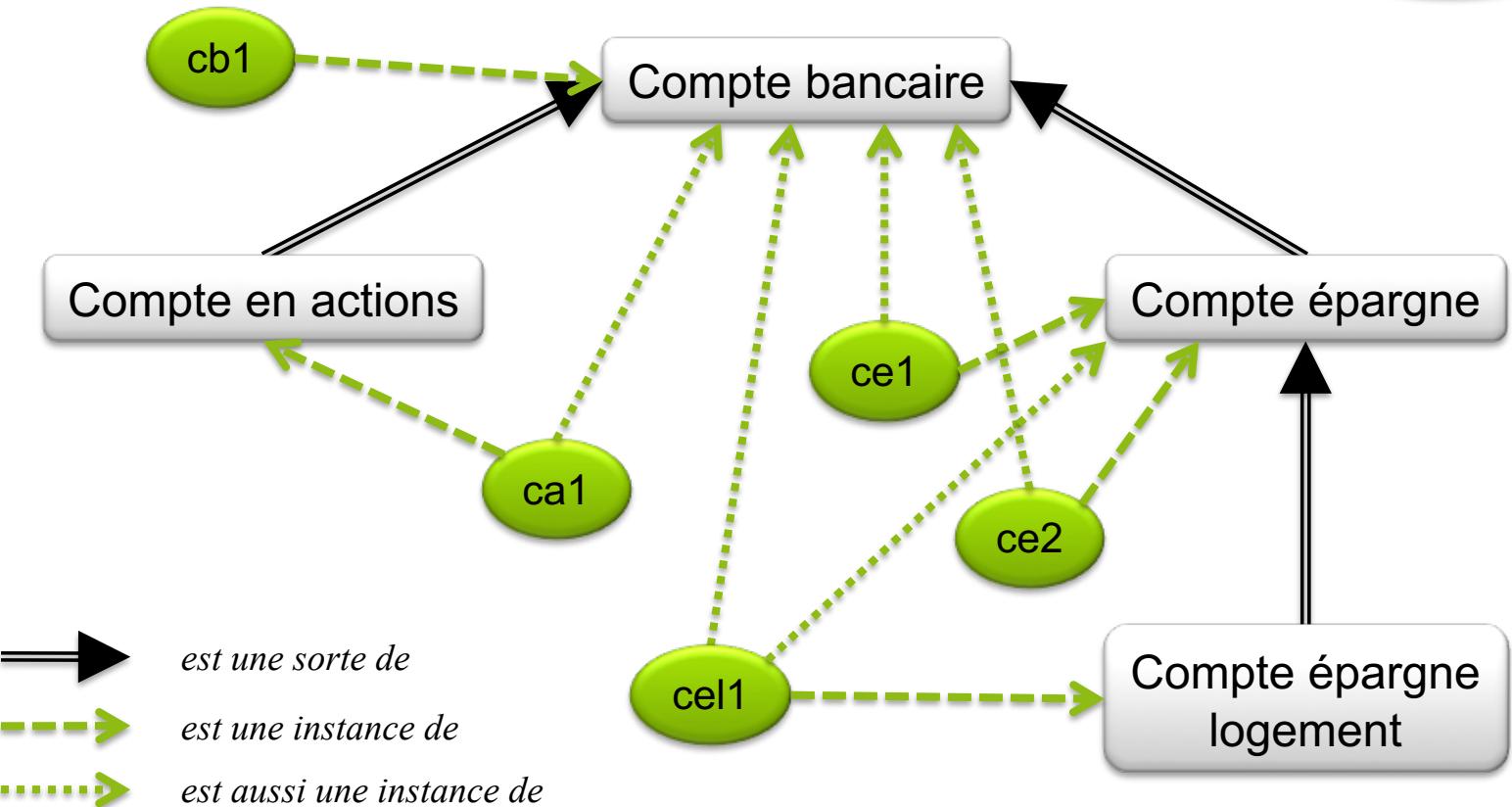
Des **EXTENSIONS** peuvent se faire

- 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, C#, Smalltalk)
 - Sans arbre d'héritage : Une classe peut ne pas avoir d'ancêtre (C++)
- Héritage simple : (Java, C#, Smalltalk)
 - Une et une seule classe mère
 - Java, C# : Une classe hérite d'une seule autre mais peut implémenter plusieurs interfaces (voir page 83)
- 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

Compte

valeur du solde (*accès protégé*)

consultation(), retrait (const double), ajout (const double)

→ Compte banque Toto

valeur du solde (*accès protégé*)

consultation(), **retrait (double) (*interdit si négatif*)**,
ajout (double)

Redéfinition : surcharge

→ Compte en actions

compte avec un tableau de 10 numéros d'actions et leur vente

int tabAct[10] (*accès protégé*), vente (int) ← Extensions

→ Compte bloqué

compte avec retrait interdit ← Restriction

Accessibilité

- Accès entre classes (rappel)
 - Par défaut :
 - Java : `public` mais limité au package (mode implicite « package »)
 - C++ / 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 l'accessibilité d'une méthode en la redéfinissant dans une classe héritière, mais cette accessibilité ne peut pas être **plus** restrictive que dans la classe ancêtre.

Accès entre classes

Java

Instances d'une même classe

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

Membres d'une classe définie par héritage

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

```
public 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 testM(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 testH(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
    hpub = 11; // OK
    hprot = 22; // OK
    hpriv = 33; // OK
    mpub = 11; // OK
    mprot = 22; // OK
    mpriv = 33; // interdit
}
```



Modification

Finalisation

Java

- 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

Java

- Recouvrement ou remplacement ?
 - Java : remplacement uniquement des méthodes possédant des signatures* identiques. * : nom + nombre & types des arguments

```
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 découvert;
    public void ouvrir(double montant,
                       double découvert)
    {
        ouvrir();           // OK forme héritée
        ouvrir(0.0);       // OK surcharge
        super.ouvrir(montant); // OK forme héritée
        ouvrir(montant,100.0); // OK nouvelle forme
        this.découvert = découvert;
    }
...
}
```

Nouvelle
forme

```
public void ouvrir(double montant)
{
    ouvrir();           // OK forme héritée
    ouvrir(0.0);       // OK surcharge
    super.ouvrir(montant); // OK forme héritée
    this.découvert = 0;
}
```

Surcharge



Appels récursifs infinis !

- C++ : La redéfinition recouvre toutes les surcharges héritées

Héritage/Hiérarchie des constructeurs

- Constructeurs : Ancêtres → Descendants
- Destructeur ou finalize : Descendants → Ancêtres
- Les constructeurs ne sont pas hérités, toutefois \exists un constructeur par défaut lorsqu'il n'existe aucun constructeur dans la classe héritière*.
 - 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) : super (...);
 - 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.

* : ce constructeur par défaut appellera le constructeur par défaut de la classe mère qui appellera ...

Hiérarchie des constructeurs (1/2)

Java

```
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
{
    public static void main (String args[])
    {
        A a = new A(1);
        B b = new B(3);
    }
}
```

Première instruction
« implicite » : **super()** ;

Appel explicite du
constructeur
A(int val)

Hiérarchie des constructeurs (2/2)

Java

- 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;
    }

    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() à travers super() implicite
        decouvert = 0;
    }

    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 ancêtres
- 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 (Dynamic/Late Binding*)

- Problème
 - Choix dynamique de la réalisation des fonctions membres exécutées ≠ (surcharge = résolution à la compilation, statique)
 - Résolution/Lien statique : détermination lors de la **compilation**
 - Résolution/Lien dynamique : détermination lors de l'**exécution**, ce qui revient à toujours exécuter la méthode de l'objet effectivement instancié en mémoire.
 - Pas de lien dynamique sans polymorphisme d'héritage
- Dans les langages
 - C++ :
 - Par défaut : Résolution statique (Early Binding)
 - Résolution dynamique si la méthode concernée est déclarée virtuelle (mot clé **virtual**) dans la classe mère, sauf dans les constructeurs
 - Java : Résolution dynamique
 - Smalltalk : Résolution dynamique

* ≠ Dynamic Link[age]

Lien dynamique – Exemples (1/2)

Java

```
public class A {  
    protected int a;  
    public A(int value) {  
        a = value;  
        printInstance("Construction de A");  
    }  
    public void printInstance(String msg){  
        out.println(msg + " : " + toString());  
        // ou bien + this  
    }  
    public String toString(){  
        return new String("A::a = " + a);  
    }  
}
```

A.java

```
public class B extends A {  
    protected int b;  
    public B(int value1, int value2) {  
        // appel du constructeur de A qui  
        // appellera toString  
        super(value1);  
        b = value2;  
        printInstance("Construction de B");  
    }  
    // redéfinition de toString  
    public String toString() {  
        return new String("B::a = " + a +  
                          ", B::b= " + b);  
    }  
}
```

B.java

```
public class TestAB {  
    public static void main(...){  
        ① A aInstance = new A(1);  
        ② A bInstance = new B(2,3);  
        // Appels implicites à toString  
        ③ aInstance.printInstance("instance  
        de A");  
        ④ bInstance.printInstance("instance  
        de B");  
    }  
}
```

TestAB.java

résultats

- ① Construction de A : A::a = 1
- ② Construction de A : B::a = 2, B::b= 0
Construction de B : B::a = 2, B::b= 3
- ③ instance de A : A::a = 1
- ④ instance de B : B::a = 2, B::b= 3

Lien dynamique – Exemples (2/2)

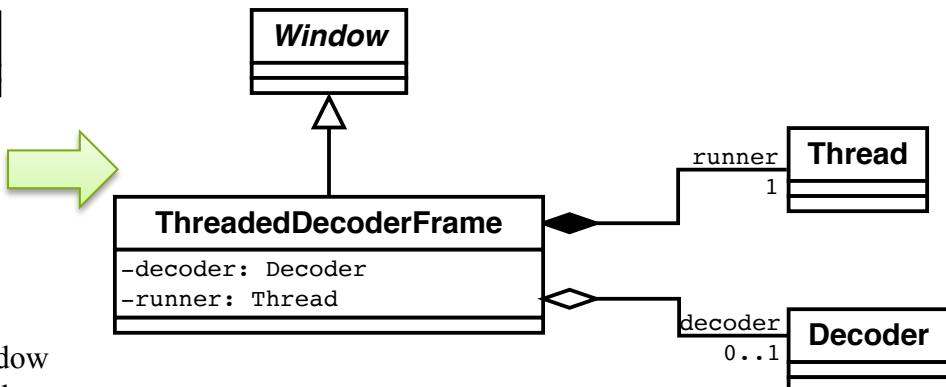
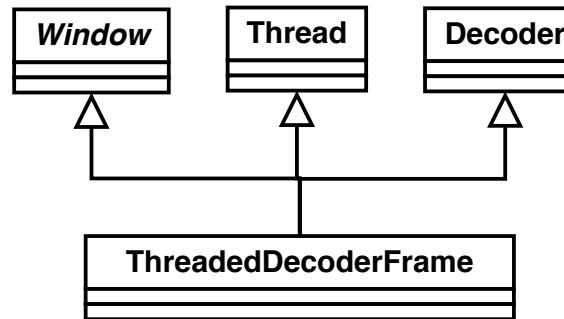
Java

1. A aInstance = **new A(1)** ;
 - A::A(int) \Rightarrow printInstance(...) \Rightarrow println(...) \Rightarrow A::toString()
 - Construction de A : A::a = 1
2. A bInstance = **new B(2,3)** ;
 - A::A(int) \Rightarrow printInstance(...) \Rightarrow println(...) \Rightarrow B::toString()
 - Construction de A : B::a = 2, B::b = 0
 - B::B(int, int) \Rightarrow printInstance(...) \Rightarrow println(...) \Rightarrow B::toString()
 - Construction de B : B::a = 2, B::b = 3
3. aInstance.**printInstance("instance de A")** ;
 - println(...) \Rightarrow A::toString()
 - instance de A : A::a = 1
4. bInstance.**printInstance("instance de B")** ;
 - println(...) \Rightarrow B::toString()
 - instance de B : B::a = 2, B::b = 3

Lien dyn.
Mais accès à
b non initialisé

L'héritage multiple

- Capacité d'une classe à hériter de plusieurs classes mères
 - Polymorphisme d'héritage préservé.
 - (mauvais) Exemple : ThreadedDecoderFrame une sorte de fenêtre destinée à afficher la progression d'un décodeur fonctionnant de manière indépendante du reste de l'application.



Solution par héritage : T...D...F est une Window mais aussi un Thread et aussi un Decoder alors qu'il ne spécialise ni Thread, ni Decoder

Solution par composition : T...D...F est une Window qui lancera un Decoder dans un Thread.

- Implémentations
 - Java : Une classe ne peut hériter que d'une seule classe mère, mais peut implémenter plusieurs « interfaces » (voir classes abstraites)
 - C++ : Une classe peut hériter de plusieurs classes mères.

