

Cours de Programmation Orientée Objet JAVA

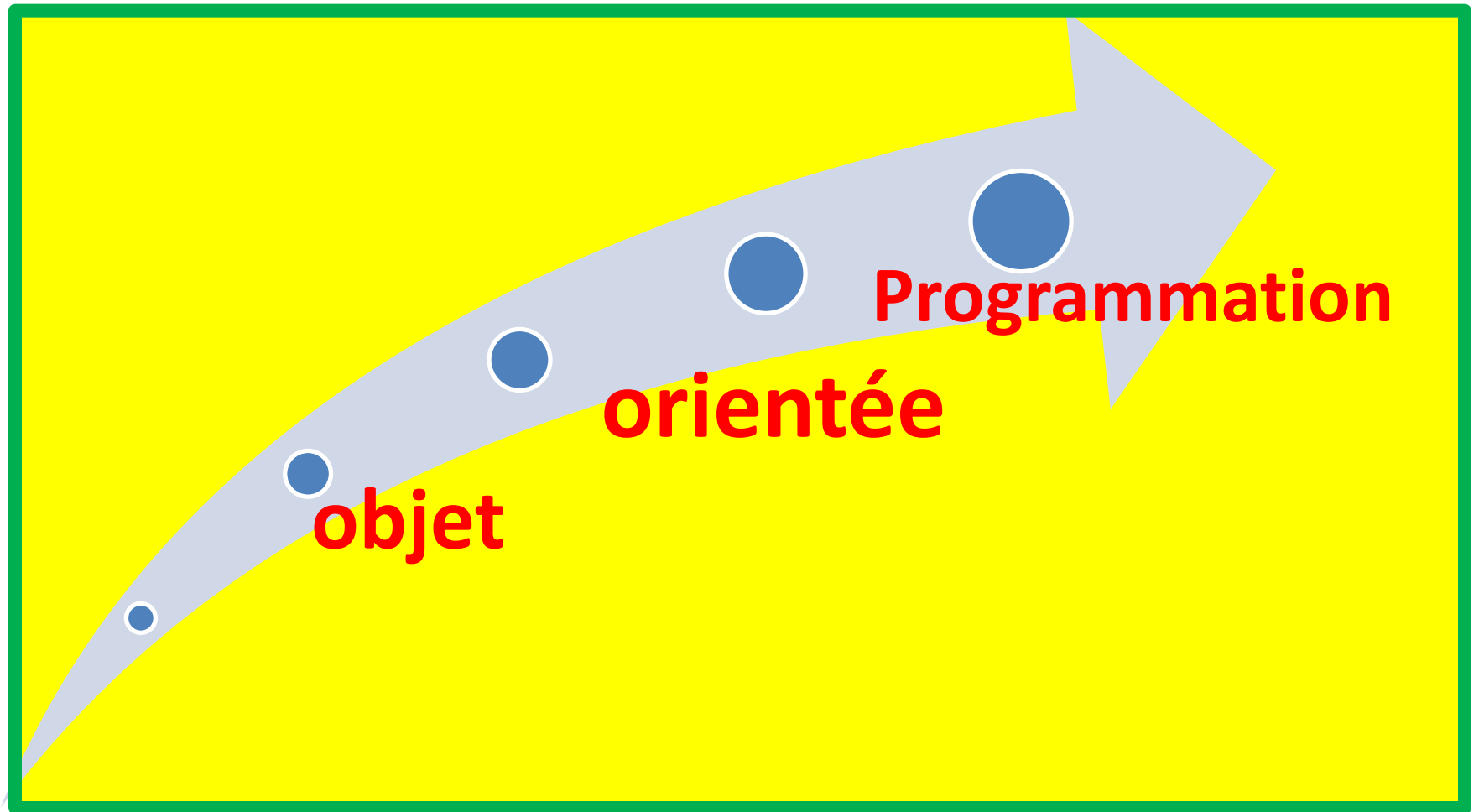
Demba SOW

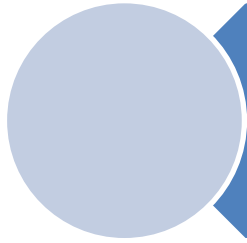
*Docteur en
Mathématiques et en
Cryptologie*

L.A.C.G.A.A.

