

Programmation orientée objet

2^e année

Sabeur ELKOSANTINI

Sabeur.Elkosantini@isima.rnu.tn

MCours.com

Bibliographie

Livres

- Michel Divay , « Java et la programmation orientée objet », Dunod eds, 2006.
- Renaud Pawlak , Jean-Philippe Retailé , Lionel Seinturier, « Programmation orientée aspect pour Java / J2EE », Eyrolles eds, 2004.
- Bruce Eckel, « Thinking in Java », 2nd revision, 2000 (Disponible sur internet).

Autres supports de cours

- Cours de Jean-Michel DOUDOUX :
<http://www.jmdoudoux.fr/java/dej/>
- Cours de Mickaël BARON :
<http://mbaron.ftp-developpez.com/javase/java.pdf>

Plan

- Chapitre 1 : Introduction
- Chapitre 2 : Le concept d'objets
- Chapitre 3 : POO avec C++
- Chapitre 4 : ... et avec Java

Plan

- **Chapitre 1 : Introduction**
- Chapitre 2 : Le concept d'objets
- Chapitre 3 : POO avec C++
- Chapitre 4 : ... et avec Java

Introduction

Historique de la POO

- Les années 60 : le langage Simula-67, langage de simulation informatique
 - ✓ Les premiers pas de la programmation orientée objet

- Les années 70 : SmallTalk , apparition des concepts de base :
 - ✓ objet, encapsulation, polymorphisme, héritage , etc.

- Les années 80 : La Montée en puissance de l'orienté objet
 - ✓ Apparition de nouveaux langages : Objective C ,C++, Eiffel, Common Lisp Object System

Introduction

Historique de la POO

- Les années 90 : l'âge d'or de l'extension de la POO :
 - ✓ Standardisation de C++
 - ✓ Apparition du langage de programmation Java

- Depuis, évolution de l'orientée objet:
 - ✓ Analyse par objet (AOO).
 - ✓ La conception orientée objet COO
 - ✓ Les bases de données orientées objets (SGBDOO)

Introduction

Programmation procédurale Vs Programmation OO

- Programmation procédurale (C, Cobol, Fortran, Pascal, etc.) :
 - ✓ Programmes structurés en procédures et fonctions,
 - ✓ Des problèmes en cas de modification de la structures des données,
 - ✓ Chaque fonction ou procédure résout une partie du problème,

- Programmation OO (Java, C++, C#, Delphi, etc.):
 - ✓ Unité logique : objet,
 - ✓ Programmation par « composants »,
 - ✓ Facilité de l'évolution du code,
 - ✓ Améliorer la conception et la maintenance des grands systèmes,

Introduction

👉 Programmation par Objets ?

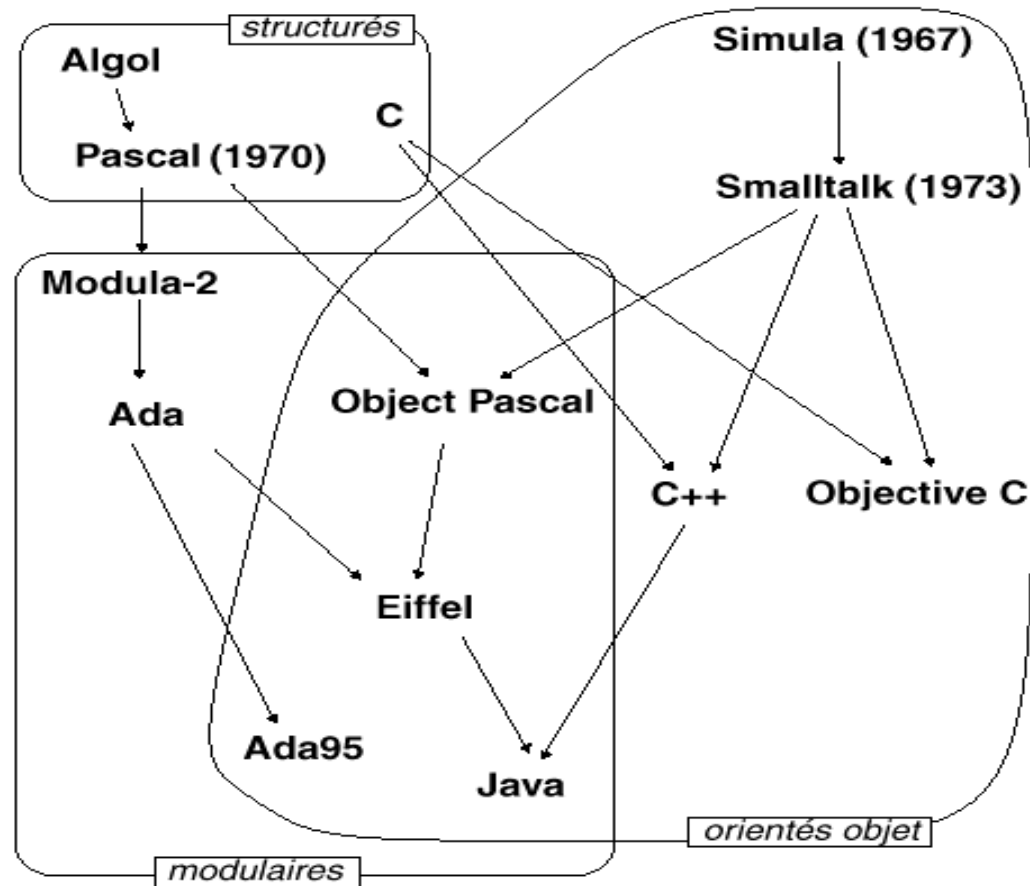


- Unité logique : l'objet
- Objet est défini par :
 - Une identité : permet de distinguer un objet d'un autre objet.
 - Un état : représenté par des attributs (variables) qui stockent des valeurs.
 - Un comportement : défini par des méthodes (procédures) qui modifient des états.

<u>Moto</u>
Couleur : noir Vitesse : 150 Km/h
Accélérer Freiner

Introduction

👉 Les langages de programmation



Introduction

👉 Le langage de programmation C++

- Développé dans les laboratoires d'AT&T Bell au début des années 1980 par Bjarne Stroustrup.
- C++ est un langage (hybride) : à typage fort, compilé et *orienté objet*

C++ = C + typage fort + objets

Un langage hybride ?!



Un langage compilé ?!

Un langage à typage fort ?!

Objets ??

Quelles différences avec C ??

Plan

- Chapitre 1 : Introduction
- **Chapitre 2 : Le concept d'objets**
- Chapitre 3 : POO avec C++
- Chapitre 4 : ... et avec Java

Le concept d'objets

👉 Notion d'objet

- Un objet est défini à la fois par des informations : données ou attributs ou variables d'instances ; et des comportements : traitements ou méthodes ou opérations.



<u>Moto</u>
Couleur Vitesse_limite
Accélérer Freiner

Objet Moto

Le concept d'objets

Notion de classe

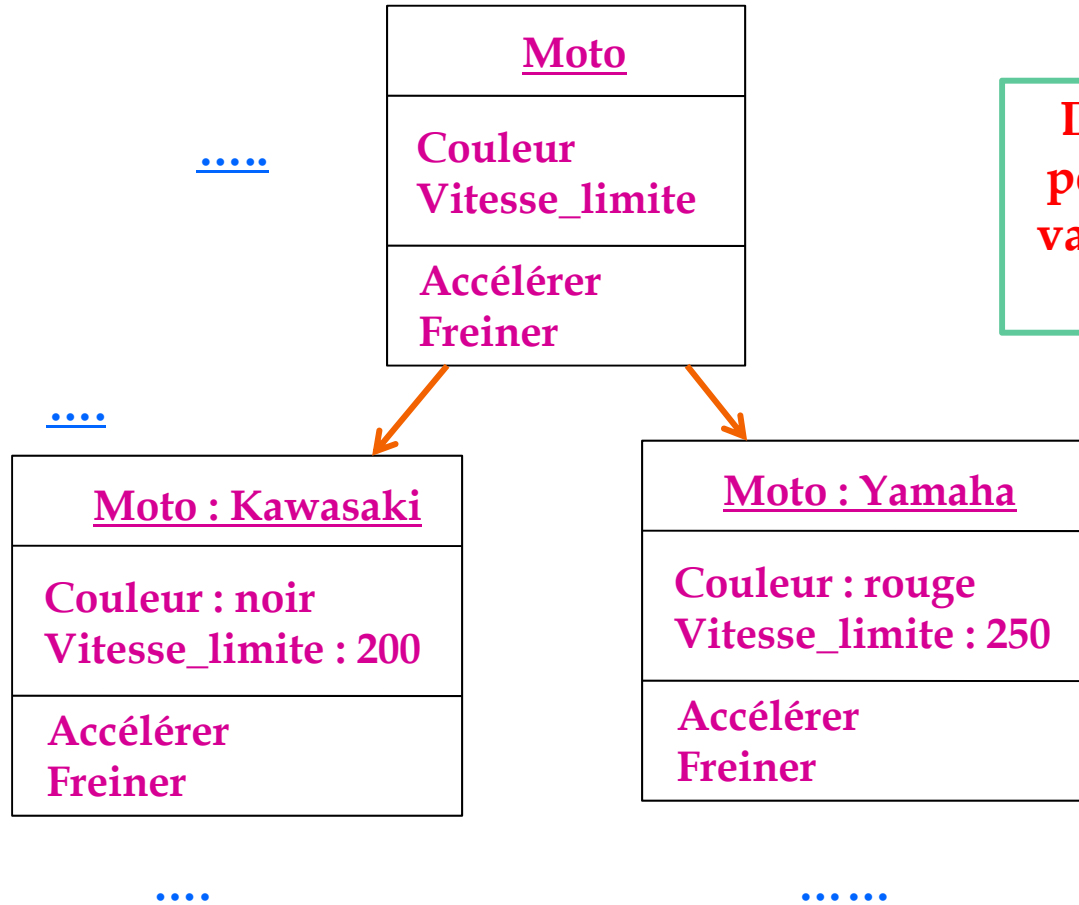
- Lorsque des objets ont les mêmes attributs et comportements : ils sont regroupés dans une famille appelée : **Classe**
 - Les objets appartenant à celle-ci sont les instances de cette classe.
- L'instanciation est la création d'un objet d'une classe.

<u>Moto</u>
Couleur : noir Vitesse_limite : 200
Accélérer Freiner

<u>Moto</u>
Couleur : rouge Vitesse_limite : 250
Accélérer Freiner

Le concept d'objets

👉 Classe, objet et instantiation

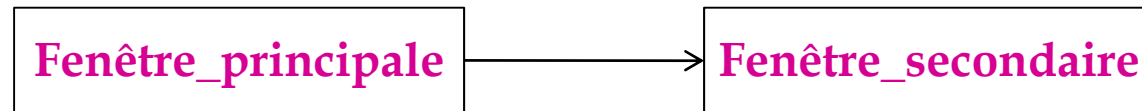


Deux instances d'une même classe peuvent avoir des attributs avec des valeurs différentes et mais partagent les mêmes méthodes.

Le concept d'objets

Classe, objet et instantiation

Comment les objets communiquent-ils ?



→ Par

Un équivaut à un appel d'une méthode.

Le concept d'objets

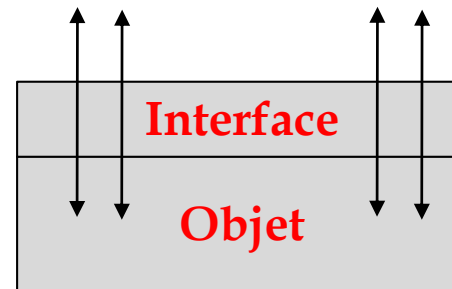
L'encapsulation et la visibilité des attributs

- De point de vue utilisation, un objet est une boîte noire qui offre un certain nombre de méthodes permettant d'interagir avec lui.
- Peu importe comment il est construit en interne, la seule chose nécessaire pour pouvoir utiliser un objet est de savoir ce qu'il peut faire et surtout comment lui demander :
 - ❖ Exemple : un poste de TV est une « boîte noire » ayant pour interface : un écran, des HP et une télécommande. Pour changer de chaîne, il suffit de
..... *.Peu importe ce qui se passe réellement en interne.*

Le concept d'objets

👉 L'encapsulation et la visibilité des attributs

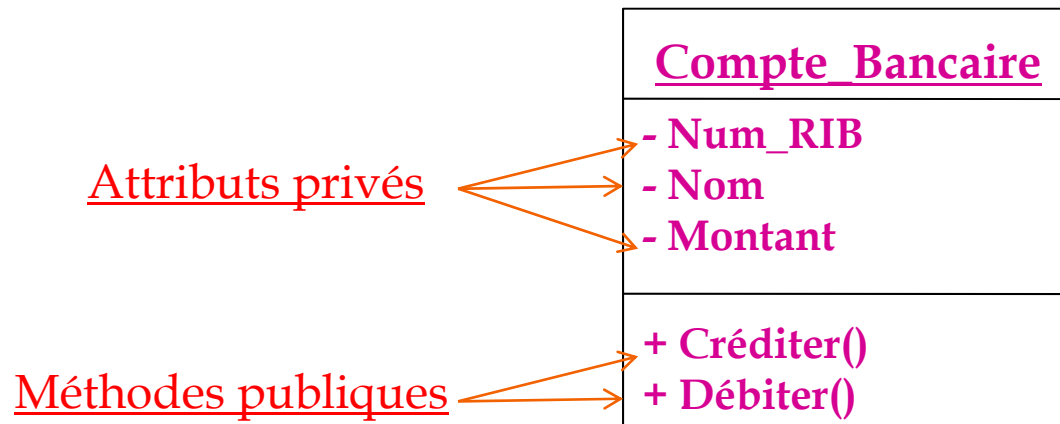
- L'ensemble des **méthodes** proposées par un objet est appelé **l'interface** de cet objet.
- On dit qu'un objet est **encapsulé** par son **interface** : la seule manière d'interagir avec cet objet est **d'invoquer** une méthodes de son interface. Peu importe de quoi cet objet est réellement constitué, ce qui est important c'est les services (les méthodes) qu'il peut fournir.



Le concept d'objets

👉 L'encapsulation et la visibilité des attributs

- L'encapsulation est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet :
 - ✓ Empêcher l'accès aux données par un autre moyen que les services proposés.
 - ✓ Garantir l'intégrité des données contenues dans l'objet .



Le concept d'objets

L'encapsulation et la visibilité des attributs

- Il existe trois niveaux de visibilité :
 - ✓ Publique : veut dire que les attributs ou les méthodes sont disponibles pour tout le monde
 - ✓ Privé : veut dire qu'aucune autre classe ne peut accéder au contenu de l'attribut et l'implémentation de la méthode concerné,
 - ✓ Protégé : l'accès aux données est réservé aux fonctions des classes héritières, c'est-à-dire par les fonctions membres de la classe ainsi que des classes dérivées.