2.10 L'introspection

Introspection (Reflection) (1/3)

Java

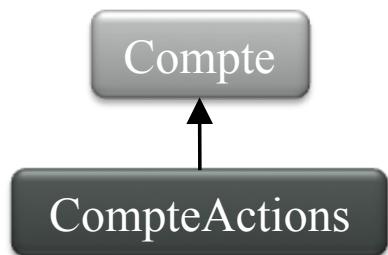
- Capacité que possède une classe à se décrire elle-même.
 - Introspection au travers de la classe « Class » dont les instances représentent une « signature » propre à chaque classe.
 - Utilisation du polymorphisme d'héritage : toute classe descend de la classe Object qui possède la méthode d'introspection:
 - `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 (2/3)

Java

- Class literal : `.class` appliqué aux types

- Opérateur de comparaison de types : `instanceof`

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

Introspection (3/3)

Java

- **instanceof** vs **isInstance** (...) vs **==**

- **instanceof**

- Opérateur du langage.
 - Fonctionne avec un type :

- **isInstance** (...)

- Méthode de la classe Class
 - Fonctionne donc sur une instance de la classe Class obtenue dynamiquement.

- **==** ou méthode **equals** entre deux instances de la classe Class



Modification

```
public class A  
{...}  
public class B  
extends A {...}  
  
A a = new A();  
B b = new B();
```

b instanceof A

true

a.getClass() .
isInstance(b)

true

a.getClass() ==
b.getClass()

false

2.11 Classes Abstraites

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 abstraites/pures de la classe abstraite

C++ : fonctions/méthodes pures

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

Java :

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

* : Mais pouvant malgré tout proposer des implémentations par défaut en Java 8

Smalltalk : méthodes abstraites → Subclass responsibility

Séparer la conception de l'implémentation !

Classes abstraites

Java

- 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); // Ok
        uneMethode(ab); // Ok grâce au lien dynamique
    }
}
```

Classes interfaces

Java

- Interfaces : pseudo-classes non implémentées*, ne contiennent que des :
 - Méthodes d'instance abstraites : *public abstract*
 - Méthodes de classe : *public static*
 - Constantes de classe : *public static final*
 - * Des méthodes (implémentées) par défaut : *public default* (Java 8)
 - Les Interfaces représentent des « comportements » à implémenter
- On peut considérer la hiérarchie des interfaces comme // à la hiérarchie des classes.
- Une classe concrète « implémente » une ou des interfaces
 - 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 (méthode **void run()**) qui lui permet de s'exécuter dans un thread (unité d'exécution concurrente).
- une classe Java peut « implémenter » plusieurs interfaces
 - Héritage multiple de « comportements »

Classes abstraites / interfaces : exemple

Java

- 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 Polygone
{
    protected int nbPoints;
    protected Point[] points;
    public Polygone(int nbPoints)
    {
        this.nbPoints = nbPoints;
        points = new Point[nbPoints];
    }
    public abstract double aire();
}
```

```
class Triangle extends Polygone
implements Figure
{
    // constructeur : propre à Triangle
    Triangle()
    {
        super(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
    //       Polygone
    public double aire()
    { ... }
}
```

Interface : exemple

Java 8

```
public interface AnswerComputer
{
    // Une constante de classe
    public static final int TheAnswer = 42;
```

```
// méthodes abstraites
public abstract int nextAnswer();
public abstract int currentAnswer();
@Override
public abstract String toString();
```

```
// Une méthode de classe publique
public static int randomAnswer()
{
    return TheAnswer +
        new Random().nextInt();
}
```



Pb : comment appeler ce « found » dans une implémentation fille si elle est redéfinie ?

```
// méthodes avec une
// implémentation par défaut
public default int assessTheAnswer()
{
    int answer = currentAnswer();
    if (answer > TheAnswer)
    {
        return 1; // plus grand
    }
    else if (answer < TheAnswer)
    {
        return -1; // plus petit
    }
    return 0; // égal
}
→ public default boolean found()
{
    return assessTheAnswer() == 0;
}
```

Interfaces fonctionnelles (1/3)

Java 8

- Un grand nombre d'interfaces du JDK ne déclarent qu'une seule méthode abstraite : Single Abstract Method interfaces. Exemples (les mots clé **public abstract** sont implicites) :
 - Runnable → `void run();` ✓ ✓
 - Comparator → `int compare(T o1, T o2);` ✓ ✓
 - ActionListener → `void actionPerformed(ActionEvent e);` ✓ X
 - Ces interfaces sont souvent implémentées par des classes anonymes (voir page 52) : 1 seule méthode à écrire.
- Java 8 introduit la notion d'Interfaces Fonctionnelles : Interfaces ne déclarant qu'une seule méthode abstraite sauf
 - des redéfinitions de méthodes déjà définies dans la superclasse Object ou dans les interfaces parentes,
 - des implémentations par défaut,
 - des méthodes de classe.
 - l'annotation **@FunctionalInterface** force le compilateur à vérifier ces conditions.

Interfaces fonctionnelles (2/3)

Java 8

```
@FunctionalInterface  
public interface SimpleFuncInterface  
{  
    public void doWork();  
    @Override  
    public String toString();  
    @Override  
    public boolean equals(Object o);  
}
```

Seule méthode d'instance abstraite

Méthodes définies dans la superclasse **Object**

```
@FunctionalInterface  
public interface ComplexFuncInterface extends SimpleFuncInterface  
{  
    default public void doSomeWork()  
    {  
        System.out.println("Working in interface default imp...");  
    }  
    default public void doSomeOtherWork()  
    {  
        System.out.println("Working in other interface default imp...");  
    }  
}
```

- Aucune nouvelle méthode d'instance abstraite
- 2 implémentations par défaut

Toujours une interface fonctionnelle

```
public class ComplexFuncClass implements ComplexFuncInterface  
{  
    @Override  
    public void doWork()  
    {  
        System.out.println("Working in class impl ...");  
        ComplexFuncInterface.super.doSomeOtherWork();  
    }  
}
```

ComplexFuncClass utilisera les implémentations de **toString** et **equals** fournies par la superclasse **Object**

Forme particulière de l'appel explicite des méthodes par défaut de l'interface mère

Interfaces fonctionnelles (3/3)

Java 8

```
public class SimpleFuncInterfaceTest
{
    public static void carryOutWork(SimpleFuncInterface sfi)
    {
        sfi.doWork();
    }

    public static void main(String[] args)
    {
        // Instance d'une classe implémentant l'interface SimpleFuncInterface
        carryOutWork(new ComplexFuncClass());
    }

    // Instance anonyme de l'interface SimpleFuncInterface
    carryOutWork(new SimpleFuncInterface() {
        @Override
        public void doWork()
        {
            System.out.println("Doing some work in anon. interface impl...");
        }
    });

    // Lambda expression remplaçant l'interface SimpleFuncInterface
    carryOutWork(() -> System.out.println("Doing some work in lambda expr..."));
}
}
```



La signature de cette lambda expression (**void()**) correspond à la signature de la seule méthode d'instance abstraite (**void doWork()**) de l'interface fonctionnelle **SimpleFuncInterface**

Lambda Expressions

Java 8

- Nécessité
 - Java : Remplacer* les instanciations anonymes des SAM (Single Abstract Method Interfaces) par des expressions plus simples.
- Usage
 - Les lambda expressions peuvent être considérées comme des « fonctions anonymes et locales » pouvant être utilisées là où l'on en a besoin
 - Elles peuvent donc se substituer à des SAM dans d'autres expressions sans avoir à déclarer explicitement ces SAM.

* : Sans toutefois exclure

Lambda Expression

Java 8

- Syntaxe
 - $(arguments) \rightarrow \{ corps \}$
- Type : La signature de l'expression doit correspondre à une interface fonctionnelle.
- Exemples

```
public interface Runnable
{
    public abstract void run();
}
```

```
Runnable runner =
() -> System.out.println("Running ...");

Thread t = new Thread(runner);
t.start();
```

```
public void printPersons(List<Person> roster,
                        PersonChecker tester)
{
    for (Person p : roster)
    {
        if (tester.test(p))
        {
            p.printPerson();
        }
    }
}
```

```
@FunctionalInterface
interface PersonChecker
{
    public abstract boolean test(Person p);
}
```

```
printPersons(roster,
             p -> (p.getGender() == Person.Sex.MALE) &&
                    (p.getAge() >= 18) &&
                    (p.getAge() <= 25));
```

Lambda Expression

Java 8

- Exemples (suite)

```
public class Calculator
{
    interface IntegerMath
    {
        int operation(int a, int b);
    }

    public int operateBinary(int a, int b, IntegerMath op)
    {
        return op.operation(a, b);
    }

    public static void main(String... args)
    {
        int epsilon = 1;
        Calculator myCalc = new Calculator();
        IntegerMath addition = (a, b) -> a + b + epsilon;
        IntegerMath subtraction = (a, b) -> {return a - b;};
        System.out.println("40 + 2 = " +
                           myCalc.operateBinary(40, 2, addition));
        System.out.println("20 - 10 = " +
                           myCalc.operateBinary(20, 10, subtraction));
    }
}
```

Corps sans accolades
(pas de return)

Type de retour implicite

Les lambdas capturent leur environnement (comme les classes internes), on peut donc accéder aux attributs de la classe, mais les lambda expressions n'ont accès qu'aux variables locales finales (ou effectivement finales) : on peut accéder à **epsilon** mais pas le modifier.

2.12 Composition

- Composition : Est composé de

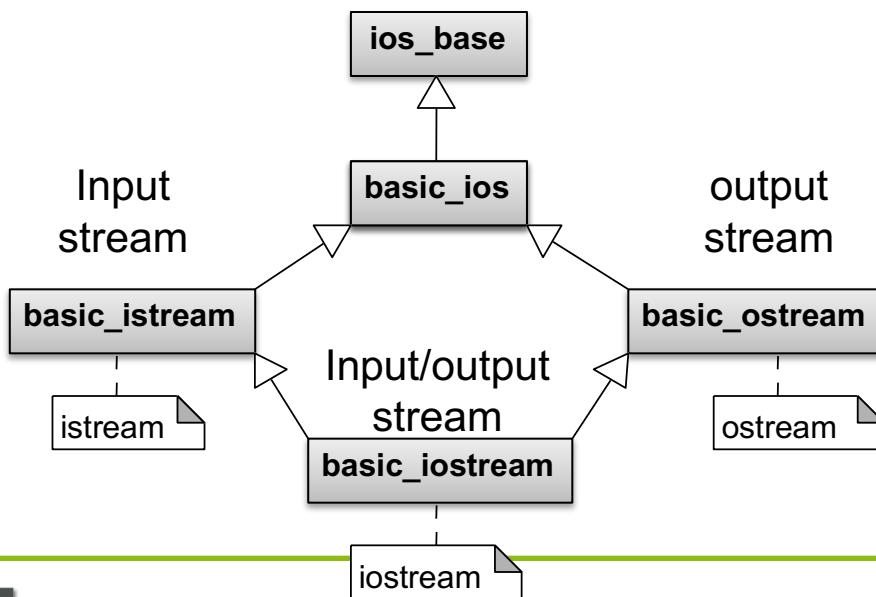
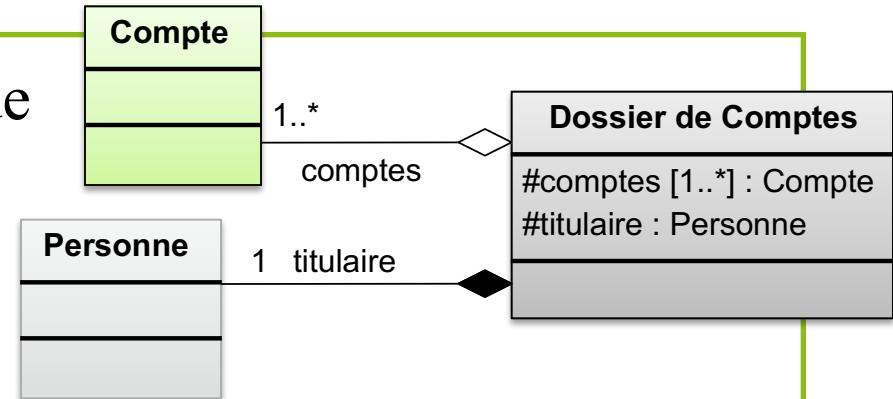
- Exemple

- Dossier de comptes

- Composition par héritage

- Nécessite l'héritage multiple

- Exemple : Les flux d'entrée sortie en C++



2.13 Généricité

Généricité

- Définition
 - Capacité d'une classe à manipuler des entités non-(*encore*)-typées
 - Exemples : Tableau de ..., Listes de ... \Rightarrow Collections
- 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 pour instancier un **patron de classe en classe**.
 - Exemple : instantiation d'un tableau d'entiers à partir d'un tableau générique : Tableau<int> tab; instantiation de la classe générique Tableau<T> avec des int

Obsolète

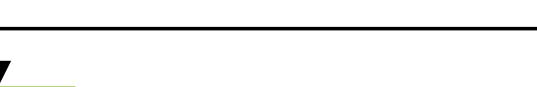
Java Generics

Java

- 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 généréricité
- Inconvénient : Existence à la compilation seulement (Type Erasure à l'exécution)