F.S.T. / U.C.A.D.







Programmation Orientée Objet

Le concept de classe

Une classe est la description d'un ensemble de **données** et de **fonctions** regroupées au sein d'une même entité (appelée objet).

On peut définir une classe comme étant aussi une description abstraite d'un objet du monde réel.

Un objet sera défini comme étant une entité concrète ou abstraite du monde réel. Les objets contiennent des attributs et des méthodes. Chaque objet sera caractérisé par son jeu de données(on parlera d'**attributs** ou aussi de **champs**). Les fonctions qui agissent sur les données de l'objet sont aussi appelées **méthodes**.



Objet = identité + état + comportement

Chaque objet possède :

-une **identité** qui lui est propre :

Même si deux personnes ont des noms identiques, elles désignent deux individus distincts. Elles ont une identité distincte .

-un **état** : les informations qui lui sont propres

Il est défini par les valeurs associées à chacun des champs de la classe.
Pour une personne : **nom ,prénom, âge, sexe, race**

-un **comportement** : les méthodes applicables à l'objet

Pour une personne : **respirer, marcher, mourir**

Définition d'une classe

La méthode est **accessible** depuis
N'importe quel programme.

```
public class Point
```

```
{
```

```
    private int x ; // champ x d'un objet Point
```

```
    private int y ; // champ y d'un objet Point
```

```
    public void initialise ( int abs, int ord )
```

```
    { x = abs ;
```

```
      y = ord ;
```

```
    }
```

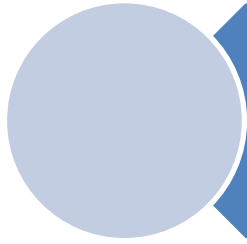
```
} // fin de la classe
```

Une méthode qui ne
fournit **aucun** résultat:
→ **void**

Les champs x et y ne sont visibles
qu'à l'intérieur de la classe et non à
l'extérieur : principe de
l'encapsulation des données.

Les données ne seront accessibles
que par l'intermédiaire de
méthodes prévues à cet effet.

Permet d'attribuer des valeurs
initiales aux champs de l'objet.
**NB : on verra que c'est au
constructeur d'effectuer ce travail.**



Remarques

Une **méthode** peut être déclarée **private** : dans ce cas elle n'est **visible** qu'à l'intérieur de la **classe** où elle est définie.

Pour pouvoir l'utiliser dans un autre programme, il faut nécessairement passer par une **méthode publique** de sa classe ayant l'appelée.

Il est fortement déconseillé de déclarer des champs avec l'attribut **public** , cela nuit à l'encapsulation des données.

Créer un objet = instancier une classe

```
int a = 10 ; // réservation de l'emplacement mémoire pour une variable de type int  
float x ; // réservation de l'emplacement mémoire pour une variable de type float  
Point a; // cette déclaration ne réserve pas d'emplacement pour un objet de type Point  
// mais simplement une référence à un objet de type Point.
```

La création d'un objet (on parle d'instanciation) se fait avec l'opérateur **new** :

```
new Point ( ) ; // crée un emplacement pour un objet de type Point et  
// fournit sa référence comme résultat
```

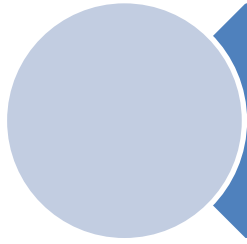
on pourra écrire :

```
a = new Point ( ) ; // crée un objet de type Point et on met sa référence  
// dans la variable a
```

référence
a



objet
Point



Manipulation d'objets

Une fois qu'une référence à un objet est convenablement initialisée, on peut appliquer n'importe quelle méthode à l'objet correspondant .

a.initialise (2,4) ; //appelle la méthode initialise du type
// Point en l'appliquant à l'objet de référence a, et
// en lui transmettant les arguments 2 et 4



Opérateur d'accès.