Le concept d'objets

Les méthodes d'une classes

- On distingue trois types de méthodes dans une classe:
 - ✓ Les constructeurs qui permettent d'initialiser les champs d'un objet

Un constructeur est une méthode particulière, sans valeur de retour, portant le même nom que la classe

Déclaration : `Moto :: Moto (string nouvelle_couleur){
Couleur= nouvelle_couleur;
}`

Le concept d'objets

Les méthodes d'une classes

- Une classe peut avoir plusieurs constructeurs ou aucun :
 - ✓ Dans ce dernier cas, C++ fournit un constructeur par défaut: c'est-à-dire un constructeur sans arguments et initialise chaque champs par la valeur par défaut nulle

Déclaration :

```
Moto :: Moto () {  
    // constructeur crée par défaut  
}
```

- ✓ Dès qu'une classe possède au moins un constructeur, le constructeur par défaut ne sera pas disponible sauf si la classe possède un constructeur sans arguments

Le concept d'objets

Les méthodes d'une classes

- Les méthodes get... et set ... utilisés en cas de protection des attributs:
 - ✓ Les **méthodes d'accès** qui permettent de renvoyer les informations relatives à un objet

```
Déclaration : int Moto :: get_vitesse(){  
                return vitesse;  
            }
```

- ✓ Les **méthodes d'altération** qui modifient l'état d'un objet (les valeurs de certains champs), donc elles comportent certains contrôlent pour valider les nouvelles valeurs.

```
Déclaration :      int Moto :: set_vitesse (int nouvelle_vitesse){  
                    vitesse = nouvelle_vitesse;  
                }
```

Le concept d'objets

Les méthodes d'une classes

- **Le constructeur** : c'est une méthode qui est appelée automatiquement à chaque fois que l'on crée un objet basé sur cette classe.
- **Le destructeur** : c'est une méthode qui est automatiquement appelée lorsqu'un objet est détruit, par exemple à la fin de la fonction dans laquelle il a été déclaré ou lors d'un *delete* si l'objet a été alloué dynamiquement avec *new*.
- **Le destructeur** libère la zone mémoire allouée par le constructeur. Il s'exécute à la fin du programme ou d'un bloc où des objets locaux ont définis.

Le concept d'objets

Les méthodes et les classes amies

- Une méthode est amie d'une classe si elle peut accéder directement à toutes les données privées de cette classe.
- Une classe est amie d'une autre classe si toutes ses fonctions lui sont amies.
- La déclaration d'amitié doit se faire dans la classe qui autorise les accès à ses données privées.

Le concept d'objets

Les méthodes et les classes amies

Déclaration :

```
class A
{ friend void Methode1 ();
  friend class C;
  ...
}
```

- La méthode *Methode1* peut accéder aux données privées de A.
- Toutes les méthodes de la classe C peuvent accéder aux données privées de la classe A.

Le concept d'objets

La surcharge des méthodes

- Redéfinir une méthode déjà existante dans la classe.
- La modification de la méthode porte sur :
 - ✓ Le type de retour de la méthode
 - ✓ Le nombre de paramètres de la méthode et leur type.
- Quand une méthode surchargée est invoquée le compilateur sélectionne automatiquement la méthode dont le nombre et le type des arguments correspondent au nombre et au type des paramètres passés dans l'appel de la méthode

Le concept d'objets

La surcharge des méthodes

- Exemple de surcharge : la surcharge de constructeurs

```
Déclaration : Moto :: Moto (int v){
                vitesse = v;
                }
Moto ::      Moto (int v; string c){
                vitesse = v;
                Couleur= c;
                }
Moto :: Moto (string c){
                Couleur= c;
                }
```

Le concept d'objets

L'héritage

- L'héritage est un mécanisme qui facilite la réutilisation du code et la gestion de son évolution. Elle définit une relation entre deux classes :
 - ✓ Une classe mère ou super classe
 - ✓ Une classe fille ou sous classe qui hérite de sa classe mère
- les objets d'une classe fille ont accès aux données et aux méthodes de la classe parent et peuvent les étendre.
- Les sous classes peuvent redéfinir les variables et les méthodes héritées.
- Les méthodes des classes filles sont redéfinies avec le même nom, les mêmes types et le même nombre d'arguments

Le concept d'objets

L'héritage

la classe Etudiant :	la classe Etudiant-Elu :
<i>nom</i> <i>capital UV</i> <i>diplôme</i>	<i>nom</i> <i>capital UV</i> <i>diplôme</i> <i>Mandat</i> <i>Syndicat</i>
<i>VérifierNom</i> <i>MajUV</i> <i>ChangerDiplôme</i>	<i>VérifierNom</i> <i>MajUV</i> <i>ChangerDiplôme</i> <i>DémissionnerMandat</i> <i>ChangerSyndicat</i>

Le concept d'objets

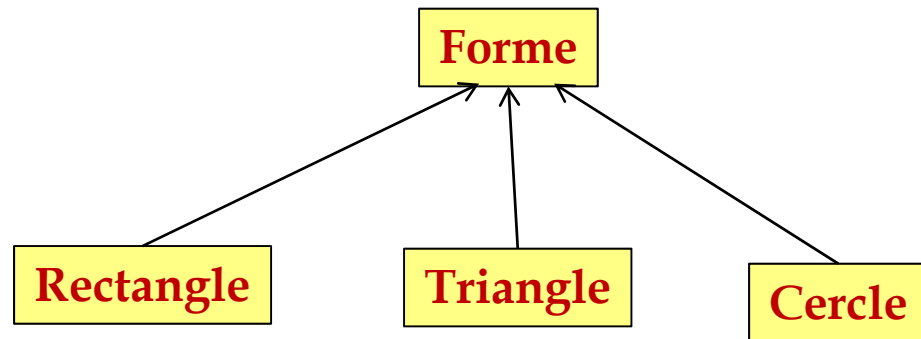
L'héritage

- L'objet *Etudiant-Elu* a les propriétés (attributs et méthodes) de l'objet *Etudiant* mais en plus possède d'autres propriétés.
- La classe *Etudiant-Elu* est une spécialisation de la classe *Etudiant*. C'est une sous classe de la classe *Etudiant*.
- Les objets de la sous classe *Etudiant-Elu* héritent des attributs et des méthodes de la classe *Etudiant*. La sous classe *Etudiant-Elu* pourra, si cela est nécessaire pour ses besoins, redéfinir une méthode héritée.

Le concept d'objets

L'héritage

- Chaque sous classe peut avoir une ou plusieurs sous classes formant ainsi une hiérarchie d'objet. On parle de classe ancêtre (ou mère) et de classes descendant (ou filles).



- L'héritage est un mécanisme qui permet d'assurer une grande variabilité dans la réutilisation des objets. Il existe deux techniques liées à l'héritage : les classes abstraites et l'héritage multiple.

Le concept d'objets

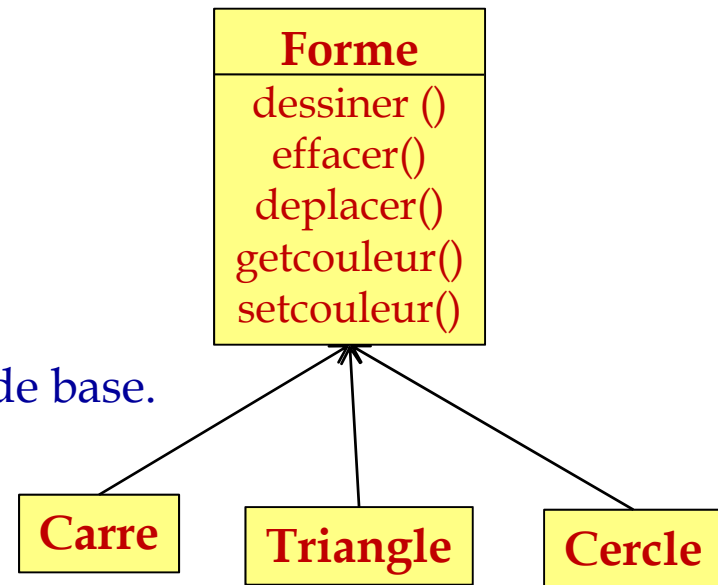
L'héritage

- Autre exemple : les formes géométrique (utiliser dans les systèmes d'aide à la conception des jeux vidéo)

- Le type de base est la « forme géométrique »

- La classe dérivée est du même type que la classe de base.

Les objets de la classe dérivée n'ont pas seulement le même type, ils ont aussi le même comportement, ce qui n'est pas particulièrement intéressant



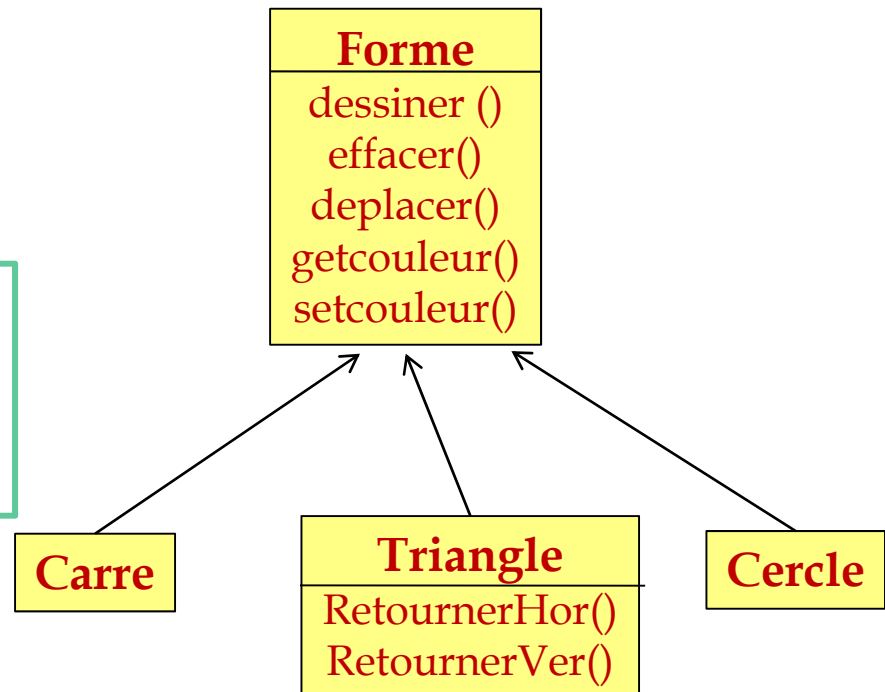
Le concept d'objets

L'héritage

- Pour différencier une classe dérivée d'une classe parent :
 - ajouter d'autres méthodes dans les classes dérivées: la classe de base n'était pas assez complète



Il faut vérifier s'il ne faut pas intégrer ces fonctions dans la classe de base qui pourrait aussi en avoir l'usage

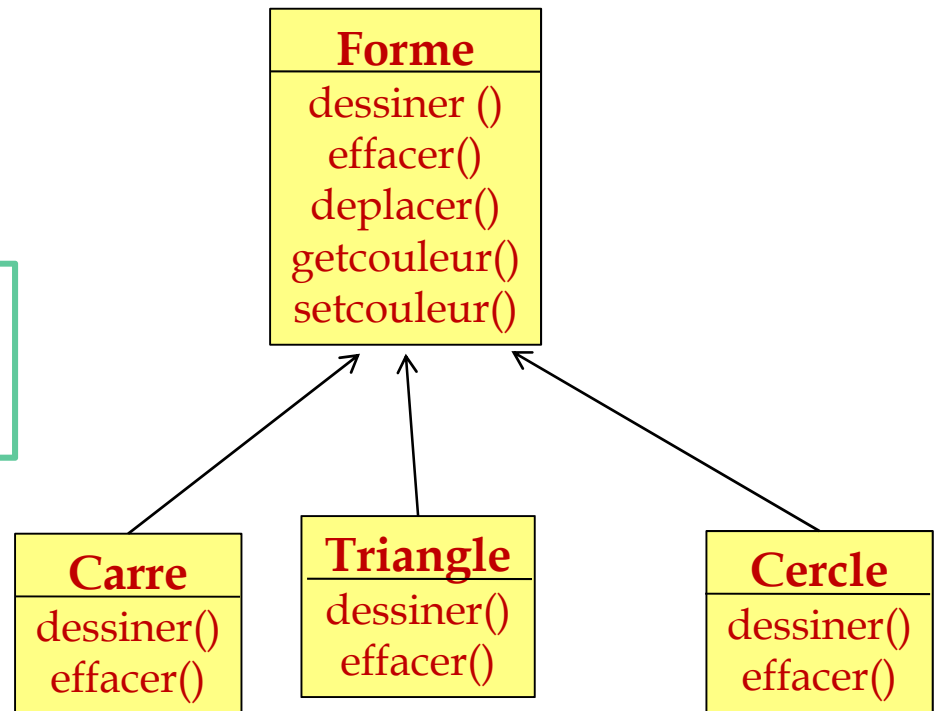


Le concept d'objets

L'héritage

- Pour différencier une classe dérivée d'une classe parent :
 - Redéfinir autrement le comportement des classes dérivées. C'est-à-dire

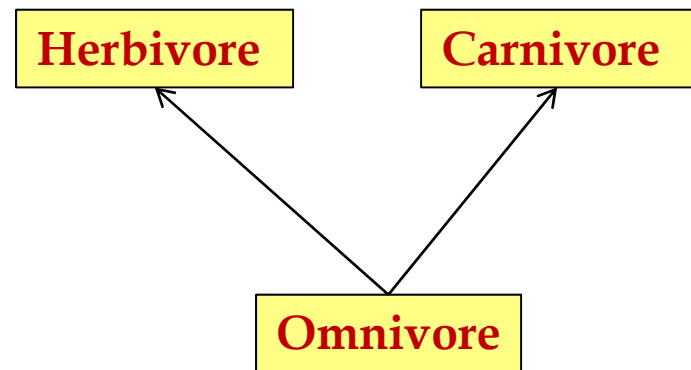
il suffit de créer une nouvelle
définition pour la fonction
dans la classe dérivée



Le concept d'objets

L'héritage multiple

- Certains langages orientés objet, tels que le C++, permettent de faire de l'héritage multiple.
- Faire hériter une classe de deux superclasses.
- Regrouper au sein d'une seule et même classe les attributs et méthodes de plusieurs classes.



Plan

- Chapitre 1 : Introduction
- Chapitre 2 : Le concept d'objets
- **Chapitre 3 : POO avec C++**
- Chapitre 4 : ... et avec Java

Spécificités de C++

- Fonction sans argument :
 - En C++ : `float sansarg();`
 - En C : `float sansarg(void);`

- Une fonction qui ne retourne rien en C++ :
 - `void sansretour(int x);`

POO avec C++

Spécificités de C++

- Expression constante (expression dont la valeur peut être calculée à la compilation) :

- En C++, le compilateur sait évaluer :

```
const int MAX = 100;
```

```
double tab1[2*MAX+1], tab2[2*MAX+1][MAX];
```

- En C, en général. On doit utiliser *#define* :

```
#define MAX 100
```

```
double tab1[2*MAX+1], tab2[2*MAX+1][MAX];
```

POO avec C++

Spécificités de C++

- En C, les commentaires sont entre `/*` et `*/` ;
- En C++, on ajoute à celle-ci les commentaires de fin de ligne qui démarrent par `//` et vont jusqu'à la fin de ligne.