Polymorphisme d'héritage vs Java Generics

Java

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

```
public class MaCollection implements Collection {  
    ...  
    public boolean add(Object e) {...}  
    public Iterator iterator() {...} ...  
}  
...  
MaCollection col = new MaCollection();  
col.add(new Integer(5));  
Integer x = (Integer)col.iterator().next();
```

Collection de « Object »

Cast explicite

Collection de « E »

- Généricité avec Java Generics

```
public class MaCollection<E> implements Collection<E> {  
    ...  
    public boolean add(E e) {...}  
    public Iterator<E> iterator() {...} ...  
}  
...  
MaCollection<Integer> col = new MaCollection<Integer>();  
col.add(new Integer(5));  
Integer x = col.iterator().next();
```

Variable de type

Paramètre de type

Collection de « Integer »

Généricité : Sous typage

Java

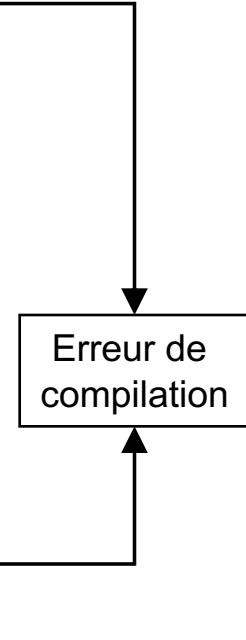
- Sous typage
 - Exemple

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

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

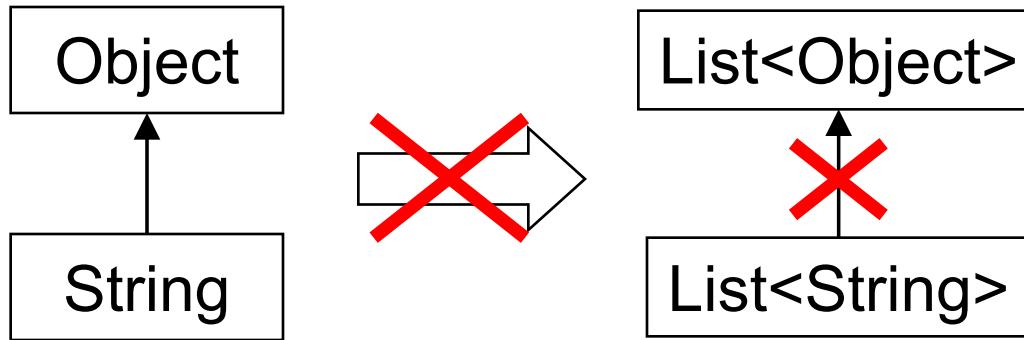
```
lo.add(new Object());  
String s = ls.get(0); //
```

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



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.
 - Type générique plutôt car ce n'est pas un héritage
- Pour désigner le supertype de tout type de liste on peut utiliser le « wildcard » : ?
 - `List<?>`
 - `List<?>`
 - `List<String>`
 - `List<Object>`

Généricité : utilisation de ?

Java

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

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

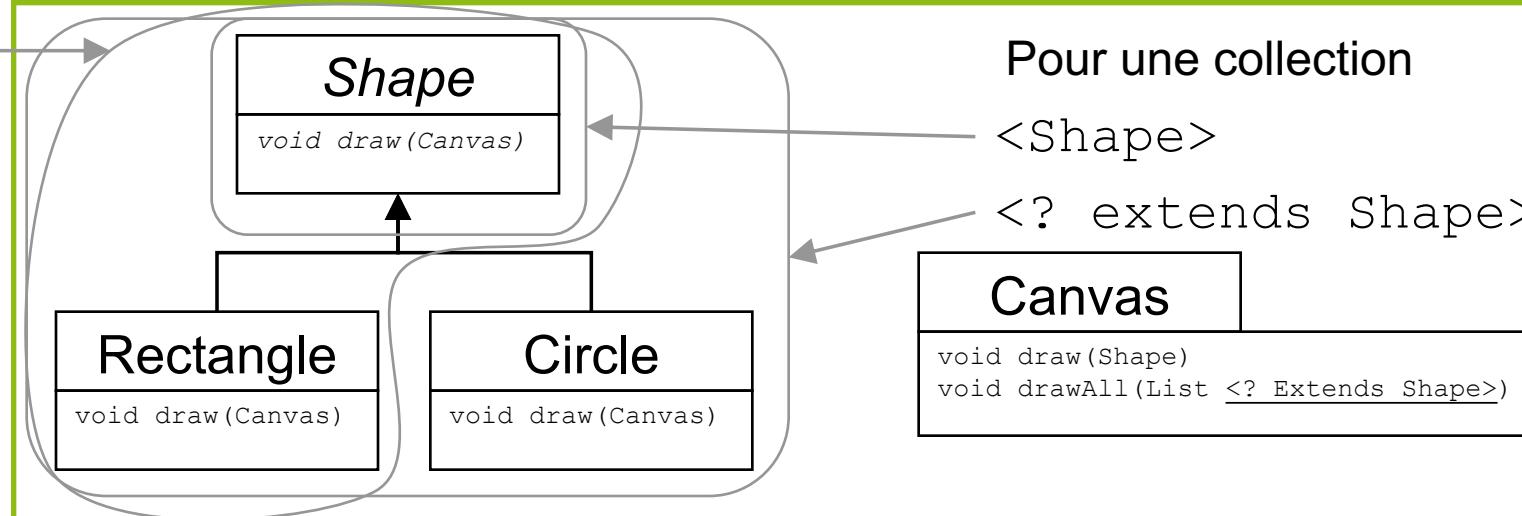
- Écriture dans une Liste Générique

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

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

Généricité : Wildcards bornés

Java



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

Tableaux génériques

Java

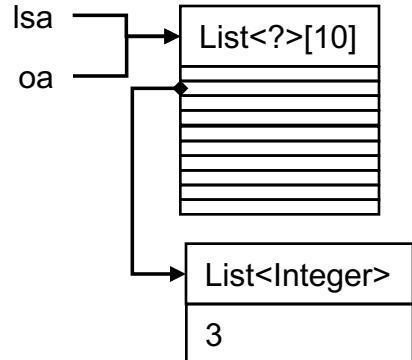
- Le type des éléments d'un tableau effectif ne doit pas être une variable de type ou un paramètre de type, sauf un wildcard non borné.

```
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
```

new T(...)
new T[...]
interdits

- 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 (~ patrons de fonctions C++).
 - 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) {  
        col.add(o); }  
}
```

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

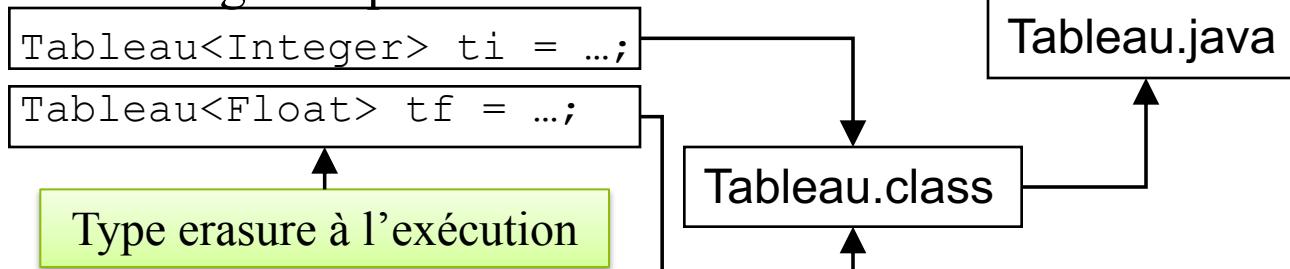
- Méthode Générique :

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

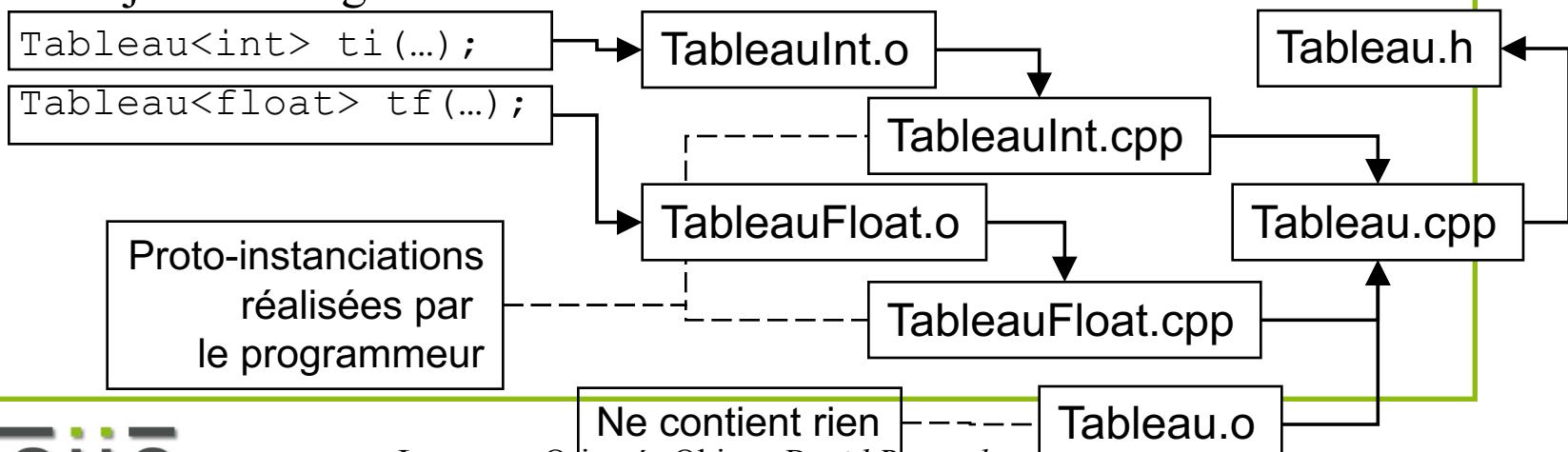
Paramètre de type

Généricité : Java vs C++

- Différence entre Java Generics et les Templates C++
 - Java : un seul fichier .class est généré 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

Java

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

- ```
Collection cs = new ArrayList<String>();
if (cs instanceof Collection<String>) { ... }
```
- ```
Collection<String> cstr = (Collection<String>) cs;
```
- ```
public static
<T> T mauvaisCast(T t, Object o) { return (T) o; }
```

Erreurs de Compilation

Unchecked Warning

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

## 2.14 Robustesse

# 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 définis dans les langages Eiffel et C# :  $\begin{cases} \text{require(s)} \\ \text{ensure(s)} \end{cases}$
- 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* }  
...  
**finally** { *bloc exécuté qq soit le nombre de clauses catch* }
- 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
happened, 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;
 }

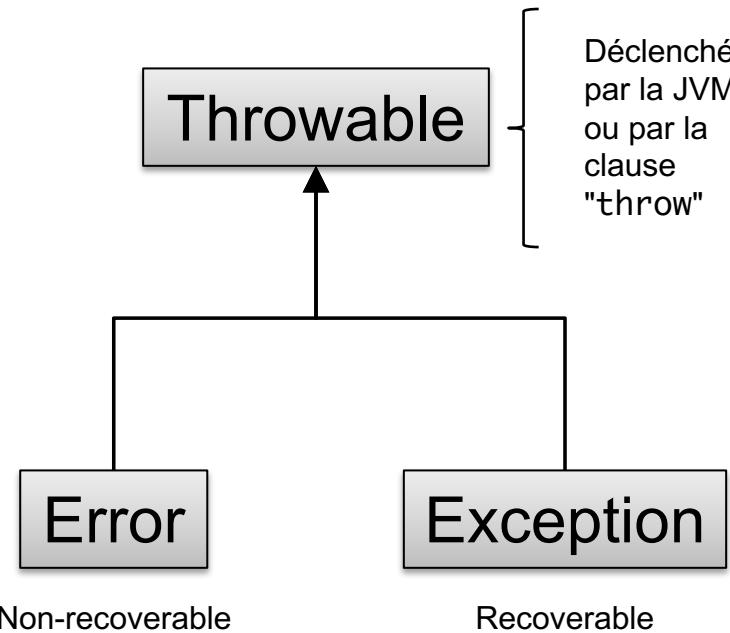
 public static void main (String args[])
 {
 Fraction frac1 = new Fraction();
 Fraction frac2 = new Fraction(1.0,
2.0);

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

# Throwable

Java

- **Error (Non Recoverable)** :  
Problème important qu'une application ne devrait pas « catcher ».
- **Exception (Recoverable)** :  
Problème ordinaire qu'une application doit « catcher ».



# C++

# 1.5 Rappels C & spécificités de C++

C

- Modules (.h/.c) → Classes (.h[pp]/.cpp)\*
  - Compilation séparée des classes comme des modules
  - .h (header) contient les déclarations
  - .cpp contient l'implémentation (appelée « definition » en C++)
- Passage d'arguments (1/3)

- par valeur :