Petit programme de test

```
class Point01
```

```
{
```

```
    private int x ; // champ x d'un objet Point
```

```
    private int y ; // champ y d'un objet Point
```

```
    public void initialise ( int abs, int ord )
```

```
        { x = abs ;
```

```
          y = ord ;
```

```
        }
```

```
    } // fin de la classe Point01
```

```
public class TestPoint01
```

```
    { public static void main( String args [ ] )
```

```
        { Point01 a = new Point01 ( ) ;
```

```
          a.initialise( 10,12 ) ; // le champ x aura la valeur 10 et y 12
```

```
        }
```

```
    } // fin de la classe TestPoint01
```

On a deux classes dans le même fichier source. Seule la classe contenant le programme principal(main) doit être déclarée avec l'attribut **public**.

Le constructeur (1/3)

En Java, la création d'objet se fait par allocation dynamique grâce à l'opérateur **new** qui appelle une méthode particulière : le **constructeur**.

Dans l'exemple précédent, il n'y avait pas de constructeur mais plutôt c'était la méthode **initialise** qui se chargeait d'initialiser correctement les champs d'un objet de type Point. La démarche proposée suppose que l'utilisateur fera appel de cette méthode au moment opportun pour initialiser correctement un objet.

En fait un constructeur permet d'automatiser l'initialisation d'un objet.

Un constructeur est une **méthode** qui porte le **même nom** que le nom de la classe et qui est **sans valeur de retour**. Il peut disposer d'un nombre quelconque d'arguments.

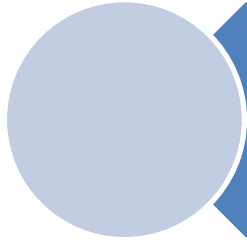
Le constructeur (2/3)

```
class Point02
{
private int x ; // champ x d'un objet Point
private int y ; // champ y d'un objet Point
public Point02 ( int abs, int ord ) // un constructeur à deux arguments
{ x = abs ;
  y = ord ;
}
public void deplace (int dx, int dy )
{ x += dx ; y += dy ;
}
public void affiche( )
{ System.out.println("Point de coordonnées" + x + " et " +y);
}
} //fin de la classe Point02
```

Le constructeur (3/3)

```
public class TestPoint02
{ public static void main( String args [ ] )
  { Point02 a = new Point02 ( 10, 12 ) ; // le champ x aura la valeur 10 et y 12
    a.affiche( ) ;
    a.deplace( 10,12 ) ;
    a.affiche( ) ;
  }
} // fin de la classe TestPoint02
```

Point de coordonnées : 10 et 12
Point de coordonnées : 20 et 24



Quelques règles sur les constructeurs

- ➡ Une classe peut disposer de plusieurs constructeurs: ils se différencieront par le nombre et le type de leurs arguments.
- ➡ Une classe peut ne pas disposer de constructeur; dans ce cas on peut instancier des objets comme s'il existait un constructeur par défaut sans arguments et ne faisant rien.
- ➡ Mais dès qu'une classe possède au moins un constructeur, ce constructeur par défaut ne peut plus être utilisé, dans le dernier exemple on ne pouvait pas faire :
Point a = new Point () ; // incorrect s'il y'a un constructeur
- ➡ Une classe peut disposer d'un constructeur sans arguments qui est bien différent du constructeur par défaut (appelé souvent *pseudo-constructeur*).
- ➡ Un constructeur peut appeler un autre constructeur de la même classe (A VOIR).
- ➡ Un constructeur peut être déclaré **public** ou **privé**.

Construction d'un objet

La construction et l'initialisation *des champs d'un objet* se font en 3 étapes:

- l'initialisation par défaut de tous les champs à une valeur "nulle",
- l'initialisation explicite lors de la déclaration du champ,
- l'exécution des instructions du constructeur.