POO avec C++

Spécificités de C++

- En C, obligation de regrouper toutes les déclarations au début du programme.
- En C++, ce n'est plus obligatoire. Elles peuvent apparaître n'importe où, avant d'être utilisée.

```
void main()
{ int i;
  i=3;
  ...
  int q=3*i;
  ...
  for (int j=0; j<q; j++) ...
}
```


POO avec C++

Spécificités de C++

- Gestion simplifiée des fonctions d'entrées/sortie grâce à 2 nouvelles fonctions : **cin** et **cout**
 - En saisie, **cin** Exemple : `cin >> x;`
 - En affichage, **cout**

Exemple :

```
cout << "coucou";
```

```
cout << "voici le nombre : " << x << endl;
```

POO avec C++

Remarques :

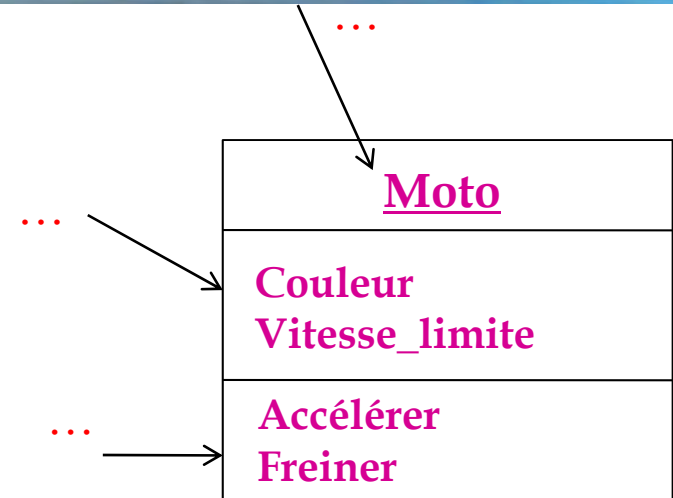
- Les anciennes fonctions sont toujours utilisables;
- L'opérateur de référence & n'est pas nécessaire pour la saisie;
- Un processus de vérification automatique de type permet de s'affranchir des multiples formats de type très utilisés en C.

Du nouveaux ?

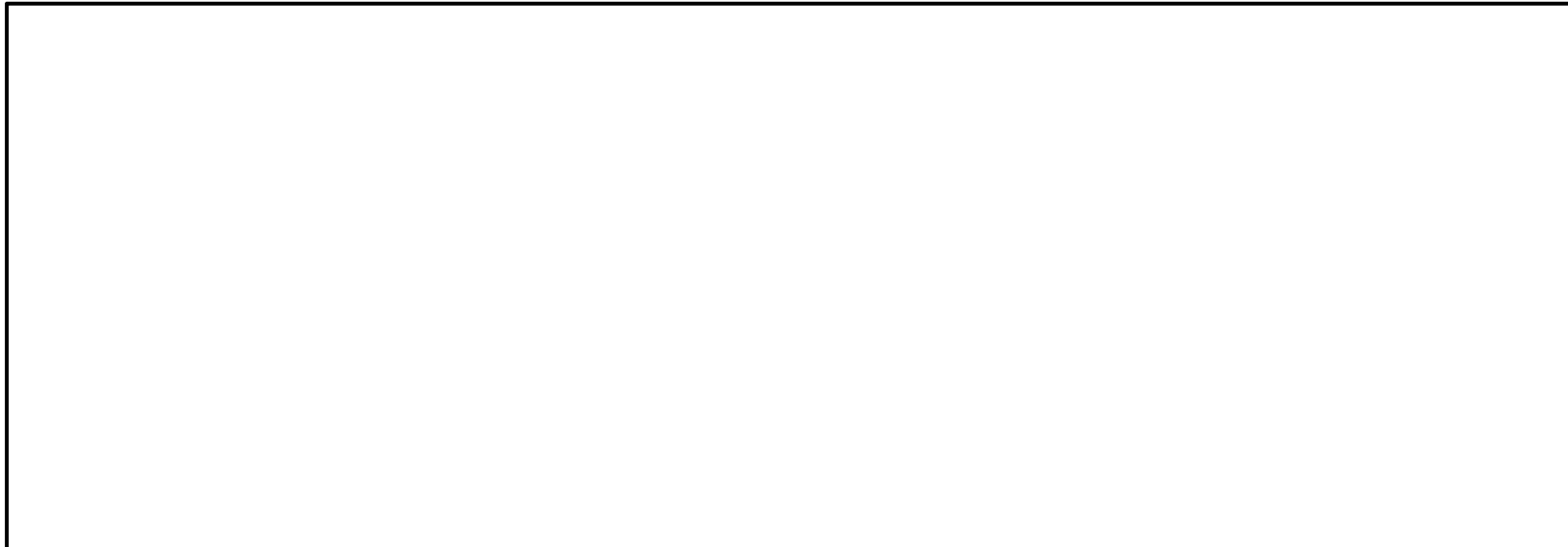
- Nouvelle forme de commentaire
- Liberté dans l'emplacement des déclarations ;
- Surcharge de fonction ;
- Allocation dynamique par les opérateurs *new* et *delete* ;
- Notions OO;

POO avec C++

👉 Les objets et les classes en C++

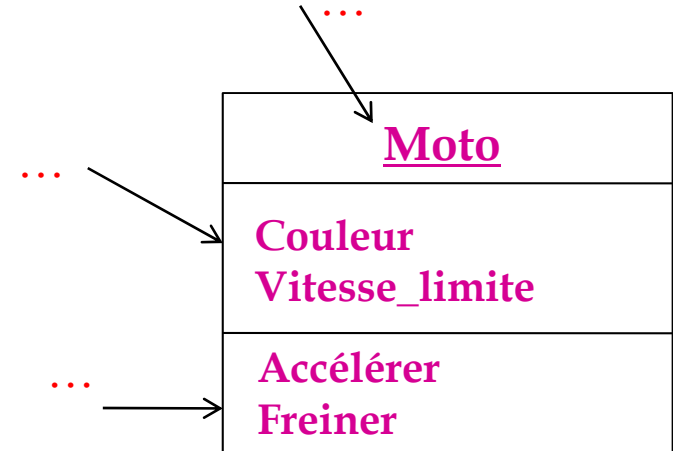


L'implémentation de cette classe en C++ (avec les headers)

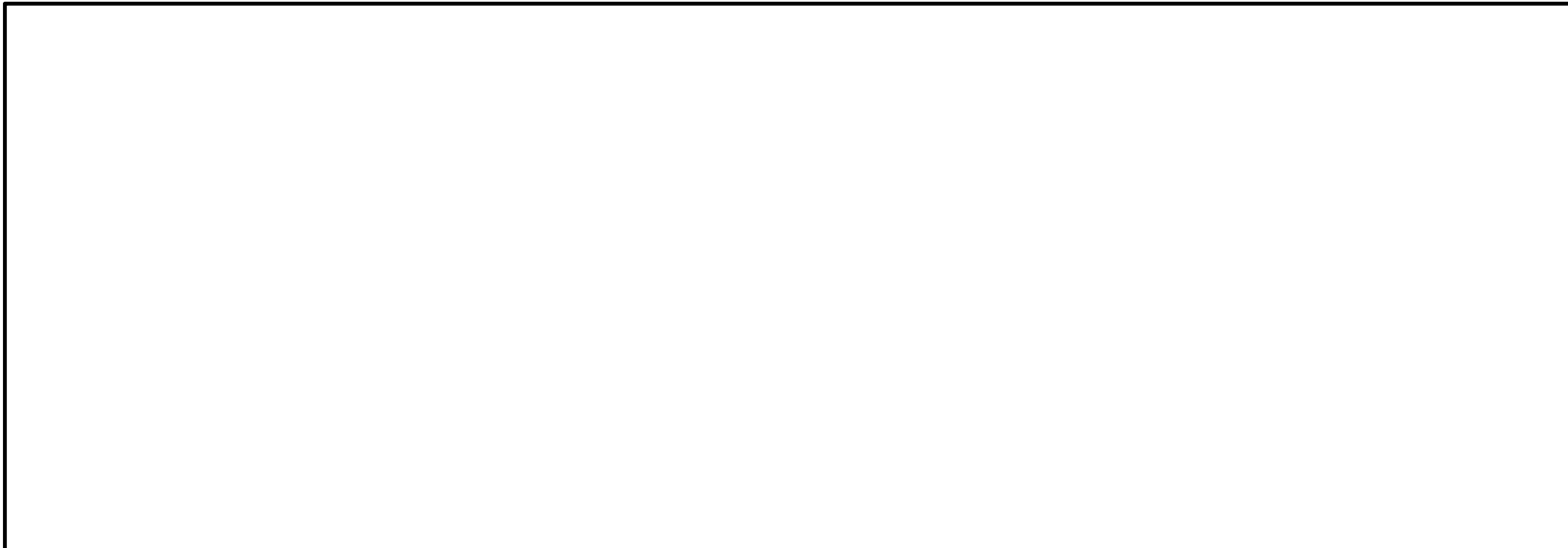


POO avec C++

👉 Les objets et les classes en C++



L'implémentation de cette classe en C++ (sans les headers)



POO avec C++

👉 La visibilité des variables et des attributs

```
class Moto {  
    int Vitesse_actuelle;  
    int Age;  
    int Prix;  
    ...  
};  
  
Moto::Moto(int age; int Vitesse_Initiale , int prix_I) :  
    Vitesse_actuelle(Vitesse_Initiale), Age (age), Prix(prix_I)  
{ }  
void Moto:: Accelerer(int v)  
{  
    Vitesse_actuelle += v;  
}  
  
int Moto:: valeur(){  
    int amortissement;  
    amortissement = 0.05;  
    return Prix - Prix * amortissement * age;  
}
```

Des remarques ?? ?



... et le constructeur en C++

- **Chaque** appel à un constructeur crée un **nouvel** objet (instance) qui obéit au patron défini par la classe :
 - l'instance créée aura les attributs et le comportement définis dans la classe.
 - réservation d'un espace mémoire pour la mémorisation de l'état.
- Le constructeur est généralement l'occasion d'initialiser les attributs (« personnaliser » l'état de l'instance).
- Il peut y avoir plusieurs constructeurs pour une même classe.
 - Plusieurs initialisations possibles.

POO avec C++

 ... et le constructeur en C++

Statique

La construction en C++:
Classe + variable + (valeurs)

C'est le constructeur

```
Produit p1 (« Portable », 1);  
Produit p2 ();
```

Comment programmer
la classe Produit ?? ?



POO avec C++

 ... et le constructeur en C++

Dynamique

La construction en C++:
Classe + * variable = new Classe (valeurs)

C'est le constructeur

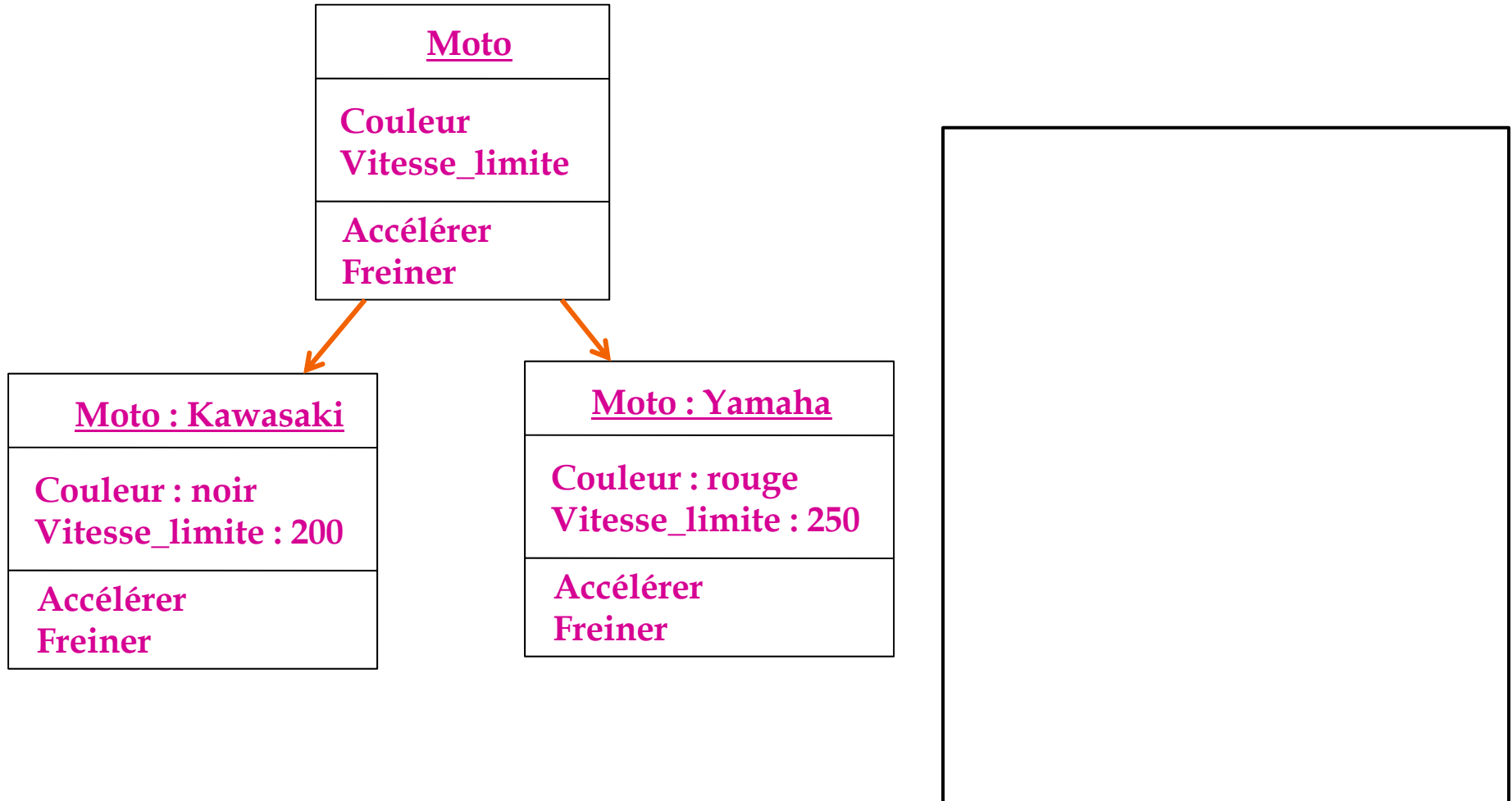
```
Produit * p1 = new Produit (« Portable », 1);  
Produit p2 = new Produit ();
```

Comment programmer
la classe Produit ?? ?