```
int f(int val)
{
 val++;
 return val;
}
```

main

```
int i = 15;
int r = f(i);
// i == 15
// r == 16
```

- une copie de la « valeur » val est réalisée dans pile d'appel
- cette copie est incrémentée, puis copiée dans la valeur de retour
  - La valeur de la variable **i** passée en argument n'est jamais modifiée

\* : ! .hpp est souvent réservé aux headers contenant directement une implémentation (pas de .cpp).

# 1.5 Rappels C & spécificités de C++

C

- Passage d'arguments (2/3)

- par adresse :

```
int f(int * ptr)
{
 (*ptr)++;
 return *ptr;
}
```

main

```
int i = 15;
int r = f(&i);
```

- une copie de la « valeur » `ptr` est réalisée dans pile d'appel ⇒ une copie de l'adresse `ptr`
    - Cette adresse est déréférencée (`*ptr`) puis la valeur ainsi obtenue est incrémentée (`++`) \*
    - la valeur de l'adresse déréférencée est copiée dans la valeur de retour
      - la valeur pointée par l'adresse passée en argument a été modifiée

\*: notez les parenthèses et consultez les priorités des opérateurs en C/C++ [page 159](#)

$(*\underline{ptr})++ \neq *\underline{ptr}++$

# 1.5 Rappels C & spécificités de C++

C++

- Passage d'arguments (3/3)

- par référence :

```
int f(int & ref)
{
 ref++;
 return ref;
}
```

main

```
int i = 15;
int r = f(i);
```

- une copie de l'adresse référencée par **ref** est réalisée dans pile d'appel
    - la valeur référencée par **ref** est incrémentée
    - la valeur référencée par **ref** copiée dans la valeur de retour
      - la valeur référencée par la référence passée en argument a été modifiée
    - Rq : il ne peut y avoir de référence vers void ni de référence de référence.

# 1.5 Rappels C & spécificités de C++

C

- Lvalues et Rvalues (1/2)

- Première approche :

- lvalue : à gauche (ou à droite) d'une affectation
    - rvalue : à droite d'une affectation

- deuxième approche :

- lvalue : « locator value » toute valeur possédant un emplacement mémoire
      - On peut toujours obtenir l'adresse d'une lvalue : opérateur `&lvalue`
    - rvalue : Tout ce qui n'est pas une lvalue
      - Valeur constante
      - Résultat d'une expression ou d'une fonction renvoyant une valeur (mais pas une référence)
      - On ne peut pas obtenir l'adresse d'une rvalue

main

```
int i = 15;
int r = f(i);
```

lvalues

rvalues

# 1.5 Rappels C & spécificités de C++

C

- Lvalues et Rvalues : exemples (2/2)

```
int & retRef() {return ...;} // retourne une ref. vers une valeur
int retVal(){return ...;} // retourne une valeur
int main(int argc, char** argv)
{
 // lvalues:
 int i = 42;
 i = 43; // ok, i is an lvalue
 int* p = &i; // ok, i is an lvalue and &i is an rvalue
 retRef() = 42; // ok, retRef() is an lvalue
 int* p1 = &retRef(); // ok, retRef() is an lvalue

 // rvalues:
 int j = 0;
 j = retVal(); // ok, retVal() is an rvalue
 int* p2 = &retVal(); // error, cannot take the address of an rvalue
 j = 42; // ok, 42 is an rvalue
 return 0;
}
```

# 1.5 Rappels C & spécificités de C++

C++11

- Lvalues & rvalues references

- lvalue reference : référence à un emplacement mémoire définit par une lvalue :

- `Type a = 3;` // Ok: *a is an lvalue*
- `Type & lvra = a;` // Ok: lvalue reference to lvalue
- ~~`Type & lrvb = 15;` // Error: non const lvalue reference to an rvalue~~
- `const Type & clvrc = 15;` // Ok: const lvalue reference to an rvalue
- `Type * pa = &a;` // Ok: address of an lvalue
- `Type * plvra = &lvra;` // Ok: address of an lvalue reference

- rvalue reference : référence (temporaire) à une rvalue

- ~~`Type && prvra = a;` // Error: rvalue reference to an lvalue~~
- `Type && rrvb = 15;` // Ok : rvalue reference to rvalue
- `Type * prvr = &rrvb;` // Ok: address of an rvalue reference (ahem!)

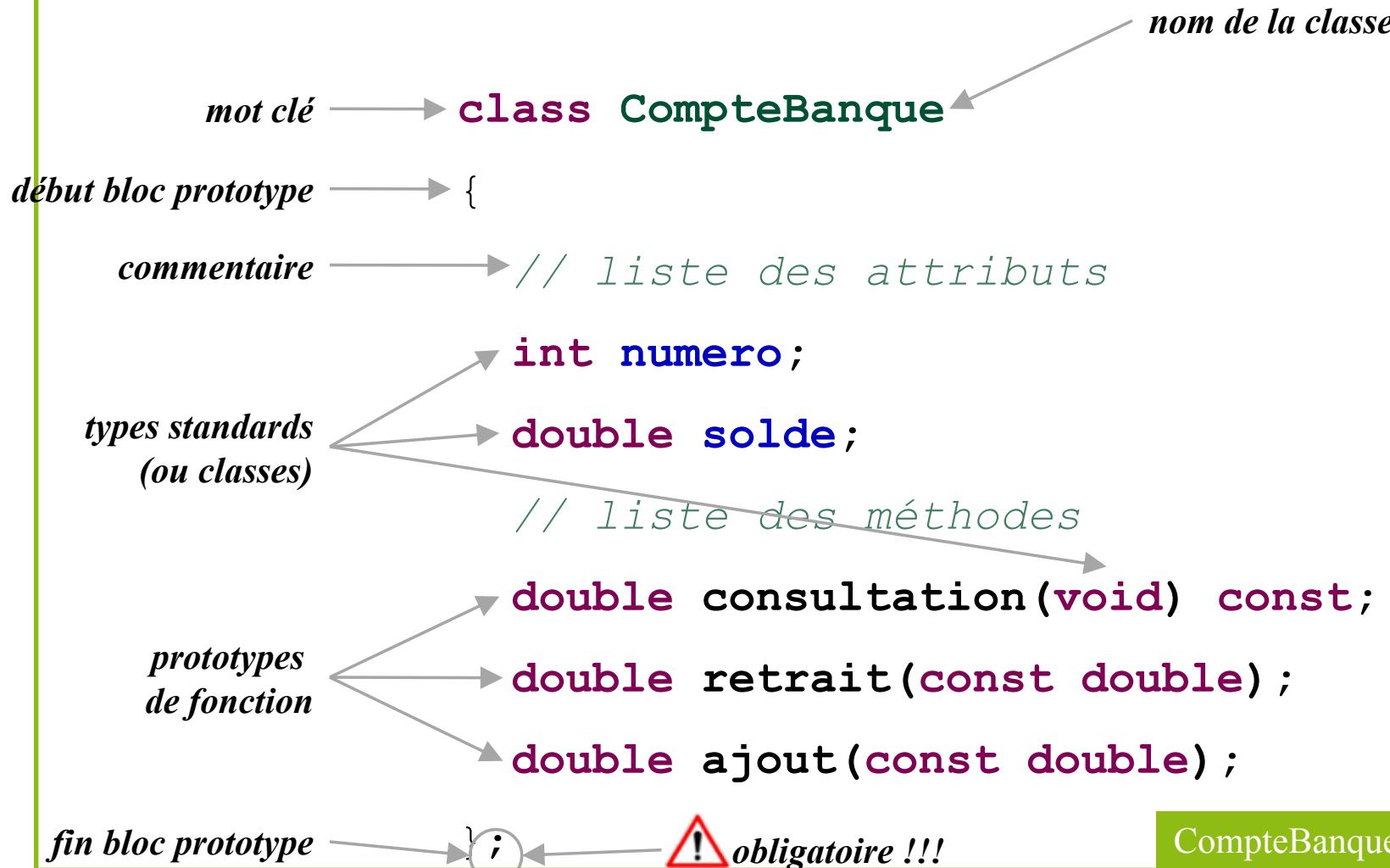
- Utilisé dans la sémantique de déplacement

Voir sémantique de déplacement page 142

## 2.3. Classe & instance

# Exemple - Déclaration

C++



# Exemple - *Implémentation*

C++

valeur de retour  
de la fonction

nom de la classe

opérateur de  
résolution de portée

nom de la  
méthode

```
double CompteBanque::consultation (void) const
{ return solde ; }
```

```
double CompteBanque::retrait (const double val)
{
 solde = solde - val ;
 return solde ;
}
```

```
double CompteBanque::ajout (const double val)
{ return solde += val ; }
// équivalent à : solde = solde + val ; return
solde ;
```

CompteBanque.cpp

# Exemple - *Instanciation*

C++

Déclaration d'une instance :

*nom de la classe  
(type)*

*nom de l'instance  
(variable)*

*déclaration et  
création d'une  
instance statique\**

```
CompteBanque monCompte;
```

Autres cas :

- Constantes : «const»

```
const Chaine VIDE ("");
```

- Tableaux : «[ ]»

```
Point matrice [10][100] ; // pas de Point[] en C/C++
```

- Pointeurs : «\*»

```
Liste * ptr ;
```

```
ptr = new Liste(...);
```



\* : *statique* ( $\neq$  static)  
par opposition à *dynamique*

*Déclaration instance*

*Création instance  
dynamique*

# Qualificateurs de types

C++

- « const »
  - Attributs, paramètres, variables : `const int intval;`
    - `intval` ne peut pas changer de valeur (permet l'optimisation par le compilateur)
    - Pointeurs
- Méthodes : `void maMéthode(T arg) const;`
  - Une telle méthode ne modifie pas l'état interne de l'objet qui la déclenche
- Autres qualificateurs : « volatile », « restrict »
  - « volatile » existe aussi en Java

| Déclaration                          | Description                                      |
|--------------------------------------|--------------------------------------------------|
| <code>const int * ptr;</code>        | Pointeur vers un <u>entier constant</u>          |
| <code>int * const ptr2;</code>       | Pointeur constant vers un entier                 |
| <code>const int * const ptr3;</code> | Pointeur constant vers un <u>entier constant</u> |

# Différents types d'instances (1/2)

C++

```
class MaClass {

 int a;

 MaClass(const int value);

 ~MaClass(void);

 int valeur(void) const;
};
```

MaClass.h

```
#include <iostream>
using namespace std;
#include "MaClass.h"

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;
}
```

MaClass.cpp

```
#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 copié dans la valeur de retour
 // dans la pile et peut donc
 // être transmis à un autre pointeur
}
```

TestMaClass.cpp

# 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, la variable c 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.
 }

 // Instance dynamique
 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 (sauf à la fin du programme)
 delete pa;

 ...

 return 0;
}
```

TestMaClass.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.soldé
CompteBanque.retrait(23) } Faux ! (CompteBanque est un type, pas une variable)
```

# Accès à soi même

C++

- 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
      - « this » est une prvalue : rvalue pour les pointeurs
    - en Java / C# : this est une référence à l'instance courante
      - En Java « this » est final.
  - SmallTalk : mot clé « self »
- Utilité : résolution de conflits de noms dans les méthodes

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

## 2.4. Accessibilité

# Choix du mode d'accès

Il faut classer les membres :

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

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

Encapsulation forte

C++

# Fonctions amies : « friend »

C++

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

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

ostream & CompteBanque::operator <<(ostream & out, const CompteBanque & cpt)
{
 out << "Compte n° " << cpt.numero << ... << endl;
 return out;
}

Compte c(...);
cout << c;
```

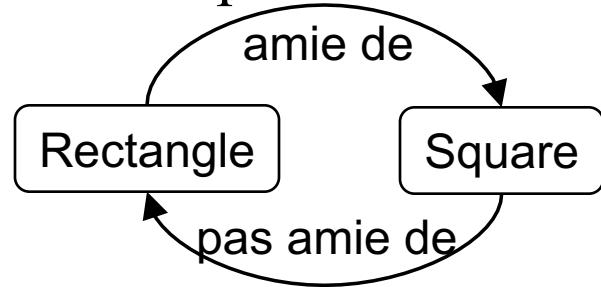
operator << n'est **pas** membre

Accès à l'attribut protégé « numero »

# Classes amies : « friend »

C++

- 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 s);
};
```

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

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

void Rectangle::convert(Square s) {
 width = s.side;
 height = s.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.  
⚠ techniquement possible mais avantageusement remplacée par des constructeurs
  - Lors de la déclaration des attributs

→ Nouvelle définition de la classe CompteBanque :

```
class CompteBanque
{
public :
 double consultation (void) const;
 double retrait (const double);
 double ajout (const double);
 void init (const int, const double);
protected :
 int numero = 0;
 double solde = 0.0;
};

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

C++ 11

Possible  
mais pas  
recommandé

Constructeurs

## 2.5. Constructeurs / Destructeur

# Constructeurs (1/3)

*Rôle* : membres prédéfinis pour l'initialisation des instances

*Nom* : celui de la classe

*Surchargeable*, comme toute méthode (plusieurs définitions possibles)

*Exemple* :

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

C++

CompteBanque.h

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

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

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

CompteBanque.cpp



Attention :  
« bad practice » en C++  
→ Utiliser les Listes  
d'initialisation

# Constructeurs (2/3)

- Vocabulaire
  - Constructeur **par défaut** : constructeur sans arguments initialisant les attributs avec des valeurs par défaut.
    - On verra quelques subtilités supplémentaires à ce sujet.
  - Constructeur **valué** : constructeur possédant un ou plusieurs arguments destinés à initialiser tout ou partie des attributs.
  - Constructeur **de copie** : constructeur possédant un seul argument du type même de la classe et permettant de copier tout ou partie des attributs d'une instance dans une autre.
    - Variante en C++11 : Constructeur **de déplacement** : constructeur possédant un seul argument du type même de la classe et permettant de « transférer » tout ou partie des valeurs des attributs d'une instance (vouée à disparaître) dans une autre. Accroît les performances lorsque les objets détiennent des ressources importantes : évite la copie explicite de ces ressources.