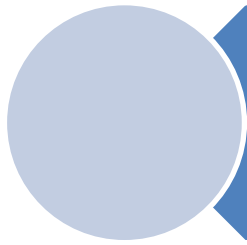
```
public class TestInit
{ private int nombre ;
  private int diviseur = 12 ;
  public TestInit( ) { nombre = 24 ;}
  public float diviser( )
  { return (float) nombre / diviseur ;
  }
}
```

nombre =24
diviseur =12

nombre =0
diviseur =12

nombre =0
diviseur =0

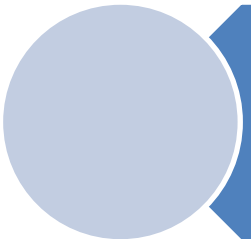
Résultat : 2.0



Valeur "nulle" suivant le type du champ

Type du champ	Valeur par défaut
booléen	false
char	caractère de code nul
entier(byte, short, int, long)	0
flottant(float, double)	0.f ou 0
objet	null

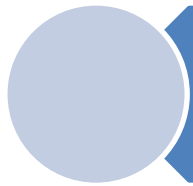
Initialisation par défaut des champs d'un objet



Champ déclaré avec l'attribut final

```
public class ChampFinal
{ private final int NOMBRE ; // initialisation différée
  private final float MAX ; // initialisation différée
  private final int DIVISEUR = 12 ; // valeur fixée à la déclaration
  public ChampFinal( int nbre)
  { NOMBRE = nbre ; //la valeur de NOMBRE dépendra de celle de nbre
    MAX = 20 ; // la valeur de MAX est fixée à 20 une seule fois.
  }
  public float diviser( )
  { return (float) NOMBRE / DIVISEUR ;
  }
}
```

ATTENTION: chaque objet possédera son propre champ NOMBRE, malgré que ce dernier est déclaré final.



Contrat et implémentation

■ Une bonne conception orientée objets s'appuie sur la notion de **contrat** , qui revient à considérer qu'une classe est caractérisée par un ensemble de services définis par :

- les **entêtes** de ses méthodes publiques ,
- le **comportement** de ses méthodes .

■ Le reste, c'est-à-dire les champs et les méthodes privés ainsi que le corps des méthodes publiques, n'a pas à être connu de l'utilisateur . Il constitue ce que l'on appelle souvent **l'implémentation** de la classe .

■ En quelque sorte, le **contrat** définit ce que fait la classe tandis que **l'implémentation** précise comment elle le fait .

■ Le grand mérite de **l'encapsulation** des données est de permettre au concepteur d'une classe d'en modifier l'implémentation sans que l'utilisateur n'ait à modifier les programmes qui l'exploitent .

Affectation et comparaison d'objets

Point a ;

a = new Point (12,10) ;

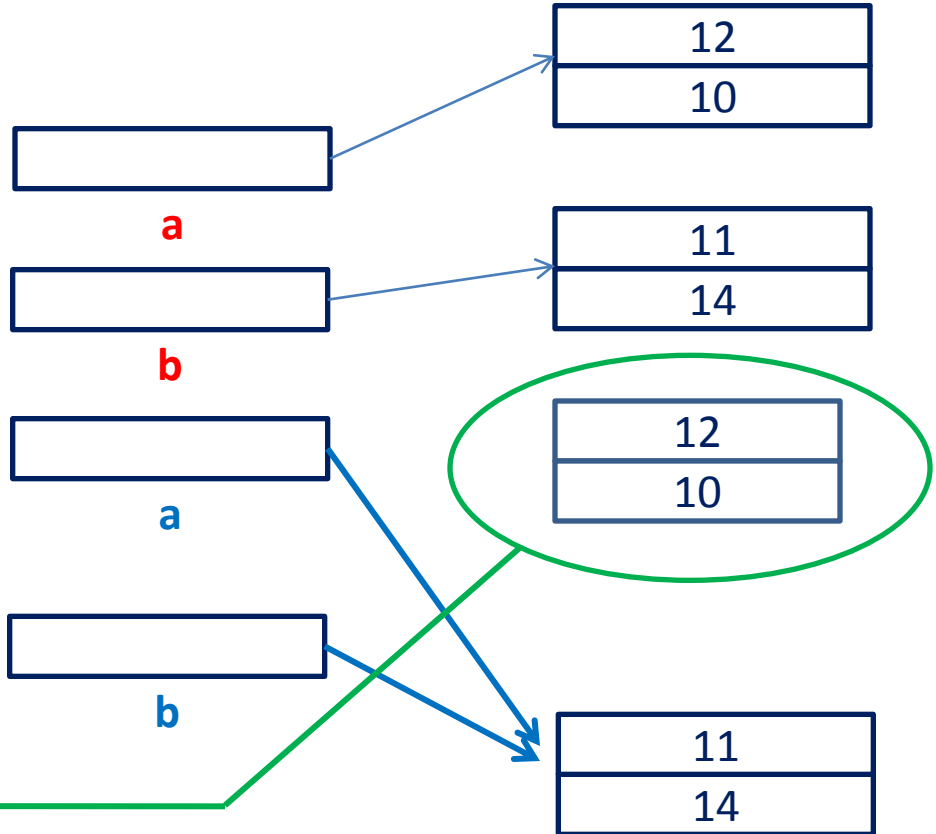
Point **b** = new Point (11, 14) ;