POO avec C++

👉 L'instanciation



Le cycle de vie d'un objet

- La création d'un objet ou, autrement dit,

Objet obj ;



Cette opération déclare uniquement le nom et le type de l'objet. Les attributs et les méthodes ne sont pas encore créer.

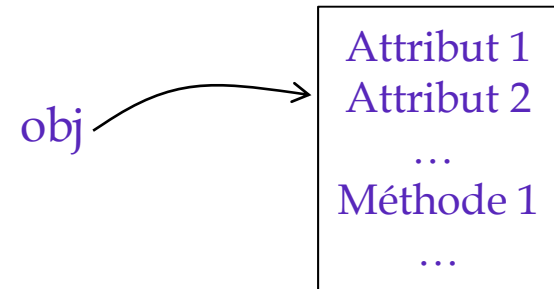
Le cycle de vie d'un objet

- La création d'un objet ou, autrement dit,

Objet obj ;



Objet obj () ;



C'est à ce moment que l'allocation mémoire est réalisée

POO avec C++

Les objets, les classes et l'instanciation en C++

- Accès aux attributs et méthodes d'un objet à partir d'un autre objet .

```
Nomobjet.Nommethode() ;  
Nomobjet.NomAttribut;
```

- L'envoi de messages entre objets :

.....



Il ne faut pas oublié les paramètres dans les messages

- Les cascades sont possibles : Magasin.produit.nom

POO avec C++

👉 Les objets, les classes et l'instanciation en C++

- Dans le traitement de l'une de ses méthodes, un objet peut avoir à **s'envoyer un message** (pour accéder à un de ses attributs ou invoquer une des ses méthodes).
- Utilisation de l'**auto-référence**, en C++: **this**.

Exemple : on se place dans une méthode de la classe *Moto* :

- Lors du traitement, l'objet appelant la méthode est une instance de la classe *Moto*.

this->Freiner() signifie « envoyer à this (= moi-même) le message Freiner() »

POO avec C++

👉 Les objets, les classes et l'instanciation en C++

- L'appel à l'objet courant : utilisation du mot clé *this*

```
this->methode();  
this->attribut;
```

- Exemple d'utilisation

```
void Moto::acc()  
{  
    .....  
}  
void Moto::acc2()  
{  
    ....  
}
```

POO avec C++

Les objets, les classes et l'instanciation en C++

- Si pas d'ambiguïté, le mot clé **this** peut être oublié :

```
this->Freiner () ;    →    Freiner();  
this->Prix;           →    Prix;
```

Exercice 5 :

Ecrire une classe Livre, caractérisée par les attributs titre, auteur et année et par les méthodes suivantes :

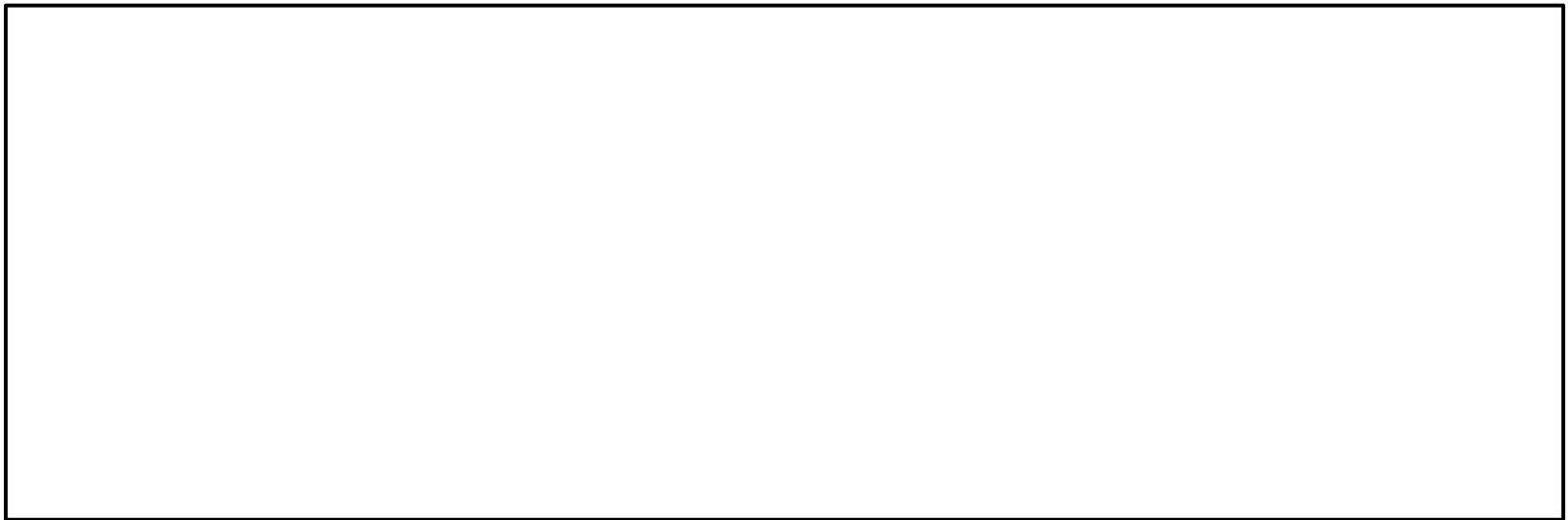
- Une méthode affichant les caractéristiques d'un livre
- Une méthode qui prend en paramètre un livre et qui permet d'afficher les deux livres et de le comparer le nom du livre passé en paramètre.

POO avec C++

Les objets, les classes et l'instanciation en C++

■ Exercice 6 :

Créer une classe Segment qui est définie par ses extrémités (des points) et par sa couleur. Avec cette classe, on peut calculer la longueur d'un segment, le déplacer et changer sa couleur.



Les méthodes et classes amies

```
friends type_de_retour nom_methode(parametres);
```

La méthode est amie à la classe concernée

```
friends class A;
```

La classe A est amie à la classe concernée

Exemple :

POO avec C++

La surcharge

- La surcharge de méthodes : un mécanisme donnant la possibilité d'appeler plusieurs méthodes avec le même nom.



Des méthodes surchargées peuvent avoir des types de retour différents à condition qu'elles aient des arguments différents.

Exemple : la surcharge de
la méthode somme

```
int Calcul::somme( int p1, int p2){  
    return (p1 + p2); }
```

```
float Calcul:: somme( float p1, float p2){  
    return (p1 + p2); }
```

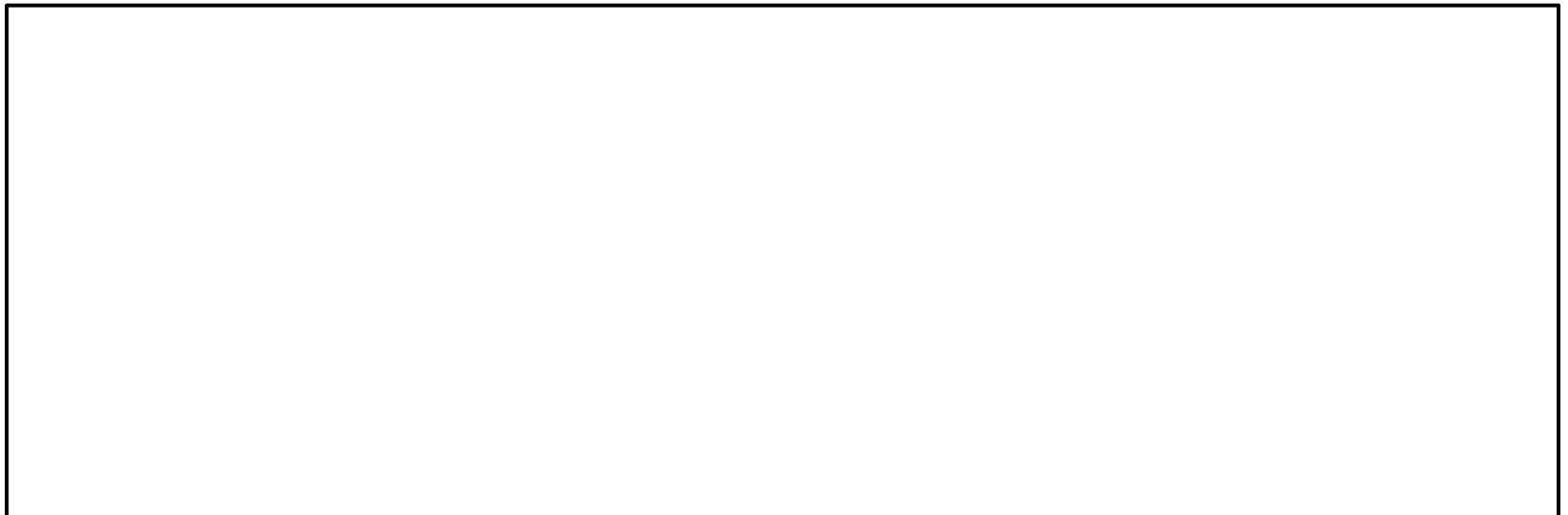
```
float Calcul:: somme( float p1, float p2, float p3){  
    return (p1 + p2 + p3); }
```

```
int Calcul:: somme( float p1, int p2){  
    return (int(p1) + p2); }
```

POO avec C++

La surcharge

- Exercice 7:
 1. Créez une classe avec un constructeur qui imprime un message.
 2. Ajoutez à cette classe un constructeur surchargé qui prend un string en argument et qui l'imprime avec votre message.
 3. Créez, dans le programme principale, deux instances de cette objet pour tester les deux constructeurs.



La surcharge d'opérateurs

- En C++, il y a la possibilité de redéfinir le comportement de certains opérateurs mathématique ou logique: == , +=, +, - , &, etc.
- Il suffit d'ajouter une méthode spécifique :

```
class Moto
{
...
bool operator+ (const Moto& m) ;
};
bool Moto::operator+ (const Moto& m)
{

couleur += m.couleur;
vitesse += m.vitesse
}
```

```
void main()
{
....
Moto m, m1, m2;
m=m1+m2;
....
}
```

La surcharge d'opérateurs

- Ou ,autrement, en utilisant une méthode amie à deux paramètres :

```
class Moto
{
...
friend bool operator + (const Moto& m1, const Moto& m1 ) ;
};

bool Moto::operator + (const Moto& m1, const Moto& m1)
{
m1.couleur += m2.couleur;
m1.vitesse +=m2.vitesse
}
```

```
void main()
{
....
Moto m, m1, m2;
m=m1+m2;
....
}
```

La surcharge d'opérateurs

- Ou ,autrement, à l'extérieur de la classe :

```
class Moto
{
...
};
```

```
bool Moto::operator + (const Moto& m1, const Moto& m1)
{
m1.couleur += m2.couleur;
m1.vitesse +=m2.vitesse
}
```

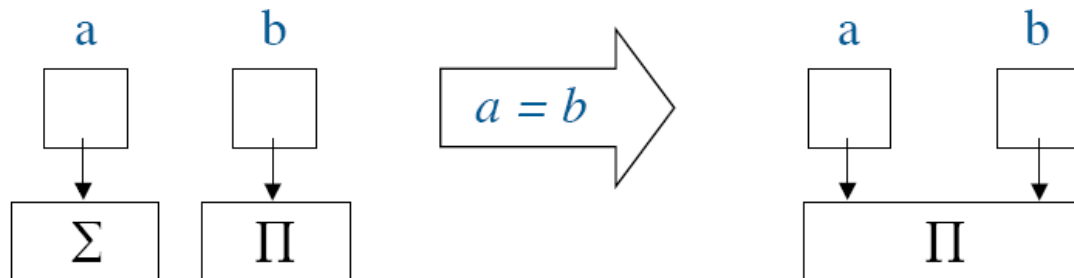
```
void main()
{
...
Moto m, m1, m2;
m=m1+m2;
...
}
```

👉 La manipulation des objets

- L'affectation et la comparaison :

```
Objet a ();  
Objet b ();  
a = b ;
```

Quel est la différence entre les deux instances a et b ?



POO avec C++

La manipulation des objets

- L'affectation et la comparaison :

Objet a ();

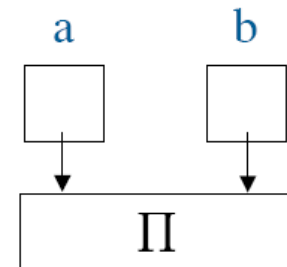
Objet ();

boolean rep = (a == b) ;

Que signifie cette comparaison ?



➔ Vérifier si les deux objets ont la même référence.



POO avec C++

La manipulation des objets

- L'affectation et la comparaison :

Quel est le résultat de l'instruction
BMW==Mercedes ?



<u>Voiture : BMW</u>
Puissance : 9 Couleur: noir
Accélérer Freiner

Voiture : Mercedes
Puissance: 9 Couleur : noir
Accélérer Freiner

➔ Les deux objets ont les mêmes valeurs d'attributs mais la référence est différente.

La destruction des objets

- Le destructeur de classe consiste en une méthode qui sera appelée lors de toute destruction d'un objet.
- Peut-on surcharger un destructeur ? **non.**
- Est-ce qu'un destructeur peut prendre des paramètres ? **non**
- Doit-on appeler un destructeur ? **non**
- Et si l'objet est créé dynamiquement avec *new* ?

La destruction des objets

```
#include <iostream>
using namespace std;
class A {
public:
A() {
    cout << "Constructeur de A.";
}
~A()
{
    cout << "Destructeur de A";
}
};

int main() {
A * a = new A;           // .....
delete a;                // .....
}
```

POO avec C++

Encapsulation en C++

- Il existe trois niveaux de visibilité :
 - ✓ Privé : en C++, private
 - ✓ publique: en C++, public
 - ✓ protégé: en C++, protected

```
public :  
string var1;  
int methode1();
```

```
protected :  
char var2;  
void methode2());
```

```
private :  
int var3;  
void methode3());
```

Compte Bancaire

```
- Num_RIB  
- Nom  
- Montant
```

```
+ Créditer()  
+ Débiter()
```


POO avec C++

Encapsulation en C++

- Exemple : Application pour la gestion des comptes en banques.
 - La classe Banque ayant comme attribut solde.
 - Dans un premier temps, l'attribut est déclaré comme publique.

Quel est l'inconvénient d'un tel programme ?



Quelle amélioration proposez-vous ?



Encapsulation en C++



+ Compte Bancaire

- Num_RIB
- Nom
- Montant

+ Créditer()
+ Débiter()

**Implémenter cette
classe en C++**

POO avec C++

Le mot clé static

- Aucune méthodes ni attributs n'est accessible avant l'instanciation de la classe en utilisant le mot clé
 - ✓ La zone mémoire n'est pas encore allouée.



Les méthodes et les attributs sont alors associés à l'objet et non à la classe.

- ... et si on veut que la donnée ou la méthode n'est pas spécifiquement rattachée à un objet instance d'une classe ?