Voir sémantique de déplacement page 142

# Constructeurs (3/3)

*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 ; ...

*appel de CompteBanque (const int, const double)*

*appel de CompteBanque (const int)*

*appel de CompteBanque (void)*



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

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

# Appels implicites aux constructeurs

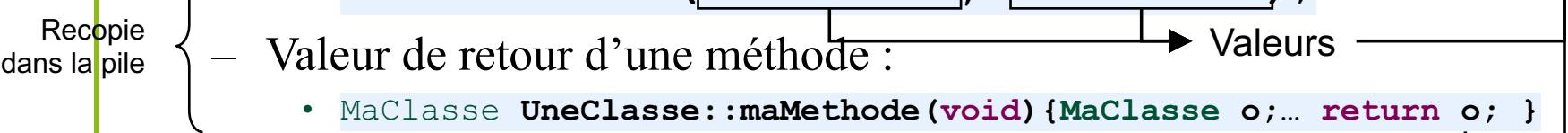
C++

- Que se passe t'il lors du passage d'un élément en argument d'une méthode ? Cela dépend du mode de passage de l'argument
  - Par valeur (Type) : La **valeur** de l'objet est recopiée dans la pile d'appel : Il y a donc un appel implicite au constructeur de copie.
  - Par adresse (Type \*) ou par référence (Type &) : l'**adresse** de l'objet est recopiée dans la pile d'appel.

- Appels implicites au constructeurs en C++ :

- Déclaration d'une instance statique : MaClasse o1;
- Déclaration suivie d'une affectation: MaClasse o2 = o1;
- Passage d'arguments : lors de passages d'arguments par valeurs :

• `int maMethode (MaClasse o, int valeur);`



- Valeur de retour d'une méthode :

• `MaClasse UneClasse::maMethode (void) {MaClasse o;... return o; }`

# Constructeur(s) par défaut (1/2)

- Lorsque aucun constructeur n'est défini :
  - en Java : Il  $\exists$  un constructeur par défaut (sans argument)
  - en C++ : Il  $\exists$ 
    - un ctor par défaut (sans argument),
    - un ctor de copie par défaut,
    - un ctor de déplacement par défaut,
    - un opérateur de copie par défaut.
    - un opérateur de déplacement par défaut
    - un dtor par défaut.
- S'il existe au moins un constructeur (Quel qu'il soit : avec ou sans arguments) :
  - en java et en C++: le(s) constructeur(s) par défaut fourni(s) automatiquement n'existe(nt) plus (il faut donc les écrire).

Voir transparent  
Membres par  
défaut ([page 170](#))

# Constructeur(s) par défaut (2/2)

## Constructeur sans arguments

Exemple :

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

C++

CompteBanque.h

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

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

CompteBanque.cpp

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 (const int n, const double val, const String & nom);
 protected :
 int numero ;
 String titulaire ;
 double solde ;
 } ;

CompteBanque::CompteBanque (const int n, const double val,
 const String & nom)
 : numero (n), titulaire (nom), solde (0)
{

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

Liste  
d'initialisation\*

Corps du  
constructeur

\* : dans l'ordre des déclarations des attributs

# Constructeur de copie

C++

Syntaxe : NomClasse (const NomClasse & objet);

Si ce constructeur n'est pas défini : il y a recopie bit à bit par le def. copy ctor.

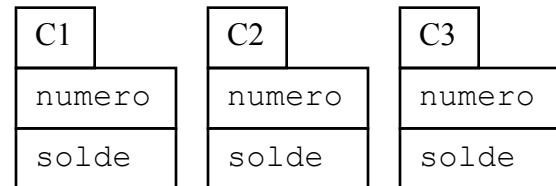
Deux types d'appels :

- 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  
(voir [« Appels implicites aux constructeurs »](#) page 136)

Exemple :

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

int main ()
{
 CompteBanque c1; // appel implicite constructeur par défaut
 CompteBanque c2(c1); // appel explicite constructeur de copie
 CompteBanque c3 = c1; // appel implicite constructeur copie
 // différent de CompteBanque c3;
 // suivi de c3 = c1;
 AfficheCompte(c1); // appel implicite, par recopie d'un argument
 // (passage par valeur ⇒ en gal bad practice)
```



# Gestion mémoire

C++

- Gestion explicite de la mémoire : C++
  - Création d'instances
    - Statiques : déclaration {  
MaClasse monObjet (...);
    - Dynamiques : **new** → MaClasse \* monObjet =  
**new** MaClasse (...);
  - Destruction d'instances : Destructeur
    - Statiques : en fin de bloc  
...  
**delete** monObjet;
    - Dynamiques :
      - Appel explicite à l'opérateur **delete** qui fait appel au **Destructeur**
- Gestion implicite de la mémoire : Langages Objets purs (Java, SmallTalk, C#)
  - Destruction automatique des instances non référencées par le Garbage Collector.

# Sémantique de déplacement

C++03

- Problème

```
MyObject o0;
```

Constructeur par défaut

```
MyObject o1(input);
```

Constructeur valué

```
MyObject o2(o1);
MyObject o3 = o1;
```

Constructeur de copie

```
MyObject o4 = MyObject(input);
```

Constructeur valué  
Constructeur de copie  
Destructeur

```
MyObject o5;
```

Rvalues

```
o5 = MyObject(input);
```

Constructeur valué  
Opérateur de copie  
Destructeur

- Comment éviter les créations / copie / destructions inutiles (surtout si celles-ci sont coûteuses en espace mémoire et/ou en temps d'exécution)

- Idée : « déplacer » les rvalues dans les lvalues,  
 $\Leftrightarrow$  les lvalues subtilisent les ressources contenues dans les rvalues lorsque cela est possible grâce à des opérations spécifiques.

# Sémantique de déplacement

C++11

- Solutions

- Elision de copie par le compilateur (copy elision)

- Lorsqu'un objet temporaire est censé être (copié | déplacé) vers un objet du même type (la copie | le déplacement) est omise : l'objet temporaire est alors directement créé dans l'espace mémoire censé le recueillir.

```
MyObject o4 = MyObject(input);
:
o5 = MyObject(input);
```

Constructeur valué

Constructeur valué  
Constructeur de déplacement  
Destructeur

Constructeur valué  
Opérateur de déplacement  
Destructeur

- Utilisation du constructeur de déplacement et de l'opérateur de déplacement (s'ils existent)

- Le déplacement est alors considéré comme une copie destructive (vol des ressources des rvalues, contrairement à la copie ordinaire qui laisse les rvalues dans leur état initial).

# Destructeur

C++

*Rôle :*

Fonction membre dédiée au «démontage» des instances

*Syntaxe :*

```
virtual ~NomClasse () ;
```

*Appel :*

Appel implicite automatique :

lorsque l'instance n'est plus définie

(sort de sa zone de portée, qui est le bloc dans lequel l'instance a été déclarée)

Appelé aussi pour chacun des attributs de la classe

Appel explicite avec le mot clé **delete** ... pour une instance créée dynamiquement (`delete []` ... pour les tableaux créés dynamiquement)

Non surchargeable (prototype figé)

# Destructeur - Exemple

C++

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

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

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

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

// résultat de l'exécution :
construction
appel de f
destruction
construction
destruction
```

Toto tin  
cout << "appel ... " } } f () ;  
} ← Destruction de tin  
Toto tex ;  
} ← Destruction de tex

# Construction / Destruction

C++

- Instances statiques
  - Crées lors de leur déclaration
  - Détruites à la fin du bloc dans lequel elles sont déclarées
- Instances dynamiquement allouées
  - Constructeurs et Destructeurs ne sont pas appelés pour les attributs de type pointeur<sup>1</sup>  
⇒ Nécessite un appel explicite avec les opérateurs **new** et **delete**.
- Tableaux :
  - Le constructeur par défaut est appelé implicitement pour chacun des éléments du tableau.
  - idem pour le destructeur lors de la destruction d'un tableau.

```
{
 Type t1(...);
 Type t2{...};
 Type t3 = ...;

 Type * pt = new Type(...);

 const size_t size = 5;
 Type tab1[size];
 Type tab2[size] = {t1, ...}

 Type * dyntab = new Type[size];

 ...

 delete pt;
 delete[] dyntab;
}
```

<sup>1</sup> : Plus exactement seuls les ctor/dtor des pointeurs sont appelés

# variable ≠ instance

C++

```
{
 A * pa1;
 {
 A a; // instance statique*
 A * pa2;

 pa1 = new A(); // instance dynamique
 pa2 = new A(); // instance dynamique
 /*...*/
 }

 /*
 * à la fin de ce bloc les variables a et *pa2 disparaissent.
 * Le destructeur est appelé pour a ce qui détruit l'objet et pour
 * pa2 ce qui détruit le pointeur mais pas l'objet pointé :
 * fuite mémoire
 */

 delete pa1;
 /* l'objet pointé par pa1 est détruit */
}

/* la variable *pa1 disparaît */
```

\* : statique ≠ static

## 2.6. Membres de classe

# 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 ;
 cout << "A::st = " << A::st << endl ; // affichage : A::st = 33 ;
 return 0 ;
}
```

## 2.7 Classes Imbriquées

- C++
  - Classes imbriquées seulement
  - Déclaration sans mot clé « static »

## 2.8. Polymorphisme

# Polymorphisme - *La surcharge*

- Prise en compte du nombre et du type d'argument pour distinguer les méthodes entre elles.
- 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 (const double v1, const double v2) ;
int moyenne (const int v1, const int v2) ;
double moyenne (const int v1, const int v2) ;
// Erreur de compilation
...
```

# Polymorphisme - *La surcharge*

C++

- Prise en compte du type et du nombre d'arguments.
- Pas de prise en compte du type de la valeur de retour pour l'identification de l'appel
- Prise en compte des qualificateurs de type const / volatile / mutable des arguments : const int & ≠ int & ≠ int &&
- Prise en compte du qualificateur des méthodes :
  - int foo(...) const ≠ int foo(...)
  - Typiquement utilisé dans les accesseurs / mutateurs en C++
    - int val() const { return \_val; } : accesseur en lecture seule de la propriété « val » : retourne une **copie de la valeur** de \_val.
    - int & val () { return \_val; } : accesseur en lecture/écriture de la propriété « val » : retourne une **référence** vers \_val.

# Polymorphisme - Conversion de types

Attention aux conversions de types implicites

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

```
double moyenne (const double v1, const double v2) ;
int moyenne (const int v1, const int 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 (const double v1,
 // const double v2) ;
 moyenne(c, c); // Appel de int moyenne (const int v1, const int 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 ?
  - C++
    - C++ 03 : Impossible
    - C++ 11+ : grâce aux constructeurs délégués (utilisation d'un constructeur dans un autre constructeur) : dans la liste d'initialisation uniquement.
      - Compte(const string & titulaire) : Compte() { ... } ←
  - Java
    - avec le mot clé `this`
    - Compte(String titulaire) { **this()**; ... } ←
  - Smalltalk: Il n'y a pas vraiment de constructeurs, seulement la réimplémentation de la méthode `new`.

Appel du constructeur par défaut dans un constructeur valued

# Polymorphisme - Les opérateurs

C++

Fonctions et opérateurs (membres ou pas) sont surchargeables

Surcharge d'opérateurs :

- L'arité et la priorité sont imposés

*Exemples :*

opérateur ~ : arité 1 et priorité 16

opérateur / : arité 2 et priorité 13

- Se définit comme une fonction dont le nom est :

*TRetour operator nomOperateur (liste arguments)*

*Exemple d'implémentation d'un opérateur membre d'une classe :*

```
NomClasse & NomClasse::operator + (type mode arg) { ... }
```

- L'argument gauche (le premier) doit toujours être une instance :

```
A a(3);
a + 2; // <==> a.operator+(2);
```

- Pour un opérateur membre d'une classe, il s'agit de l'instance courante, qui est alors implicite

# Polymorphisme - Les opérateurs

C++

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

Compte & Compte::operator +(const double v) {
 solde += v;
 return *this;
}

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

int main() {
 Compte cpt;

 cpt + 10; /*<==>*/ cpt.operator+(10);
 cpt - 2 + 10; /*<==>*/ (cpt.operator-(2)).operator+(10);
 cpt + 2 - 10; /*<==>*/ (cpt.operator+(2)).operator-(10);
 cpt + 2 + 10; /*<==>*/ (cpt.operator+(2)).operator+(10);

 return 0;
}
```

The diagram illustrates the implementation of operator overloading for the `Compte` class. It shows the class definition with its public and protected members, and the implementation of the `+ (const double)` and `- (const double)` operators. In the `main` function, four expressions are evaluated: `cpt + 10`, `cpt - 2 + 10`, `cpt + 2 - 10`, and `cpt + 2 + 10`. Annotations explain the return types and the flow of the operations:

- Renvoie une référence vers lui-même : Compte**: This annotation points to the `operator +` and `operator -` definitions, indicating that they return a reference to the object itself.
- Renvoie la valeur du solde : double**: This annotation points to the `solde` variable and the return statements in the `operator +` and `operator -` implementations.