a = b ;

affectation d'objet

Sera candidat au
ramasse-miettes
s'il n'est plus référencé

Désormais **a** et **b** désignent le même objet.



Référence nulle: le mot clé null

```
class Point03
```

```
{  
    private int x ; // champ x d'un objet Point  
    private int y ; // champ y d'un objet Point  
    public Point03 ( int abs, int ord ) // un constructeur à deux arguments  
    { x = abs ;  
      y = ord ;  
    }  
    public Point03 coincide (Point03 p )  
    { Point03 t = null ; // t est locale donc nécessite de l'initialiser  
      if ((p.x == this.x) && ( p.y == this.y )) t = this;  
      else t = null;  
      return t ;  
    }  
} //fin de la classe Point03
```

Les variables locales de type objet
doivent toujours être initialisées

Champs et méthodes de classe: le mot clé static .

Champs de classe

Les champs de classe ou champs statiques existent en un *seul exemplaire pour* toutes les instances de la classe. On les déclare avec le mot clé **static** .

```
public class ChampStatic
{
    int n ;
    static int k ;
}
```

```
ChampStatic a = new ChampStatic( ) ;
ChampStatic b = new ChampStatic( ) ;
```

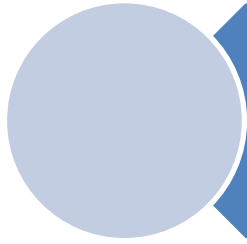
a.n →   ← b.n
a.k →  ← b.k

a.k et **b.k** peuvent être remplacés par **ChampStatic.k** .

Exemple d'utilisation de champs de classe

```
public class ChampStatic {  
    private static int nombreInstanceCrees; // champ static pour stocker le nombre  
    public ChampStatic() // d'objets créés  
    { nombreInstanceCrees++; // on incrémente de 1 à chaque création d'un objet  
    }  
    public void affiche()  
    {  
        System.out.println ("nombre d'objets créés :" + nombreInstanceCrees );  
    }  
    public static void main (String args[])  
    { ChampStatic a = new ChampStatic ( );  
      a.affiche ( );  
      ChampStatic b = new ChampStatic ( );  
      b.affiche ( );  
    }  
}
```

nombre d'objets créés : 1
nombre d'objets créés : 2



Méthodes de classe

Une méthode d'une classe ayant un rôle indépendant de toute instance de la classe doit être déclarée avec le mot clé **static** et elle ne pourra être appliquée à aucun objet de cette classe, contrairement aux méthodes d'instances.

L'appel de la méthode ne nécessitera que le nom de la classe.



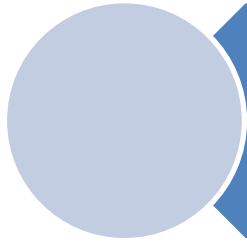
ATTENTION

Une méthode statique ne peut pas agir sur des champs usuels, c'est-à-dire non statiques.



Exemple d'utilisation de méthodes de classe

```
public class MethodeStatic {  
    private long n;  
    private static long nombreInstanceCrees; // champ static pour stocker le  
    public MethodeStatic( long k)           // nombre d'objets créés  
    {  
        nombreInstanceCrees++;  
        n = k ;  
    }  
    public void affiche ( )  
    {  
        System.out.println ("nombre d'objets créés :" + nombreObjet( ) );  
    }  
    public static long nombreObjet( )  
    {  
        return nombreInstanceCrees;  
    }  
}
```



Bloc d'initialisation statique

Remarque :

l'initialisation d'un champ statique se limite uniquement à :

- **l'initialisation par défaut,**
- **l'initialisation explicite éventuelle.**

Les blocs statiques sont souvent utilisés pour initialiser des variables complexes dont l'initialisation ne peut être faite par une simple instruction.