Utilisation du mot clé static

Le mot clé static

```
class StaticTest {  
    static int i = 47;  
}
```

- Et dans une autre classe:

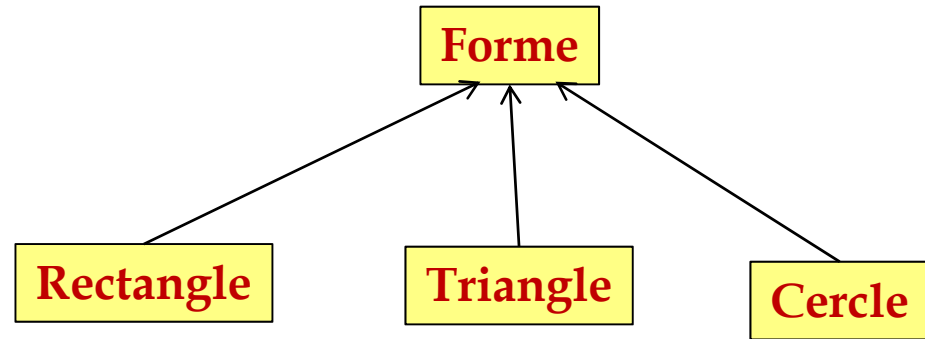
```
StaticTest st1 = new StaticTest();  
StaticTest st2 = new StaticTest();
```

Comment incrémenter l'attribut i ?



POO avec C++

L'héritage



En C++, en utilisant les deux points « : »

```
class Forme
```

```
{
  // ...
};
```

```
class Rectangle :Forme
```

```
{
  // ...
};
```

```
class Triangle :Forme
```

```
{
  // ...
};
```

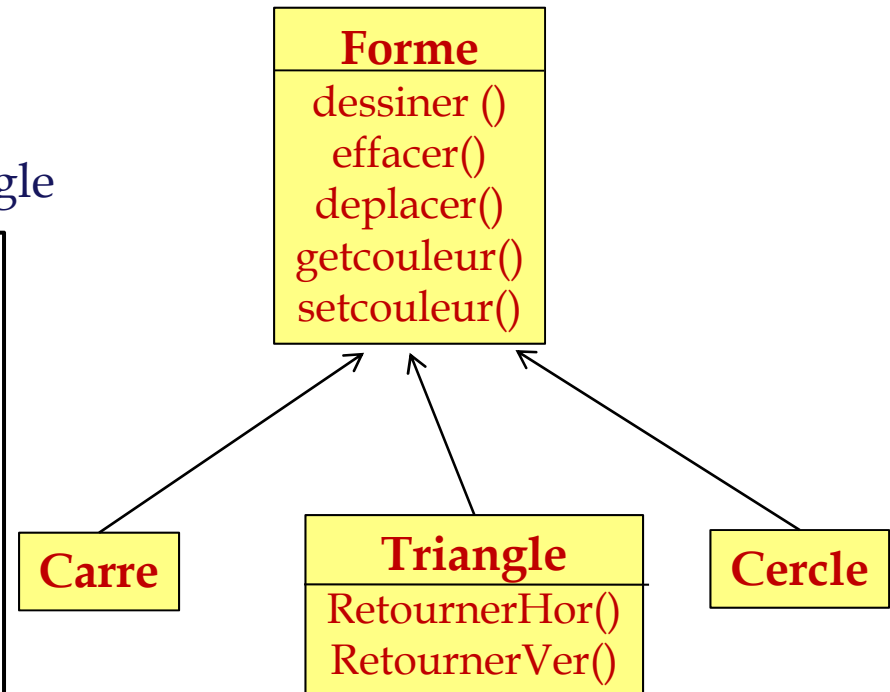
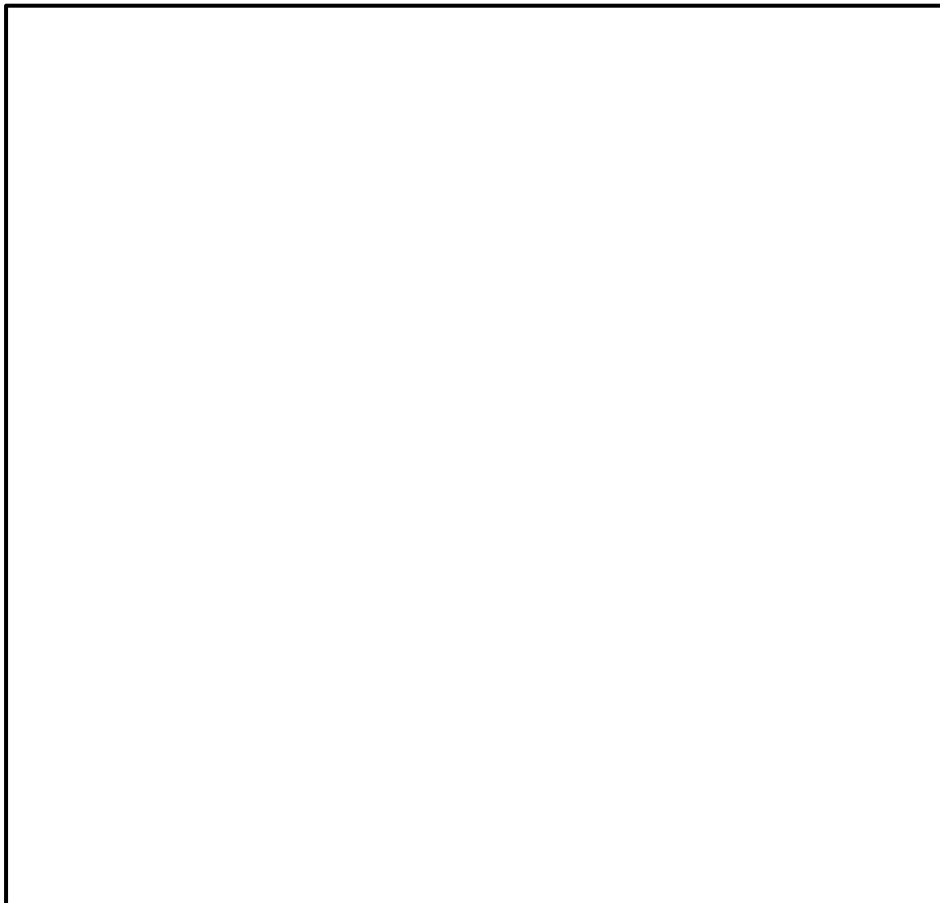
Et le mot clé *protected* ?



POO avec C++

L'héritage

L'implémentation en C++ de la classe Triangle



POO avec C++

L'héritage et amitié:

Pas d'héritage au niveau des déclarations d'amitié.



```
class A {
friend class Base;
public:
A() { cout << "Constructeur de A."; }
~A() { cout << "Destructeur de A"; }
private :
int nb;
};
class Base {
public:
Base() { cout << "Constructeur de Base .";
cout << att.nb."; }
~ Base () { cout << "Destructeur de Base "; }
private :
A att;
};
```

```
class Derive : Base{
public:
Derive() { cout << "Constructeur de Derive.";
cout << att1.nb}
~ Derive () { cout << "Destructeur de Derive "; }
private :
A att1;
};
```

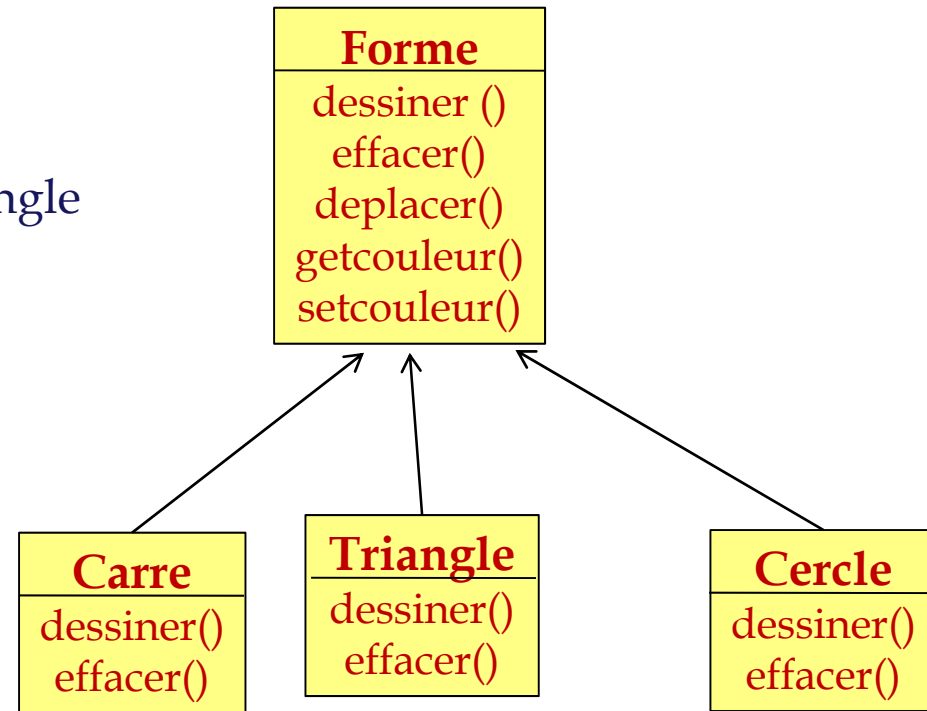
Quel est le résultat de ce programme ?? ?



POO avec C++

👉 L'héritage : le polymorphisme

L'implémentation en C++ de la classe Triangle

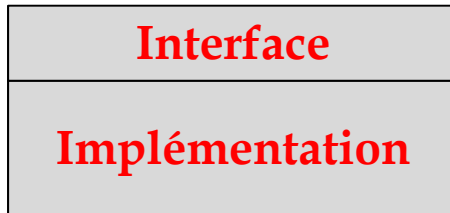


POO avec C++

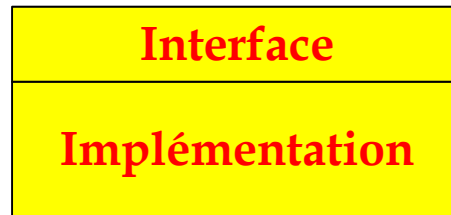
👉 L'héritage : le polymorphisme indésirable

*Forme *f = new Triangle ();*

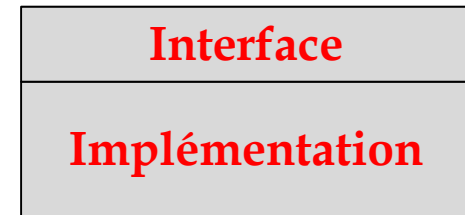
Classe Forme



Classe Triangle



L'instance f



f->dessiner();

Selon le diagramme suivant, quel est le résultat de l'instruction *f.dessiner()* ?

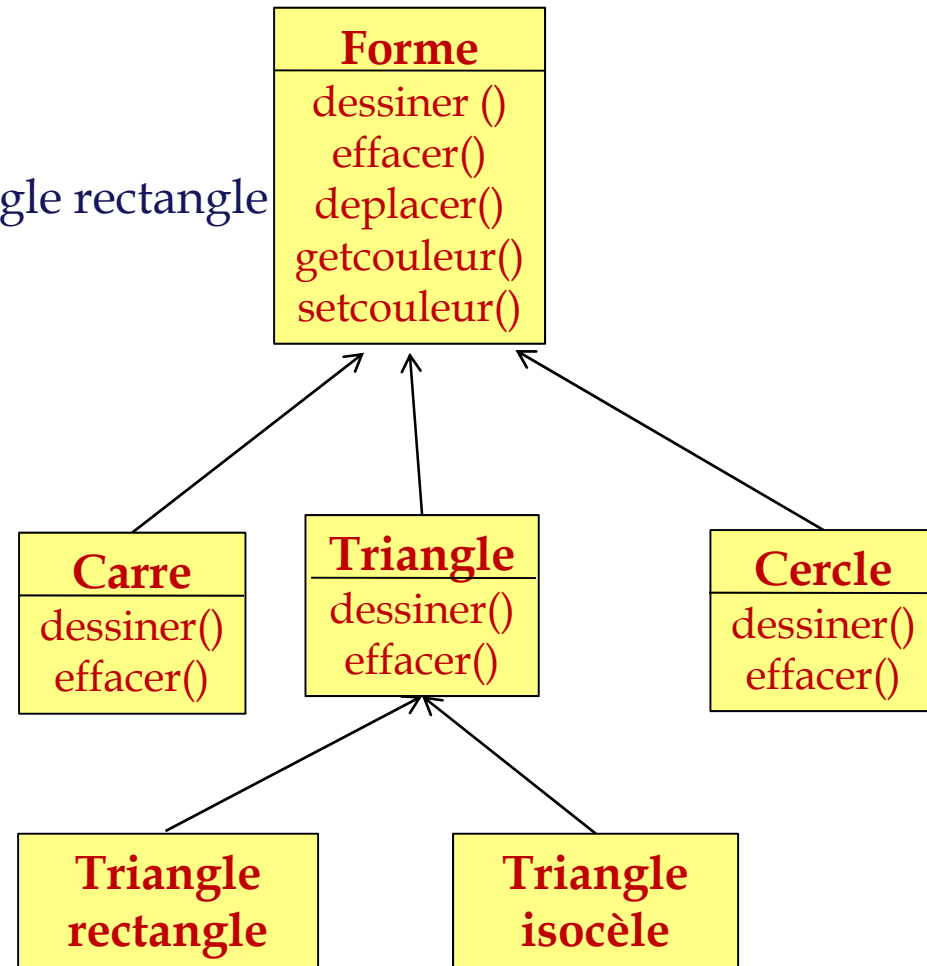


- Un pointeur vers une classe de base ne permet d'accéder qu'aux membres hérités et son usage ne saurait donc respecter la règle d'accès par défaut aux membres propres.

POO avec C++

👉 L'héritage à plusieurs niveaux

L'implémentation en C++ de la classe Triangle rectangle



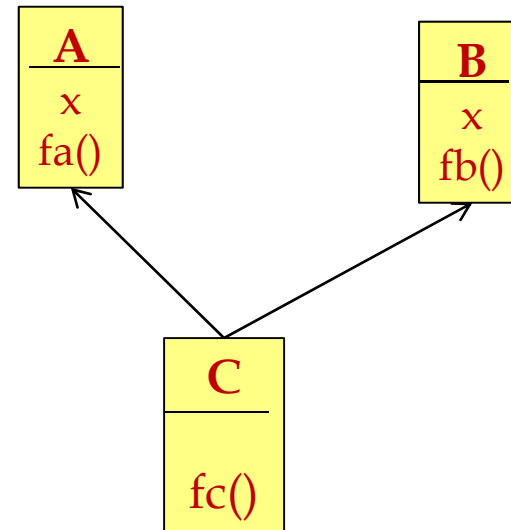
L'héritage : Ordre d'appel des constructeurs et des destructeurs

- L'instanciation d'une classe dérivée implique nécessairement l'exécution d'un constructeur de la classe de base puis celle d'un constructeur de la classe dérivée.
- Lors de la destruction d'un objet avec *delete*, l'exécution d'un destructeur de la classe dérivée puis celle d'un destructeur de la classe de base.