On the right side of the diagram, there are two boxes labeled `Compte` and `double`, which likely represent the types involved in the operations.

# Polymorphisme - Les opérateurs

C++

- Opérateurs internes (membres) vs opérateurs externes

```
class Compte
{
 ...
 // Opérateur interne de comparaison : méthode
 bool operator ==(const Compte & c) const;
 ...
 // Opérateur externe (ami) de comparaison : fonction
 friend bool operator ==(const Compte & c1, const Compte & c2);
 ...
};
```

The diagram illustrates the relationship between internal and external operators. A central box labeled "XOR" has two arrows pointing to the respective operator definitions: one to the internal operator `operator ==(const Compte & c) const;` and another to the external operator `friend bool operator ==(const Compte & c1, const Compte & c2);`.

- Les opérateurs externes sont souvent déclarés amis « friend » dans la classe afin qu'ils puissent accéder au contenu des classes qu'ils manipulent:
  - `friend ostream & operator <<(ostream & out, const Compte & c);`

# Polymorphisme - Les opérateurs

C++

| Priorité | Opérateur          | Description                                                        | Associativité   | usage                                                                    | Surcharge membre / non membre (global)                                                           | Note                                                  |
|----------|--------------------|--------------------------------------------------------------------|-----------------|--------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------|-------------------------------------------------------|
| 18       | ::                 | Opérateur de résolution de portée                                  | gauche à droite | NameSpace::Classe::membre                                                |                                                                                                  |                                                       |
| 17       | ()                 | Parenthèses                                                        | gauche à droite | instance(...)                                                            | T & MaClasse::operator() (const unsigned int,...)                                                |                                                       |
|          | []                 | Tableau (subscript)                                                |                 | instance[...]                                                            | T & MaClasse::operator[](unsigned i)                                                             |                                                       |
|          | .                  | Sélection d'un membre par un identificateur (structures et objets) |                 | instance.membre                                                          |                                                                                                  |                                                       |
|          | ->                 | Sélection d'un membre par un pointeur (structures et objets)       |                 | pointeur->membre                                                         | T* MaClasse::operator->(void)                                                                    | smart pointers                                        |
|          | ++ --              | Incrémantation post-fixée (1)                                      |                 | instance++                                                               | void MaClasse::operator++(int) ← dummy int                                                       |                                                       |
| 16       | ++ --              | Incrémantation pré-fixée                                           | droite à gauche | ++instance                                                               | void MaClasse::operator++(void)                                                                  |                                                       |
|          | + -                | Opérateurs + et - unaires                                          |                 | -instance                                                                | MaClasse & MaClasse::operator-(void)<br>friend MaClasse operator-(const MaClasse &);             |                                                       |
|          | ! ~                | Opérateurs unaires négation logique et bit à bit                   |                 | !instance                                                                | MaClasse & MaClasse::operator!(void) const;<br>friend MaClasse operator! (const MaClasse &);     |                                                       |
|          | *                  | Déréfancement                                                      |                 | *instance                                                                | T& MaClasse::operator* (void)                                                                    | smart pointers                                        |
|          | &                  | Adresse                                                            |                 | &instance                                                                | T* MaClasse::operator&(void);<br>friend MaClasse * operator&(MaClasse &);                        | smart pointers                                        |
|          | sizeof             | Opérateur de taille (d'objet ou de type)                           |                 | sizeof(Type ou instance)                                                 |                                                                                                  |                                                       |
|          | new<br>new[]       | Allocation mémoire et mémoire tableau                              |                 | Type * ptr = new(user args)<br>Type(...)<br>Type[] tab = new[nbElements] | void *MaClasse::operator new(size_t s, user args );<br>void* MaClasse::operator new[ ] (size_t); | implicitement déclarés en tant que méthodes de classe |
|          | delete<br>delete[] | Libération mémoire et mémoire tableau                              |                 | Type * ptr; ... ; delete ptr;<br>Type[] ptr; ... ; delete[] ptr;         | void MaClasse::operator delete (void *);<br>void MaClasse::operator delete[ ] (void*);           |                                                       |
| 15       | (type)             | Cast                                                               | droite à gauche | (Type)instanceDeMaClasse                                                 | MaClasse::operator Type(void)                                                                    | plusieurs                                             |
| 14       | .*                 | Déréfancement d'un pointeur de membre d'un objet                   | gauche à droite | instance.*membre                                                         |                                                                                                  |                                                       |
|          | ->*                | Déréfancement d'un pointeur de membre d'un objet pointé            |                 | (pointeur->*membre)()                                                    |                                                                                                  | pointeur de fonction membre                           |

# Polymorphisme - Les opérateurs

C++

| Priorité | Opérateur | Description                                                            | Associativité   | usage                                                | Surcharge membre / non membre (global)                                                                                                                            | Note   |
|----------|-----------|------------------------------------------------------------------------|-----------------|------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------|
| 13       | * / %     | Multiplication, Division et Modulo                                     | gauche à droite | instance1 * instance 2                               | MaClasse & MaClasse::operator*(const MaClasse &) const                                                                                                            |        |
| 12       | + -       | Addition, Soustraction                                                 | gauche à droite | instance1 + instance 2                               | MaClasse & MaClasse::operator+(const MaClasse &) const                                                                                                            |        |
| 11       | << >>     | Décalages de bits à gauche ou à droite                                 | gauche à droite | instance << 1<br>cout << Instance<br>cin >> instance | void MaClasse::operator<<(unsigned)<br>friend ostream & operator << >> (ostream &, const MaClasse &);<br>friend istream & operator >> >> (istream &, MaClasse &); |        |
| 10       | < <=      | Opérateurs relationnels "strictement inférieur" et "inférieur ou égal" |                 | instance1 < instance2                                | bool MaClasse::operator<(const MaClasse &) const                                                                                                                  |        |
|          | > >=      | Opérateurs relationnels "strictement supérieur" et "supérieur ou égal" |                 | instance1 > instance2                                | bool MaClasse::operator>(const MaClasse &) const                                                                                                                  |        |
| 9        | == !=     | Opérateurs relationnels "égal" et "différent de"                       | gauche à droite | instance1 == instance2                               | bool MaClasse::operator==(const MaClasse &) const                                                                                                                 |        |
| 8        | &         | ET bit à bit                                                           | gauche à droite | instance1 & instance2                                | MaClasse & MaClasse::operator&(const MaClasse &) const                                                                                                            |        |
| 7        | ^         | OU exclusif bit à bit                                                  | gauche à droite | instance1 ^ instance2                                | "                                                                                                                                                                 |        |
| 6        |           | OU bit à bit                                                           | gauche à droite | instance1   instance2                                | "                                                                                                                                                                 |        |
| 5        | &&        | ET logique                                                             | gauche à droite | instance1 && instance2                               | bool MaClasse::operator&&(const MaClasse &) const                                                                                                                 |        |
| 4        |           | OU logique                                                             | gauche à droite | instance1    instance2                               | "                                                                                                                                                                 |        |
| 3        | c?t:f     | Opérateur de condition ternaire                                        | droite à gauche | (expression)?instrTrue:instFalse                     |                                                                                                                                                                   |        |
| 2        | =         | Affectation                                                            | droite à gauche | instance1 = instance2                                | MaClasse & operator = (const MaClasse & m)                                                                                                                        |        |
|          | += -=     | Affectation avec somme ou soustraction                                 |                 | instance += valeur<br>instance1 += instance2         | void operator+=(const Type &)<br>MaClasse & operator+=(const MaClasse &)                                                                                          |        |
|          | *= /= %=  | Affectation avec multiplication, division ou modulo                    |                 | instance *= valeur ou instance                       | "                                                                                                                                                                 |        |
|          | <=>=      | Affectation avec décalages de bits                                     |                 | instance <=> valeur ou instance                      | "                                                                                                                                                                 |        |
|          | &= ^=  =  | Affectation avec ET, OU exclusif ou OU inclusif bit à bit              |                 | instance &= valeur ou instance                       | "                                                                                                                                                                 |        |
| 1        | ,         | Séquence d'expressions                                                 | gauche à droite | e=a, b, c, d ⇔ (((e=a),b),c),d                       | MaClasse & operator,(const Maclasse &)                                                                                                                            | concat |

# Polymorphisme - Valeurs par défaut

C++

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

```
class Forme
{
private:
 int x_pos;
 int y_pos;
 int color;
public:
 Forme (int x, int y, int c = 1);
 void deplace(int dx = 0, int dy = 0);
};

Forme::Forme(int x, int y, int c)
: x_pos(x), y_pos(y), color(c)
{}

void Forme::deplace(int dx, int dy)
{x_pos+=dx; y_pos+=dy;}
```

Définition

Valeurs par défaut

```
int main(void)
{
 Forme maForme2(10,10);
 Forme maForme3(10,10,2);
 maForme2.deplace();
 maForme2.deplace(3);
 maForme3.deplace(3, 4);

 return 0;
}
```

Utilisation

# Polymorphisme – liste d'arguments variable

C/C++

- Nombre variable d'arguments de même type dans une méthode ou fonction :

```
void vaTest(const string & msg, const int nbArgs, ...)
{
 cout << msg << " (" << nbArgs << "): ";
 va_list args;
 va_start(args, nbArgs);
 for (int i = 0; i < nbArgs; i++)
 cout << va_arg(args, int) << " ";
 va_end(args);
 cout << endl;
}

int main()
{
 vaTest("One vararg", 1, 10);
 vaTest("Three varargs", 3, 1, 2, 3);
 vaTest("No varargs", 0);

 return EXIT_SUCCESS;
}
```



- Pas pratique
- Sujet à erreurs de typage

et pourtant ...

```
int printf(const char * _fmt, ...);
```

# Polymorphisme – liste d'arguments variable

C++11

- Nombre variable d'arguments [de même type de types quelconques] dans une méthode ou une fonction : templates variadiques

```
void vaTest() { cout << endl; }
```

```
template <class T, class ...R>
```

```
void vaTest(const T & head, R... tail)
```

```
{
```

```
 cout << head << " ";
```

Quand tail... n'est pas vide

```
 vaTest(tail...);
```

```
}
```

Quand tail... est vide

```
template<class ...T>
```

```
void vaTest(const string & msg, T... args)
```

```
{
```

```
 cout << msg << ": ";
```

```
 vaTest(args...);
```

```
}
```

int main()

```
{
```

```
 vaTest("One vararg", 10);
```

```
 vaTest("Three varargs", 1, 2, 3);
```

```
 vaTest("No varargs");
```

```
}
```

```
return EXIT_SUCCESS;
```

Voir Variadic Templates  
page 188

sortie

One vararg: 10

Three varargs: 1 2 3

No varargs:

Function parameter pack

Pack expansion

## 2.9. L'Héritage

# Héritage à carte

C++

Restriction d'accès uniquement

Héritage global de l'accessibilité :

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

*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 publics : accès externe

**Protected** :

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

**Private** : (mode par défaut)

Tous les membres hérités deviennent privés

| Héritage | Public       | Protégé      | Privé        |
|----------|--------------|--------------|--------------|
| Membre   |              |              |              |
| Public   | Public       | Protégé      | Privé        |
| Protégé  | Protégé      | Protégé      | Privé        |
| Privé    | Inaccessible | Inaccessible | Inaccessible |

# Héritage à la carte - Exemple

C++

Héritage à la carte :

Seuls certains membres sont hérités sans changements

*Exemples :*

```
→ class CompteBloque1 : protected Compte
{
 public:
 Compte::consultation; Compte::ajout; /* ... */
}; // seul retrait est modifié et devient protégé
```

Mode pour la classe

Mode spécifiques pour ces membres

```
→ class CompteBloque2: private Compte
{
 public:
 Compte::consultation; Compte::ajout; /* ... */
 protected:
 Compte::solde;
}; // seul retrait est modifié et devient privé
```

Mode pour la classe

Modes spécifiques pour ces membres

'Compte' is an inaccessible base of



# Surcharge lors de l'héritage

C++

- Recouvrement et/ou remplacement
  - C++ : La redéfinition recouvre toutes les surcharges héritées

```
class Compte
{
public :
 void ouvrir (void) ;
 void ouvrir (const double); // surcharge OK
protected :
 double solde;
};

class CompteToto : public Compte
{
public :
 void ouvrir (const double, const double) ;
 // redéfinition : remplace les 2 formes héritées !
protected :
 double decouvert ;
};

void CompteToto::ouvrir (const double dec,
 const double sol)
{
 Compte::ouvrir(sol); // OK
 ouvrir(sol); // Erreur de compilation : non défini
 ouvrir(sol,dec); // OK nouvelle forme
}
```

↑  Appel récursif

- Java : remplacement uniquement des méthodes possédant des signatures\* identiques. \* : nom + nombre & types des arguments

# Hiérarchie des constructeurs

C++

```
class A
{
 private :
 int a;
 public :
 A(int val);
};

class B : public A
{
 private :
 int b;
 public :
 B();
 B(int val);
};

A::A(int val) : a(val)
{
 cout << "Creation A("
 << val << ")" << endl;
}

B::B() : b(0) // A(1), b(0)
{
 cout << "Creation B()" << endl;
}

B::B(int val) : A(2), b(val)
{
 cout << "Creation B("
 << val << ")" << endl;
}
```

```
int main(void)
{
 A a(1);
 B b(3);

 return 0;
}
```

Appel implicite du constructeur par défaut  
**A()** inexistant : Erreur !