MCours.com

👉 L'héritage multiple

L'implémentation en C++ de la classe Triangle rectangle



POO avec C++

L'héritage multiple : Ordre d'appel des constructeurs

- Les constructeurs sont appelés dans l'ordre de déclaration de l'héritage.

```
class A {  
    public:  
        A(int n) {...}  
};  
class B {  
    public:  
        B(int n) {...}  
};
```

```
class C : public B, public A {  
    public:  
        C(int i, int j) : A(i), B(j) {...}  
};  
void main() {  
    C obj_c;  
    ...  
}
```

Appel des constructeurs B(), A() et C()

POO avec Java

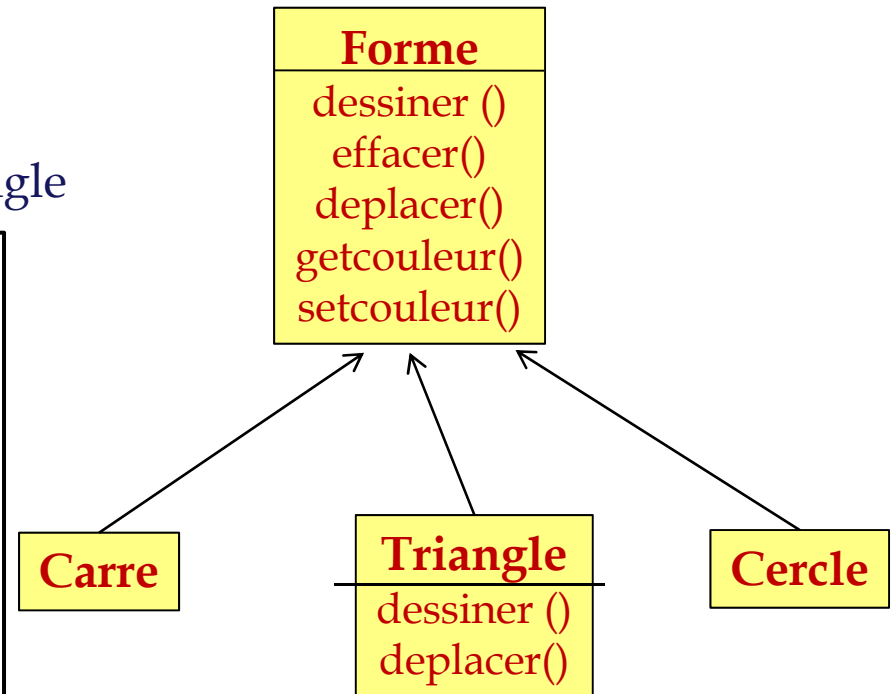
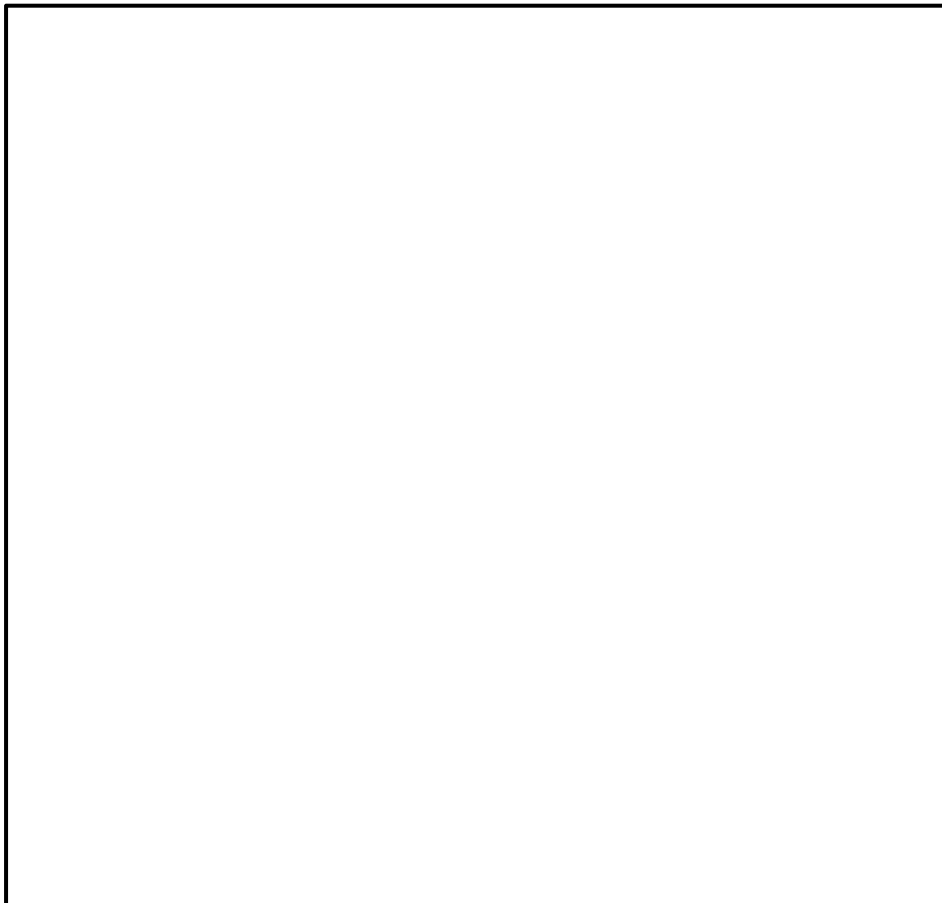
Les méthodes virtuelles

- Lorsqu'une méthode est déclarée comme virtuelle dans la classe de base, sa redéfinition ne donne pas naissance au phénomène de « polymorphisme indésirable ».
- Si une méthode virtuelle est invoquée par l'intermédiaire d'un pointeur sur la classe de base, c'est le type de l'objet dont l'adresse se trouve dans le pointeur au moment de l'appel qui détermine quelle fonction est exécutée

POO avec C++

👉 Les méthodes virtuelles

L'implémentation en C++ de la classe Triangle



dessiner() et deplacer() sont virtuelles

POO avec C++

Les méthodes virtuelles



Lorsqu'une méthode est redéfinie dans une classe dérivée, elle doit être virtuelle

```
class A
{
public:
    A() { std::cout << "Constructeur de A.\n"; }
    ~A() { std::cout << "Destructeur de A.\n"; }
    Virtual void PrintName() { std::cout << "Classe A.\n"; int j=0 ; }
};
```

```
void main()
{
    A * p= new B();
    p->PrintName();
}
```

```
class B : public A
{
public:
    B() { std::cout << "Constructeur de B.\n"; }
    ~B() { std::cout << "Destructeur de B.\n"; }
    void PrintName() { std::cout << "Classe B.\n"; }
};
```

Quel est le résultat ??

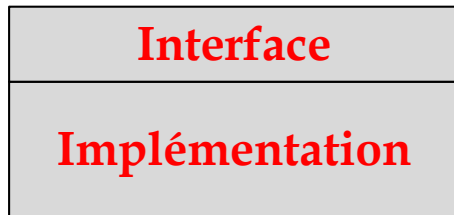


POO avec C++

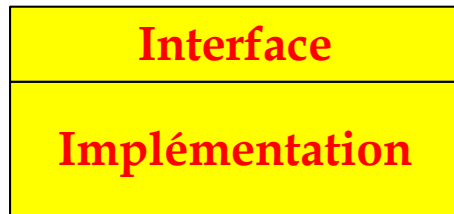
Les méthodes virtuelles

- Pour l'exemple suivant :

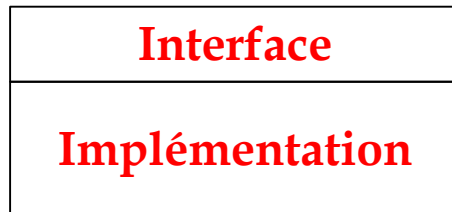
Forme f = new Trianglerectangle ();



Classe Forme

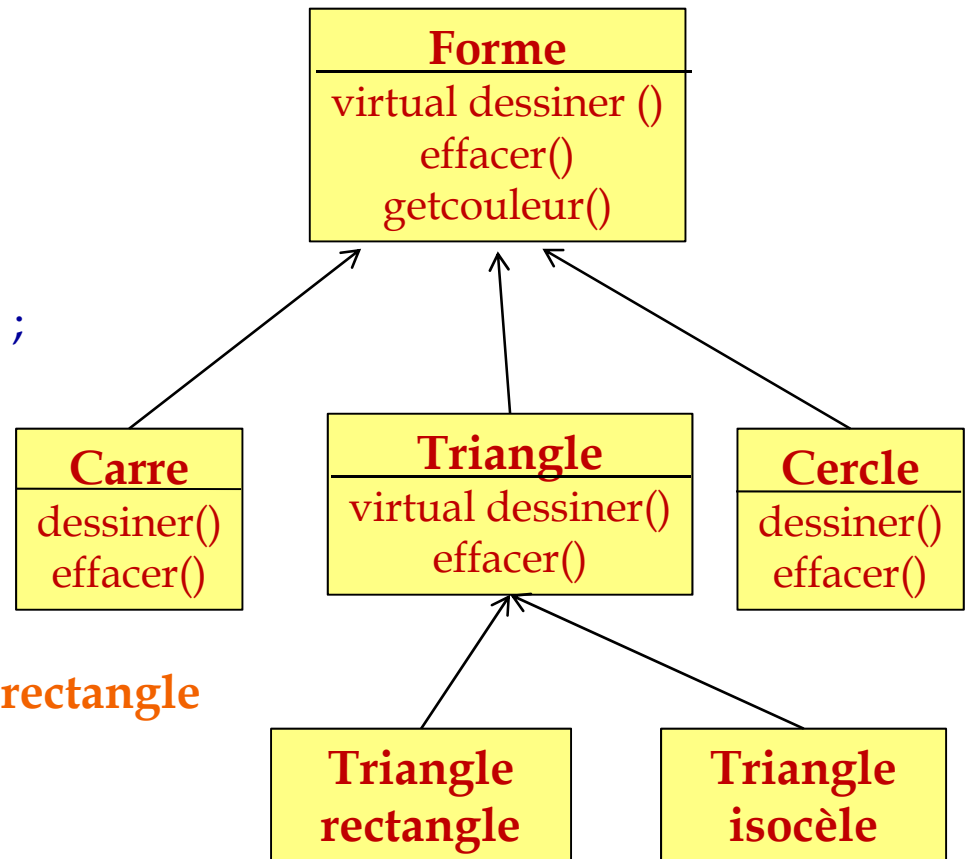


Classe Trianglerectangle



L'instance f

Selon le diagramme suivant, quel est le résultat de l'instruction *f.dessiner()* ?



POO avec C++

👉 Les méthodes virtuelles : Ordre d'appel des constructeur et des destructeurs

```
class A
{
public:
    A() { std::cout << "Constructeur de A.\n"; }
    ~A() { std::cout << "Destructeur de A.\n"; }
};
```

```
void main()
{
    A * p= new B();
    delete p;
}
```

```
class B : public A
{
public:
    B() { std::cout << "Constructeur de B.\n"; }
    ~B() { std::cout << "Destructeur de B.\n"; }
};
```

Quel est le résultat ?? ?



Et si on ajoute le mot *virtual* aux destructeur de la classe A ??



POO avec C++

Les méthodes virtuelles

- Récapitulation : Dans quel cas les méthodes virtuelles et le polymorphisme sont utilisés ?

✓

```
class Employe {  
public :  
string nom;  
int id, salaire;  
Employe(String n) : nom(n){  
virtual void affiche ()  
cout<<"Je suis un employe";}
```

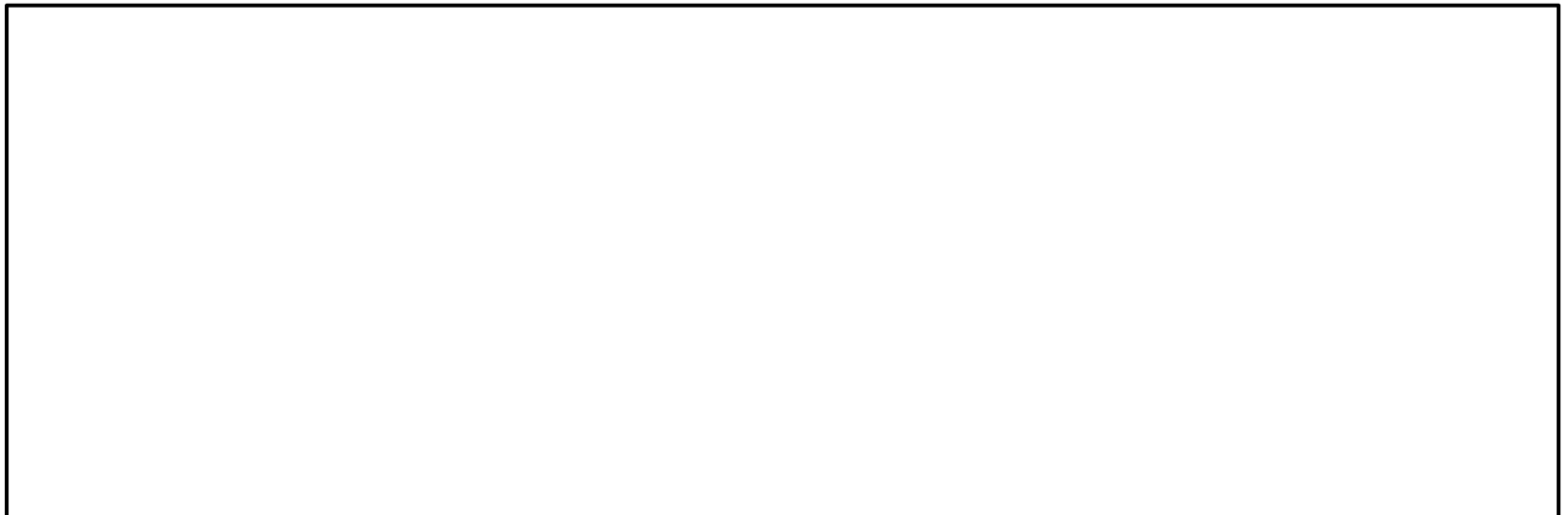
```
class magasinier : Employe {  
public magasinier(String n) : Employer(n),  
salaire(250){  
  
} void affiche ()  
{  
cout<<"Je suis un magasinier";  
}
```

Dans le programme principal, créer un magasinier en exécuter la méthode affiche de l'objet créer. Quel est le résultat ?

POO avec C++

L'héritage

- Exercice 8: Créer une hiérarchie d'héritage de **Rongeur**: **Souris**, **Gerbille**, et **Hamster**. Dans la classe de base, fournir des méthodes qui sont communes à tous les Rongeurs, et les redéfinir dans les classes dérivées pour exécuter des comportements différents dépendant du type spécifique du Rongeur. Créer un tableau de Rongeur, le remplir avec différent types spécifiques de Rongeurs, et appeler vos méthodes de la classe de base pour voir ce qui arrive.