// Correction

Appel explicite du constructeur  
**A(int val)**

# Constructeur(s) par défaut (3/3)

C++11

- Rappel : S'il existe au moins un constructeur (Quel qu'il soit : avec ou sans arguments), les constructeurs par défaut fournis automatiquement n'existent plus.
- On peut toutefois réutiliser explicitement un constructeur par défaut grâce au mot clé = **default** ;

```
class C
{
private:
 int a;
public:
 // Constructeur valué ==> les autres constructeurs par défaut n'existent plus
 C(int a) : a(a) {}

 C() = default; // Réutilisation du ctor par défaut (! warning a non init.)

 C(const C & c) = default; // Réutilisation du ctor de copie par défaut

 C(C && c) = default; // Réutilisation du ctor de déplacement par défaut
};
```

Voir sémantique de déplacement page 142

# Membres par défaut

C++11

- Membres par défaut (générés automatiquement si absents\*)
  - Constructeur par défaut : `MaClasse()`
  - Constructeur de copie : `MaClasse(const MaClasse & mc)`
  - Constructeur de déplacement : `MaClasse(MaClasse && mc)`
  - Opérateur de copie : `MaClasse & operator =(const MaClasse & mc)`
  - Opérateur de déplacement : `MaClasse & operator =(MaClasse && mc)`
  - Destructeur : `virtual ~MaClasse()`
- \*: Règles (X = not generated, O = Obsolete)  $\Leftrightarrow$  à déclarer explicitement

|                | Default Ctor | Default Copy Ctor | Default Move Ctor | Default Copy Operator | Default Move Operator | Default Dtor |
|----------------|--------------|-------------------|-------------------|-----------------------|-----------------------|--------------|
| Any Ctor       | X            | X                 | X                 |                       |                       |              |
| Copy Ctor      |              | X                 | X                 | O                     | X                     |              |
| Move Ctor      |              | X                 | X                 | X                     | X                     |              |
| Copy Operator  |              | O                 | X                 | X                     | X                     |              |
| Move Operator  |              | X                 | X                 | X                     | X                     |              |
| [Virtual] Dtor |              | O                 | X                 | O                     | X                     | X            |

# Contrôle des membres par défaut

C++11

- Defaulted member : = **default**
  - Réutilisation explicite d'un membre par défaut lorsque celui ci n'est plus généré automatiquement (voir les règles précédentes)
- Deleted member : = **delete**
  - Suppression explicite de la génération automatique d'un tel membre

```
class C
{
private:
 int a;
public:
 C(int a) : a(a) {}
 C() = default;
 C(const C & c) = default;
 C(C && c) = delete;
 C& operator =(const C & c) = default;
};
```

```
int main(...)
{
 C c0; // Default ctor : Ok
 C c1(3); // Valued ctor : Ok
 C c2 = c1; // Default copy ctor : Ok
 c0 = c1; // Default copy op. : Ok
 C e3 = C(4); // No Move ctor : Ko

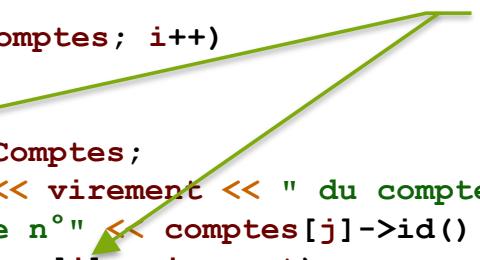
 return EXIT_SUCCESS;
}
```

# Polymorphisme d'héritage

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

```
void affiche(const Compte & c) {...}
...
const size_t nbComptes = 4;
Compte * comptes[nbComptes];
comptes[0] = new Compte(100);
comptes[1] = new CompteActions(200);
comptes[2] = new Compte(300);
comptes[3] = new CompteBloque(400);
Action action(453, 10.5);
static_cast<CompteActions *>(comptes[1])->achatAction(&action, 15);
double virement = 50.0;

for (size_t i = 0; i < nbComptes; i++)
{
 affiche(*comptes[i]);
 size_t j = (i + 1) % nbComptes;
 cout << "virement de " << virement << " du compte n° " << comptes[i]->id()
 << " vers le compte n°" << comptes[j]->id() << endl;
 comptes[i]->virer(*comptes[j], virement);
}
```



Conversions implicites en Compte

# Lien dynamique – Exemples (1/3)

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(const char *);
}

void Compte::printInfo(
 const char * 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 » ?

# Lien dynamique – Exemples (2/3)

C++

```
class A {
protected:
 int a;
public:
 A(const int value);
 virtual string toString(void) const;
 friend ostream & operator <<(ostream
 & out, const A & aInstance);
};
A::A(const int value) : a(value) {
 cout << "Construction de A : "
 << *this << endl;
}
string A::toString(void) const {
 ostringstream oss;
 oss << "A::a = " << a;
 return oss.str();
}
ostream & operator << (ostream & out,
 const A & aInstance) {
 out << aInstance.toString();
 return out;
}
```

A.h|cpp

```
int main(void) {
 ①A aInstance(1);
 ②B bInstance(2,3);
 ③cout << "instance de A : "
 << aInstance << endl;
 ④cout << "instance de B : "
 << bInstance << endl;
 return 0;
}
```

TestAB.cpp

```
class B : public A {
protected:
 int b;
public:
 B(const int value1, const int value2);
 // redéfinition de toString pour B
 string toString(void) const;
};
B::B(const int value1, const int value2)
: A(value1), b(value2) {
 cout << "Construction de B : "
 << *this << endl;
}

string B::toString(void) const {
 ostringstream oss;
 oss << "B::a = " << a
 << ", B::b = " << b;
 return oss.str();
}
```

B.h|cpp

résultats

- ① Construction de A : A::a = 1
- ② Construction de A : A::a = 2
- Construction de B : B::a = 2, B::b = 3
- ③ instance de A : A::a = 1
- ④ instance de B : B::a = 2, B::b = 3

# Lien dynamique – Exemples (3/3)

C++

1. A aInstance(1);
  - A::A(int) ⇒ operator <<() ⇒ A::toString()
    - Construction de A : A::a = 1
2. B bInstance(2, 3);
  - A::A(int) ⇒ operator <<() ⇒ A::toString()
  - Construction de A : A::a = 2
  - B::B(int, int) ⇒ operator <<() ⇒ B::toString()
  - Construction de B : B::a = 2, B::b= 3
3. cout << "instance de A : " << aInstance << endl;
  - operator <<(...) ⇒ A::toString()
  - instance de A : A::a = 1
4. cout << "instance de B : " << bInstance << endl;
  - operator <<(...) ⇒ B::toString()
  - instance de B : B::a = 2, B::b= 3



Pas de lien dynamique dans les constructeurs

# Lien Dynamique – Destructeurs virtuels

C++

```
#include <iostream>
using namespace std;

class Base {
public:
 Base() { cout << "Constructor: Base" << endl; }
 virtual ~Base() { cout << "Destructor : Base" << endl; }
};
```

```
class Derivee : public Base {
 // Spécialisation de la classe de base
public:
 Derivee() { cout << "Constructor: Derivee" << endl; }
 virtual ~Derivee() { cout << "Destructor : Derivee" << endl; }
};
```

↑  
Optionnel si pas d'héritières

```
int main(void) {
 Base *var = new Derivee();
 delete var;
 return 0;
}
```

Exécution avec destructeurs virtuels

Constructor: Base  
Constructor: Derivee  
Destructor : Derivee  
Destructor : Base

Exécution avec destructeurs non-virtuels

Constructor: Base  
Constructor: Derivee  
Destructor : Base

# L'héritage multiple (1/2)

C++

- Déclaration : Liste de plusieurs classes mères
- Construction : Liste des différents constructeurs à appeler pour chaque classe
- Tous les destructeurs sont appelés
- Problèmes de conflits :
  1. 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)

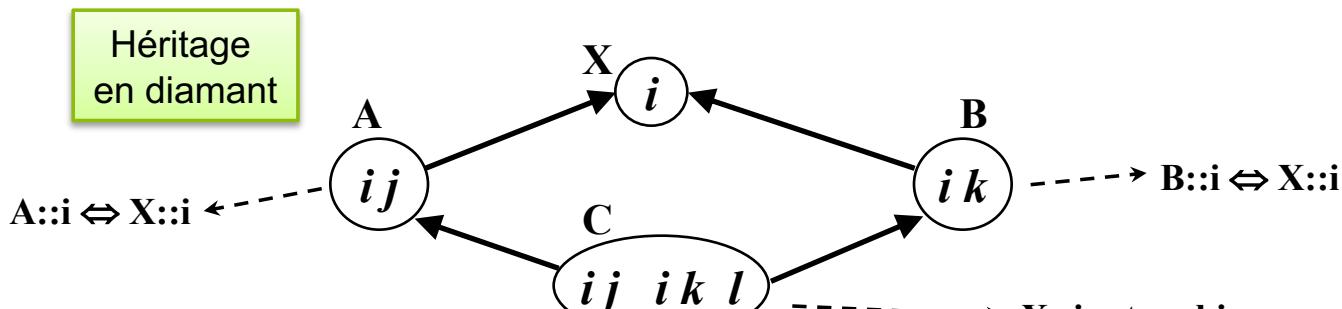
C++

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

## 2.10 L'introspection

# Introspection en C++

C++

- Utilisation du RTTI  
(Run-Time Type Information : #include <typeinfo>)

- Opérateur de cast dynamique\* (\*: à l'exécution):

- `Pile * poPtr = dynamic_cast<Pile *>(obj);`  
– pointeur `poPtr` NULL si `obj` n'est pas un `Pile *`
    - `Pile & poRef = dynamic_cast<Pile &>(obj);`  
– `bad_typeid` exception si `obj` n'est pas un `Pile &`

- Opérateur d'identification de type :

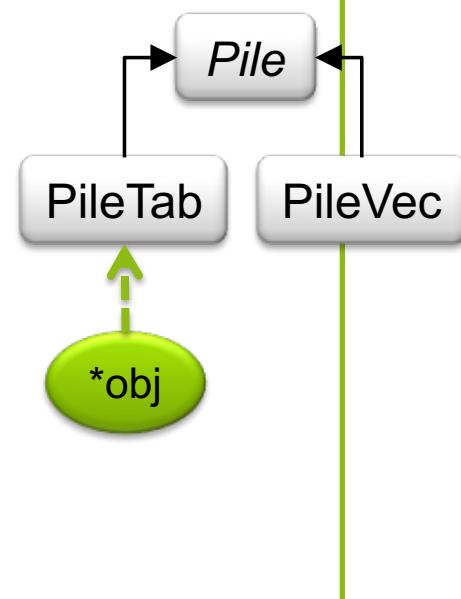
`Pile * obj = new PileTab(...)`

`type_info typeid(type ou instance)`

- `typeid(Pile) == typeid(*obj)` true
    - `typeid(PileTab) == typeid(*obj)` true
    - `typeid(PileVec) == typeid(*obj)` false

- `<type_traits>` (C++ 11)

- Différentes structures : `is_abstract<T>`, `is_array<T>`, `is_arithmetic ...`



## 2.11 Classes Abstraites

# Méthodes pures

C++

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

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

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

double CompteToto::vire (Compte & c, const double v)
{ solde -= 5 ; return Compte::vire (c, v) ; } // Erreur à l'édition de liens

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

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

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

↑ Méthode Purement virtuelle

# Lambda Expressions

C++11

- Nécessité
    - C++ : Remplacer\* les pointeurs de fonctions ou de méthodes ou plus généralement les foncteurs par des expressions plus simples.
  - Usage
    - Les lambda expressions peuvent être considérées comme des « fonctions anonymes et locales » pouvant être utilisées là où l'on en a besoin
      - Elles peuvent donc se substituer à des fonctions (en C) ou à des foncteurs (en C++) dans d'autres expressions sans avoir à déclarer explicitement ces fonctions ou foncteurs.
- \* : Sans toutefois exclure

# Lambda Expression

C++11

- Syntaxe
  - `[capture] (arguments) -> return_type {corps}`
  - `capture`: spécification des éléments de l'environnement capturés dans la lambda expression (et utilisables dans corps donc).
    - `[]` : Rien n'est capturé
    - `[&]` : Capture des variables de l'environnement par référence
    - `[=]` : Capture des variables de l'environnement par copie
    - `[=, &foo]` : Capture des variables par copie, sauf foo par référence
    - `[bar]` : Capture de la variable bar par copie, rien d'autre n'est capturé.
    - `[this]` : Capture du pointeur this de la classe englobante.
  - `arguments` : Pas de parenthèses si aucun argument.
  - `-> return_type` : Optionnel s'il peut être déduit par le compilateur.
  - `{corps}` : corps de l'expression lambda.
- Type
  - Quel est le type d'une expression lambda ? (Si l'on veut la référencer par une variable)
    - Avec capture : `template <class Ret, class... Args> class function<Ret(Args...)>;` (voir templates variadiques page 188),
    - Sans capture : `function<Ret(Args...)>` ou n'importe quel pointeur de fonction équivalent (`Ret(*)(<liste des arguments>)`).