POO avec C++

L'héritage

■ Exercice 9:

Ecrivez les classes nécessaires au fonctionnement du programme suivant. Donnez uniquement le nom des classes et les méthodes (sans détailler le contenu).

```
void main()
```

```
    Batiments *B[3] ;  
    B [0] = new Appartement(" Atef" ) ;  
    B [1] = new Villa("Walid" ) ;  
    B [2] = new Studio(" Anis" ) ;  
    For (int i = 0; i<3 ; i++)  
        B [i]->affiche() ;  
    }  
}
```

Le résultat de ce programme est :

```
L'appartement appartient à Atef  
La villa appartient à Walid  
Le studio appartient à Anis
```

Les classes abstraites : les méthodes virtuelles pures

- La présence d'une méthode virtuelle pure dans une classe a 2 conséquences:
 - ✓ La classe ne peut plus être instanciée,
 - ✓ Toute classe fille de cette classe doit redéfinir la méthode en question.

Syntaxe : `virtual void affiche() = 0;`



Lorsqu'une classe contient une méthode virtuelle pure, elle est qualifiée d'abstraite.

Les classes abstraites

- Une classe abstraite n'existe que pour être héritée.
- Une classe est dite abstraite si elle contient au moins une fonction virtuelle pure.

```
class X {  
    // Affiche est une fonction virtuelle pure car = 0.  
    virtual void affiche() = 0;  
};
```



Il est impossible de créer (instancier) un objet à partir d'une classe abstraite.

```
X a(); // Erreur
```

Les classes abstraites

- Les classes qui héritent d'une classe abstraite doivent obligatoirement définir la ou les fonctions virtuelles pures.

```
class Y:public X {  
    void affiche() {  
        cout << "Y:f<< <<endl;  
    }  
};
```



Même si le mot-clé *virtual* ne précède pas le nom de la fonction affiche, elle reste quand même virtuelle car dans la classe de base, elle est déclarée ainsi. Donc, pas besoin de le préciser encore une fois.

Plan

- Chapitre 1 : Introduction
- Chapitre 2 : Le concept d'objets
- Chapitre 3 : POO avec C++
- **Chapitre 4 : ... et avec Java**

Introduction

Le langage Java : historique

- 1990 - Sté Sun Microsystems (James Gosling, Naughton, Sheridan) : projet de langage petits systèmes
- 1991 : Introduction du langage « Oak »
- 1993 : Essor d'Internet : adapter Java au Web
- 1993 à 1995 : adaptation d'Oak pour le Web (Exécution d'applets)
- 1995 : présentation de Java TM par Sun et mise à disposition gratuite du JDK sur le net

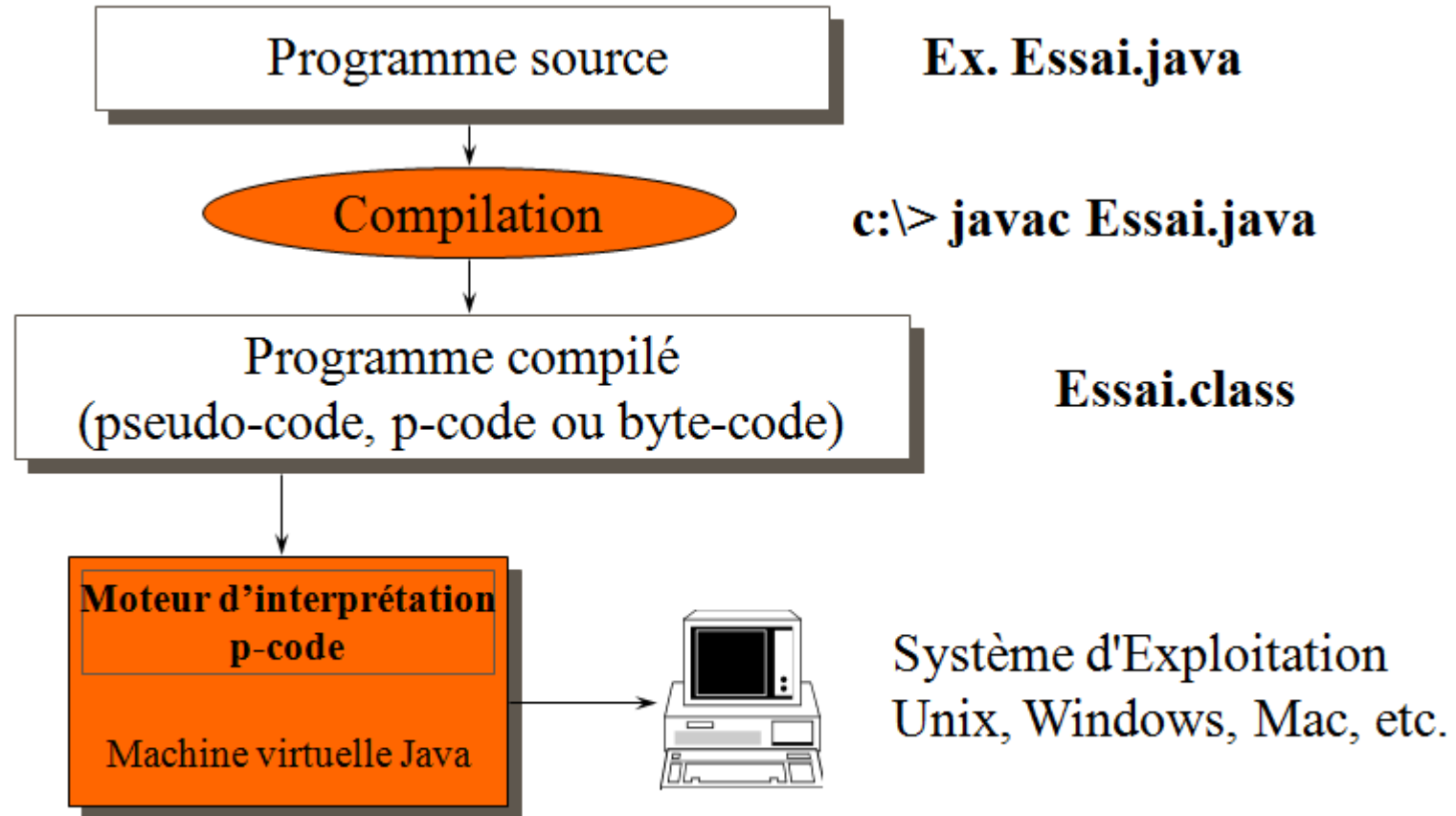
Introduction

Qualités majeures de Java

- Simple (comparé à C++)
 - ✓ Gère les débordements mémoire.
 - ✓ Gère lui-même la désallocation mémoire (ramasse-miettes).
 - ✓ Pas de manipulation explicite de pointeurs.
 - ✓ Pas de surcharge d'opérateurs.
- Fondamentalement Orienté-Objet : pas de fonctions/procédures : que des méthodes !
- Portable : principe de machine virtuelle

Introduction

👉 Le principe de la machine virtuelle



Ex. Interpréteur Java dédié ou inclus dans un navigateur

Introduction

Un programme Java, pour voir ...

Édition du programme source :

```
// Exemple de programme minimal : Salut.java  
import java.io.*;           //....
```

```
public class Salut {  
    public static void main (String args[]) {  
        System.out.println("Salut tout le monde");  
    }  
}
```

Compilation :

C:\Temp\> javac Salut.java → génération du fichier

Exécution :

D:\Temp\> java Salut

Introduction

Java Vs C++

■ En C++:

```
// Exemple de programme minimal : Salut.cpp
#include <stdio.h>           //.....
    void main () {
        printf("Salut tout le monde");
    }
```

■ En Java:

```
// Exemple de programme minimal : Salut.java
import java.io.*;         //.....

public class Salut {
    public static void main (String args[]) {
        System.out.println("Salut tout le monde");
    }
}
```

Les premiers pas en Java

Plan

- Les opérateurs mathématiques , logiques et de conversion
- Les boucles et les structures de contrôles
- Les tableaux
- Les commentaires

Les premiers pas en Java

La structure d'un programme Java

```
import java.io.*;
public class Salut {
    public static void main (String args[]) {
        afficher (5);
    }
    public void afficher (int val ) {
        System.out.println(" le nombre à afficher est " +val);
    }
}
```

Diagram illustrating the structure of a Java program with annotations:

- `import java.io.*;` is annotated with `...` and an arrow pointing left.
- `public class Salut {` is annotated with `...` and an arrow pointing left.
- `public static void main (String args[]) {` is annotated with `...` and an arrow pointing left.
- `afficher (5);` is annotated with `...` and an arrow pointing left.
- `public void afficher (int val) {` is annotated with `...` and an arrow pointing left.
- `System.out.println(" le nombre à afficher est " +val);` is annotated with `...` and an arrow pointing left.
- `}` is annotated with `...` and an arrow pointing left.



Nom du fichier = nom de la classe

Les premiers pas en Java

Les types primitifs

- Ne pas confondre les types primitifs et les objets

- Les types primitifs :

- ✓ Entiers : byte (1 octet) - short (2 octets) - int (4 octets) - long (8 octets)
- ✓ Flottants : float (4 octets) - double (8 octets)
- ✓ Booléens : boolean (true ou false)
- ✓ Caractères : char



Quelle est la différence ??

Déclaration en java :

```
Int a, b, c ;  
float x;  
char ch  
Booléens u, b;
```



Des erreurs ???!

Les premiers pas en Java

Les constantes

- Le mot clé final permet de déclarer une constante: la valeur ne doit pas être modifiée pendant l'exécution du programme.

Déclaration en java :

```
final int n=10, m ;  
n=n+1;
```

Le résultat de l'instruction est

Les premiers pas en Java

Les opérateurs et l'affectation

■ Les opérateurs mathématiques

Opérateur	Exemple	Equivalent à
=	Note = 18	Note = 18
+=	Note+ = 2	Note = Note + 2
-=	Note- = 2	Note = Note - 2
=	Note = 1.5	Note = Note * 1.5
/=	Note/ = 2	Note = Note / 2
%=	Note% = 2	Note = Note % 2
^=	Note^ = 2	Note = Note ^ 2

```
double nb1;  
float nb2;
```

nb1/nb2 =

Quel est le type du résultat de l'opération ?



```
float nb1;  
int nb2;
```

nb1 + nb2 =

Les premiers pas en Java

Les opérateurs et l'affectation

■ Priorité



■ Les instructions

```
final int N=50;  
short p=10;  
char c=2*N+3; // la variable c contient ....  
byte b= 10*N; // le résultat est ....
```

Quel est le résultat de chaque instruction ??



Les premiers pas en Java

Les opérateurs et l'affectation

- l'incrémentation et la décrémentation automatique

Incrémentation : $Nb++$ ou $++Nb$

Décrémentation : $Nb--$ ou $--Nb$



Quelle est la différence ??

```
i=2;  
j=i++;
```



Quelles sont les valeurs de i et j
dans les deux cas ?

```
i=2;  
j=++i;
```

Les premiers pas en Java

Les opérateurs et l'affectation

■ Autres opérateurs

Opérateur	
&&	L'opérateur ET logique
	L'opérateur OU logique
==	Opérateur d'égalité
!=	Opérateur de différence
!	L'opérateur négation

```
double nb1;  
double nb2;  
boolean res1, res2;
```

```
res1 = ( nb1==nb2)  
res2= ( nb1==nb2) && (nb1>10)
```

■ Les opérateurs de conversion (*cast*)

```
int i = 200;  
long l = (long)i; ← ...  
long l2 = (long)200; ← ...
```

Les premiers pas en Java

Les opérateurs et l'affectation

- Les opérateurs bits à bits

Opérateur	
&	L'opérateur ET
	L'opérateur OU
^	Opérateur OU exclusif
~	L'opérateur négation

Exemple :

Nb1	00000101
Nb2	00000011
Nb1 & Nb2
Nb1 Nb2
Nb1 ^ Nb2
~Nb1

Les premiers pas en Java

Les opérateurs et l'affectation

- Les opérateurs de décalage :
 - ✓ Manipules des bits,
 - ✓ Utilisables uniquement avec des types primitifs entiers,
 - ✓ Les opérateurs sont :
 - $Nb \ll n$: décalage vers la gauche du nombre Nb de n bits
 - $Nb \gg n$: décalage vers la droite du nombre Nb de n bits

Exemple :

$N=10000101$

$N \ll 2$ donne

`int N = 6;`

$N \gg 1$ donne

Les premiers pas en Java

Les boucles et les structures de contrôles

■ if... else...

✓ Syntaxe:

if (condition logique)

Instructions exécutées si la condition logique est vraie

else

Instructions exécutées si la condition logique est fausse

✓ Exemple 1:

```
If (moyenne >= 10)
```

```
    System.out.println(" l'étudiant a réussi");
```

```
else
```

```
    System.out.println(" l'étudiant doit repasser ses examens");
```

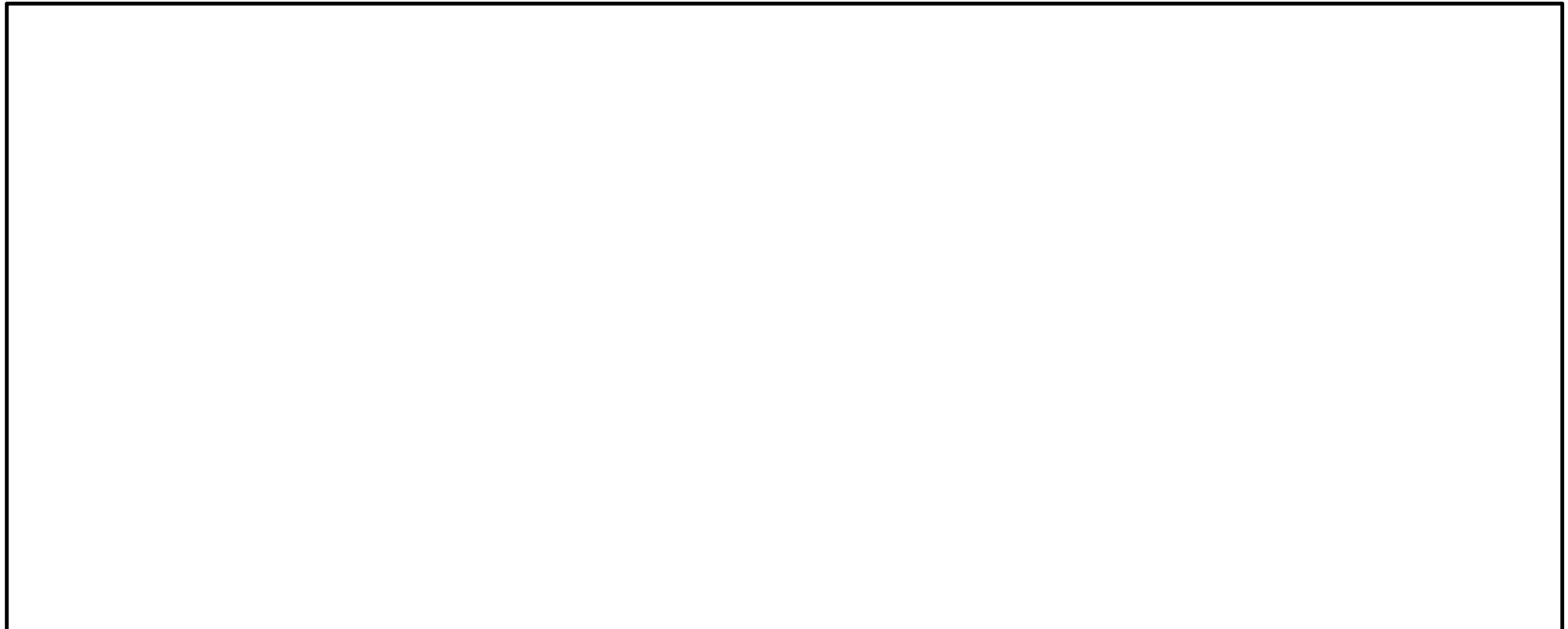
Les premiers pas en Java

Les boucles et les structures de contrôles

■ if... else...

✓ Exercice 1:

Ecrire la partie du programme Java qui vérifie si un entier n est paire ou impaire et affiche le résultat.



Les premiers pas en Java

Les boucles et les structures de contrôles

■ switch... case...default

✓ Syntaxe:

```
switch(expression)
{
    case constante_1: suite d'instructions; break;
    ...
    case constante_n: suite d'instructions; break;
    default : suite d'instruction;
}
```

Les premiers pas en Java

Les boucles et les structures de contrôles

■ switch... case...default

✓ Exercice 2:

Ecrire la partie du programme Java qui affiche:

- « Très bien » si la note est 'A',
- « Bien » si la note est 'B',
- « Insuffisant » si la note est 'C',
- un message d'erreur sinon



Les premiers pas en Java

Les boucles et les structures de contrôles

■ Les boucles itératives

✓ while ... :

```
while ( condition logique) {  
... // code a exécuter dans la boucle  
}
```

✓ do ... while :

```
do {  
Bloc d'instructions ;  
}  
while ( condition logique)
```

Les premiers pas en Java

Les boucles et les structures de contrôles

■ Les boucles itératives

✓ for... :

```
for (initialisation du compteur ; condition ; modification) {  
... // code a exécuter dans la boucle  
}
```

✓ Exemple 2:

```
for (i = 0 ; i > 10; i++)  
{  
System.out.println (i);  
}
```

Quel est le résultat de la boucle ?

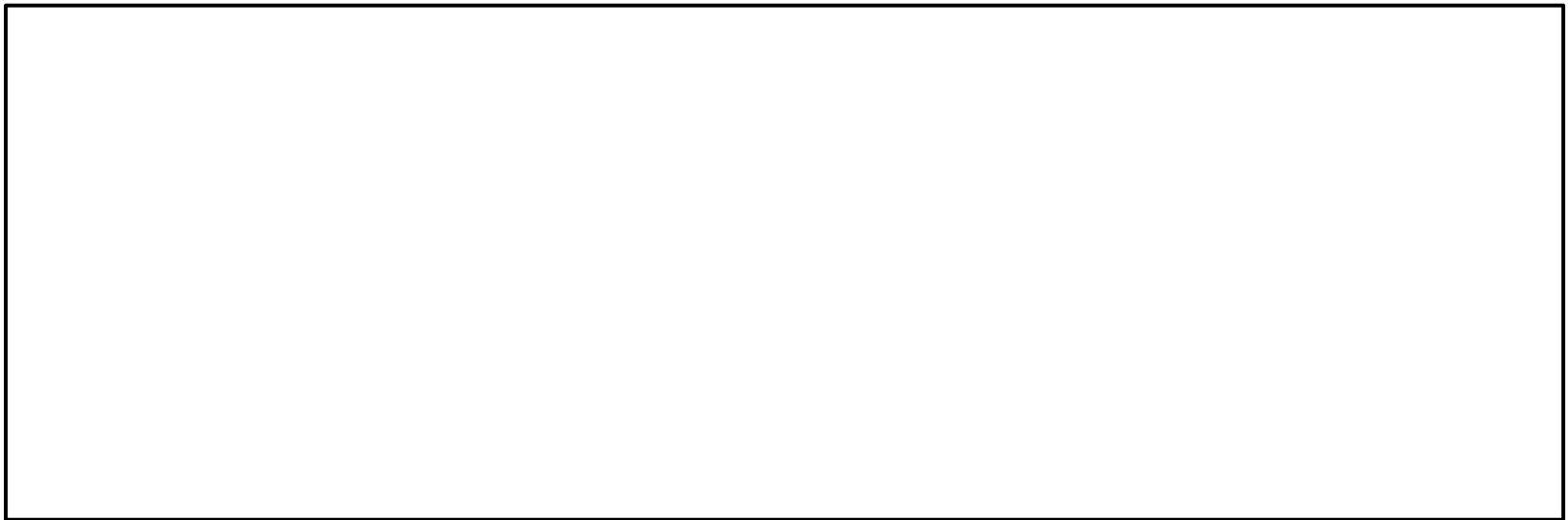


Les premiers pas en Java

Les boucles et les structures de contrôles

Exercice :

On dénomme nombre de Armstrong un entier naturel qui est égal à la somme des cubes des chiffres qui le composent. Ecrire un programme Java qui affiche de tels nombres. Ex : $153 = 1^3 + 5^3 + 3^3$



Les premiers pas en Java

Les boucles et les structures de contrôles

- break et continue
 - ✓ break : permet de sortir d'une boucle sans exécuter la suite des instructions
 - ✓ continue : arrête l'exécution de l'itération courante, et l'exécution reprend en début de boucle avec l'itération suivante

Les premiers pas en Java

Les tableaux

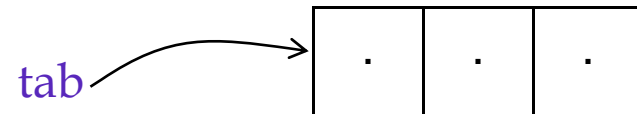
■ Tableau unidimensionnels

Déclaration : `int [] tab;`
ou `int tab[];`



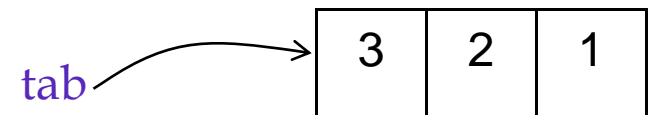
Pas de dimensions dans la déclaration

Dimensionnement: `int [] tab = new int [3]`



Allocation dans la mémoire selon le type du tableau

Initialisation: `tab [0]=3; tab [1]=2; tab [2]=1`



ou autrement , `int [] tab={3,2,1};`

Les premiers pas en Java

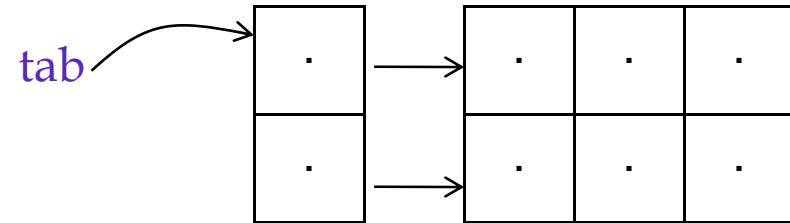
👉 Les tableaux

- Tableau multidimensionnels

Déclaration : `type [][] tab;`

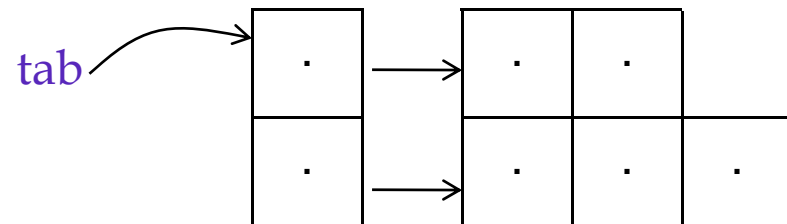


Dimensionnement: `tab = new type [2][3];`



ou aussi

`tab = new type [2];`
`tab [0] = new type[2];`
`tab [1] = new type[3]`

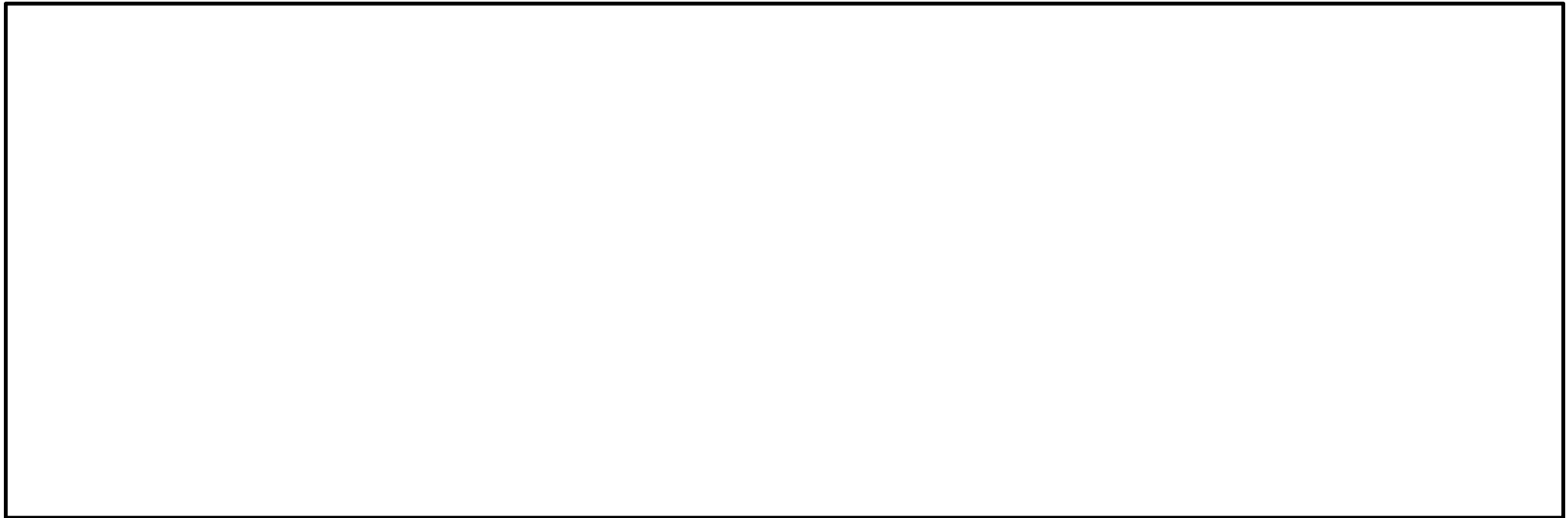


Les premiers pas en Java

Les tableaux

■ Exercice 3:

Soit T une matrice carrée de 3 lignes et 3 colonnes. Ecrire un algorithme qui affiche un message informant si la matrice est symétrique ou pas.

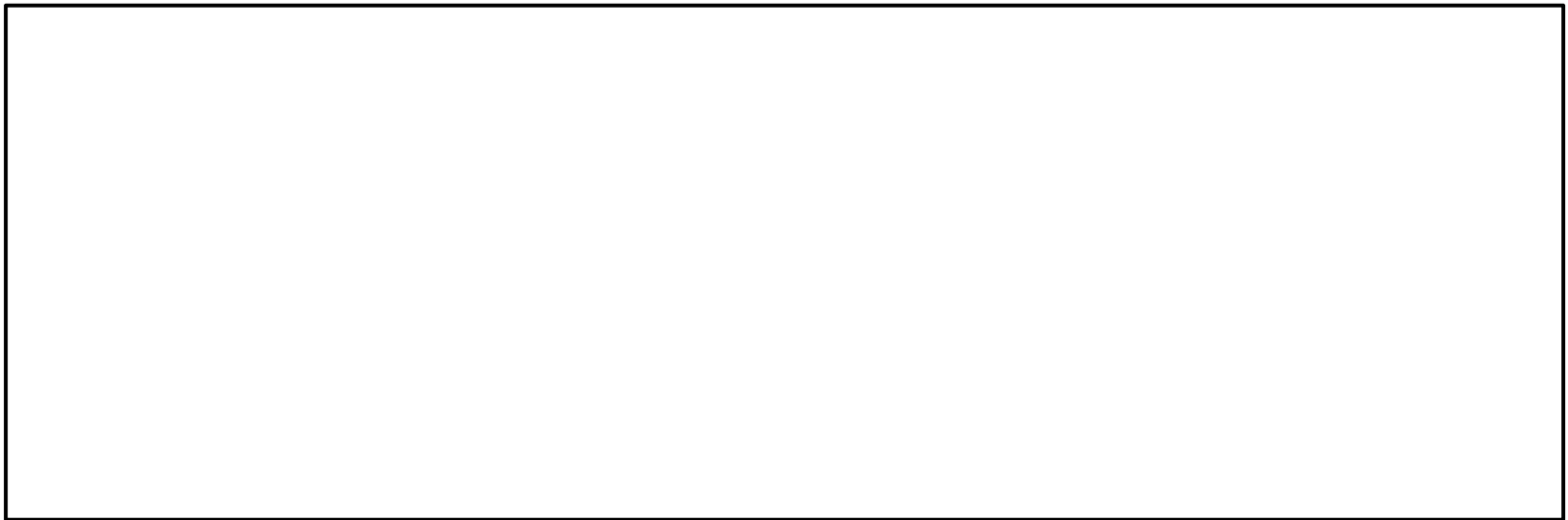


Les premiers pas en Java

Les tableaux

■ Exercice 4:

Ecrire un algorithme qui tri un tableau en utilisant la méthode de tri par insertion.



Les premiers pas en Java

Les vecteurs

- La classe vecteur permet de gérer des listes contenant plusieurs types
- La taille du vecteur est dynamique, contrairement à celle d'un tableau.

Déclaration : `Vector v = new Vector()`
 `Vector v = new Vector(5)`

Quel est la différence ?



Ajout d'élément dans le vecteur :

```
v.addElement(new Integer(1)); // Ajouter un premier élément
v.addElement(new Float(1.9999)); // Ajouter un autre élément
for (int i=2; i<10; i++) {
int lastInt = ((Number) v.lastElement()).intValue();
v.addElement(new Integer(i + lastInt)); } // Ajouter d'autres éléments
```

Les premiers pas en Java

Les commentaires


■ Importance des commentaires

- ✓ Clarté du code
- ✓ Réutilisation facile du code
- ✓ Génération automatique du javadoc ou le Help du programme développé

■ Deux types de commentaire

- ✓ Commentaire classique `/* ...*/` ou `// ...`
- ✓ Génération d'un fichier (HTML) de documentation `/** ...*/`

```
/**  
 * Ceci est un commentaire pour Javadoc  
 * @author Sabeur  
 * @version 2.0  
 */
```



Fin du cours

MCours.com