# Lambda Expressions : exemples

C++11

```
string message("Hello World");
// omission des parenthèses quand pas d'arguments et capture par ref
auto f0 = [&message]{cout << message << endl;};
f0(); // Appel de la fonction
// expression complète
auto f1 = [](int x, int y) -> int{return x + y;};
// type de retour implicite, inféré de par la clause return
auto f2 = [](float x, float y){return x + y;};
// pas de clause return, type de retour = void
auto f3 = [](int & x){++x;};
// accès à une variable globale (bèèèh)
auto f4 = [](){++global_x;};
```

```
vector<float> operate(const vector<float> & v1, const vector<float> & v2,
 const function<float(float, float)> & func)
{
 ...func(..., ...);
}
```

```
vector<float> v1 = {1.1f, 2.1f, 3.1f, 4.1f, 5.1f};
vector<float> v2 = {5.2f, 4.2f, 3.2f, 2.2f, 1.2f, 0.2f};
vector<float> v3 = operate(v1, v2, atan2f);
vector<float> v4 = operate(v1, v2, f2);
vector<float> v5 = operate(v1, v2, [](float x, float y) -> float {return x + y;});
```

## 2.13 Généricité

# Généricité - Patrons de classe

C++

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

## 1. Patrons de fonctions :

Les types génériques 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) { ... }
```

utilisation des mots clés **class** ou **typename**  
pour définir les paramètres de type

## 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

# Généricité - Patrons Variadiques

C++ 11

- Définition : template de classe ou de fonction dont tout ou partie de la liste des paramètres de types est variable

- Exemple Fonction / Méthode :

```
template <typename T, typename ...R>
void Tableau<T>::addElement(const T & first, const R &... others)
{
 addElement(first);
 addElement(others...); // Appel récursif
}
```

- Exemple Classe : `template <class ... _Tp> class tuple`

- `tuple<int, string, Compte>`
  - `tuple<bool, int>`

- Introductions de nouveaux opérateurs

- opérateur ...

- à gauche d'un nom d'argument : désigne un « paquet de paramètres » (parameter pack) : permet de grouper [0..n) arguments

- à droite d'un nom d'argument : sépare les paramètres (pack expansion)

- opérateur `sizeof...(<parameter pack name>)`

- permet d'obtenir la taille d'un paquet de paramètres

# Patrons de classe – Exemple (1/6)

C++ 11

```
template<typename T> class // ou bien template<class T>
{
public:
 // Constructeur valué avec valeurs par défaut ==> default ctor
 Tableau(T * data = nullptr, const size_t size = 0);
 // Constructeur de copie
 Tableau(const Tableau<T> & tab);
 // Constructeur de déplacement
 Tableau(Tableau<T> && tab);
 // Constructeur variadième à partir d'éléments
 template <typename ...R> Tableau(const T & first, const R &... others);
 // Constructeur à partir d'une liste {elt1, elt2, ...}
 Tableau(initializer_list<T> list);
 // Destructeur
 virtual ~Tableau();
 // -- ACCESSEURS / OPERATEURS -----
 size_t size() const;
 // opérateur de copie et déplacement
 Tableau<T> & operator =(const Tableau<T> & tab);
 Tableau<T> & operator =(Tableau<T> && tab);
 // Opérateurs d'accès en lecture seule, puis écriture
 T operator [](const size_t index) const throw (TabException); // lecture seule
 T & operator [](const size_t index) throw (TabException); // lecture/écriture
 // opérateur de cast en T*
 operator T*() const;
 // opérateur ami de sortie standard
 friend ostream & operator << <>(ostream & out, const Tableau<T> & tab);
private:
 size_t size = 0;
 T * _data = nullptr;
 static void newHandler() throw (TabException);
 template <typename ...R> void addElement(const T & first, const R& ... others);
 void addElement(const T & element);
};
```

Template parameter pack

Method parameter pack

2 formes, l'une const et l'autre pas

Déclaration opérateur templatisé interne ami

Tableaux.h

# Patrons de classe – Exemple (2/6)

C++ 11

```
// Constructeur valué
template <typename T> Tableau<T>::Tableau(T * data, const size_t size) :
 _size(size), _data(size > 0 ? new T[size] : nullptr)
{
 set_new_handler(newHandler);
 for (size_t i = 0; i < _size; i++)
 {
 _data[i] = data[i];
 }
}

// Constructeur de copie
template <typename T> Tableau<T>::Tableau(const Tableau<T> & tab) :
 Tableau(tab._data, tab._size) // Constructeur délégué
{}

// Constructeur de déplacement
template <typename T> Tableau<T>::Tableau(Tableau<T> && tab) :
 _size(tab._size), _data(tab._data)
{
 set_new_handler(newHandler);
 tab._data = nullptr;
 tab._size = 0u;
}

// Constructeur variadique à partir d'éléments
template <typename T>
template <typename ...R>
Tableau<T>::Tableau(const T & first, const R &... others) :
 _size(0), _data(new T[sizeof...(others)+1]) // _size sera incrémenté dans addElement
{
 set_new_handler(newHandler);
 addElement(first, others...); // Pack expansion
}
```

Constructeur délégué

Tableaux.cpp

Pack expansion

# Patrons de classe – Exemple (3/6)

C++ 11

```
// Constructeur à partir d'une liste {elt1, elt2, ...}
template <typename T> Tableau<T>::Tableau(initializer_list<T> list) :
 _size(list.size()), _data(new T[_size])
{
 size_t i = 0;
 for (auto it = list.begin(); it != list.end(); ++it, i++) { _data[i] = *it; }
}

// Destructeur
template <typename T> Tableau<T>::~Tableau()
{
 if (_data != nullptr){ delete [] _data; }
 _size = 0;
}

// Accesseur taille
template <typename T> size_t Tableau<T>::size() const { return _size; }

// opérateur de copie
template <typename T> Tableau<T> & Tableau<T>::operator =(const Tableau<T> & tab)
{
 _size = tab._size;
 delete [] _data;
 if (_size > 0)
 {
 _data = new T[_size];
 for (size_t i = 0; i < _size; i++) { _data[i] = tab._data[i]; }
 }
 else
 {
 _data = nullptr;
 }
 return *this;
}
```

Tableaux.cpp

# Patrons de classe – Exemple (4/6)

C++ 11

```
// opérateur de déplacement
template <typename T> Tableau<T> & Tableau<T>::operator =(Tableau<T> && tab)
{
 size = tab.size;
 delete[] tab._data;
 _data = tab._data;

 tab.size = 0;
 tab._data = nullptr;

 return *this;
}

// Opérateurs d'accès en lecture seule, puis écriture
template <typename T> T Tableau<T>::operator [](const size_t index) const
 throw (TabException) // lecture seule
{
 if (index < _size) { return _data[index]; }
 else
 {
 cerr << "Tableau<T>::operator [](" << index << ">=" << _size << ")" << endl;
 throw TabException(TabException::INDICES_OUT_OF_BOUNDS);
 }
}

template <typename T> T & Tableau<T>::operator [](const size_t index)
 throw (TabException) // lecture/écriture
{
 if (index < _size) { return _data[index]; }
 else
 {
 cerr << "Tableau<T>::operator [](" << index << ">=" << _size << ")" << endl;
 throw TabException(TabException::INDICES_OUT_OF_BOUNDS);
 }
}
```

codes identiques

Tableaux.cpp

# Patrons de classe – Exemple (5/6)

C++ 11

```
// opérateur de cast en T*
template <typename T> Tableau<T>::operator T*() const { return _data; }

// opérateur ami de sortie standard
template <typename T> ostream & operator <<(ostream & out, const Tableau<T> & tab)
{
 out << '(' << tab._size << ")";
 for (size_t i = 0; i < tab._size; i++)
 {
 out << " " << tab._data[i];
 }
 out << "]" " ";
 return out;
}

// handler d'échec d'allocation
template <typename T> void Tableau<T>::newHandler() throw (TabException)
{
 cerr << "Can not allocate new array" << endl;
 throw TabException(TabException::LOCATION_PROBLEM);
}

// Ajout d'éléments variadique
template <typename T>
template <typename ...R> void Tableau<T>::addElement(const T & first,
 const R &... others)
{
 addElement(first);
 addElement(others...);
}

// Ajout d'éléments terminal
template <typename T> void Tableau<T>::addElement(const T & element)
{
 _data[_size++] = element;
}
```

Appel ajout terminal

Appel récursif ajout variadique

Tableaux.cpp

# Patrons de classe – Exemple (6/6)

C++ 11

```

// ----- Proto instantiations
// -----
template class Tableau<double>;
template ostream & operator <<(ostream &, const Tableau<double> &);
template Tableau<double>::Tableau(const double & first,
 const double & second,
 const double & third);
template void Tableau<double>::addElement(const double & first,
 const double & second,
 const double & third);

template class Tableau<int>;
...

int main(int argc, char ** argv)
{
 int i1[] = {1,5,7,6,1};
 // Ctor valué
 Tableau<int> t1(i1, 5);
 // Ctor avec initializer_list
 Tableau<double> td1 =
 {1.0, 5.7, 8.4, 3.2};
 // Ctor de copie
 Tableau<double> td2 = td1;
 // Ctor valué avec valeurs par défaut
 // ==> Ctor par défaut
 Tableau<double> td3;
 // Ctor de déplacement
 Tableau<double> td4(std::move(td2));
 // Ctor variadique
 Tableau<double> td5(1.0, 2.0, 3.0);
}

```

dépend des compilateurs

Autre solution :  
 implémenter tout ou partie du template dans le header → plus de compilation séparée

TestTableaux.cpp

const

non const

Tableaux.cpp

```

cout << td1[1] << endl;
td1[1] = 0.0;
cout << td1[1] << endl;

cout << t1 << td1 << td2 << td4
 << td5 << endl;

td3 = td4;
cout << td3 << endl;
td3 = {3.5, 4.4, 5.2, 6.1};
cout << td3 << endl;

try
{
 cout << td1[17] << endl;
}
catch (TabException & ex)
{
 cerr << ex.what() << endl;
}

double * tabData = (double *)td1;
...

```

opérateur de cast

# Transfert parfait (Perfect Forwarding) C++11

- Pb : Transmettre un argument d'une fonction/méthode à une autre tel quel (les lvalues en tant que lvalues et les rvalues en tant que rvalues).

```
template <typename T> void foo(T & lvalue)
{
 cout << "lv = " << lvalue << " ";
}

template <typename T> void foo(T && rvalue)
{
 cout << "rv = " << rvalue << " ";
}

template <typename T> void goodForward(T && t)
{
 foo(t);
 foo(std::forward<T>(t)); // sauf ds badForward(const T&
// <==> static_cast<decltype(t)&&>(t)
}
```

Comportement attendu du **forward** :

- rvalue transmise en tant que rvalue
- lvalue transmise en tant que lvalue

Langages Orientés Objet - David Roussel

Pas de forwarding (copie de la valeur passée en argument)

Fonctionne avec les lvalues mais pas les rvalues

Spécialisation pour les constantes (rvalues) : Mais forward interdit car cast d'un **const T &** en **T &&** looses const qualifier !

On parle non plus de « rvalue references » mais de « universal references »

```
int main(void)
{
 int i = 2;
 noForward(1); // lv = 1 rv = 1
 noForward(i); // lv = 2 rv = 2
 badForward(1); // lv = 1
 badForward(i); // lv = 2 rv = 2
 goodForward(1); // lv = 1 rv = 1
 goodForward(i); // lv = 2 lv = 2
}
```

# Patrons de classes - Héritage

C++

```
class A { ... };
```

```
template <typename T> class M { ... };
```

- Patron héritant d'un patron :
  - `template <typename T> class N : public M<T> { ... } ;`
- Patron héritant d'une classe :
  - `template <typename T> class P : public A { ... } ;`
- Classe héritant d'une instantiation de patron :
  - `class B : public M<int> { ... } ;`

## 2.14 Robustesse

# 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] = zeroElement();

 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

### 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