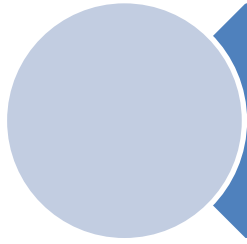
Les instructions n'ont accès qu'aux champs statiques de la classe.

Les instructions d'un bloc statique sont exécutées de façon automatique et une seule fois lorsque la classe est chargée.

Exemple d'utilisation de bloc statique

```
public class BlocStatic {  
    private double solde;  
    private static int [ ] tab ;  
    static { tab = new int[10]; // bloc d'initialisation délimité par des accolades  
        for ( int i = 0; i < tab.length; i++)  
            tab[ i ] = i + 1;  
    } // fin du bloc static  
    public BlocStatic (double solde) {  
        this.solde = solde;  
    }  
    public static void main(String[] args) {  
        BlocStatic a = new BlocStatic( 12000 );  
        for (int i=0;i < tab.length; i++)  
            System.out.print (tab[i]+" ");  
    }  
} // fin de la classe
```

1 2 3 4 5 6 7 8 9 10



Sur-définition de méthodes

La *surdéfinition* de méthodes signifie qu'un même nom de méthode peut être utilisé plusieurs fois dans une même classe. Dans ce cas, **le nombre et/ou le type des arguments** doit nécessairement changer.

On peut parler indifféremment de *surdéfinition*, *surcharge* ou *overloading* (en Anglais).

Exemple de sur-définition de méthode

```
public class ExempleSurdefinition {  
    private int x ;  
    private int y ;  
    public ExempleSurdefinition (int abs, int ord ) { x=abs; y=ord;  
    }  
    public void deplace (int dx, int dy) { x += dx ; y += dy ; }  
    public void deplace (int dx ) { x += dx ;  
    }  
    public void affiche(){ System.out.println(" Point de coordonnees :"+ x+ " "+y);}  
    public static void main(String[] args) {  
        ExempleSurdefinition ex = new ExempleSurdefinition(10,10);  
        ex.deplace ( 10 ); // appel de deplace ( int )  
        ex.affiche ( );  
        ex.deplace( 10, 10 ); // appel de deplace (int , int )  
        ex.affiche( );  
    }  
}
```

Point de coordonnees : 20 10
Point de coordonnees : 30 20



Il peut y avoir des cas d'ambiguïté :

```
public void deplace (int dx, short dy)
    { x += dx ;
      y += dy ;
    }
public void deplace (short dy, int dx )
    { x += dx ;
    }
```

avec : **ExempleSurdefinition a = new**
ExempleSurdefinition(10, 12) ; short b;

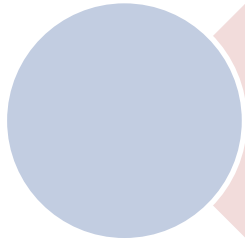
l'appel a.deplace(b ,b) causera une erreur.

Sur-définition de constructeurs

Les constructeurs peuvent être sur-définis comme toute autre méthode.

```
public class Individu {  
    private String nom;  
    private String prenom;  
    private Compte c;  
    /* constructeur à deux arguments */  
    public Individu ( String lenom, String leprenom ) {  
        nom = lenom;  
        prenom = leprenom;  
    }  
    /* constructeur à trois arguments */  
    public Individu (String lenom, String leprenom, Compte cp) {  
        nom = lenom;  
        prenom = leprenom;  
        c = cp;  
    }  
}
```

Attribut de type objet. Il doit exister obligatoirement une classe **Compte**.



Autoréférence : **this**

L'utilisation de **this** est très pratique dans l'écriture des méthodes et surtout des constructeurs.

```
// noms de champs et noms d'attributs  
//différents  
public Point ( int abs, int ord ) {  
    x = abs;  
    y = ord;  
}
```



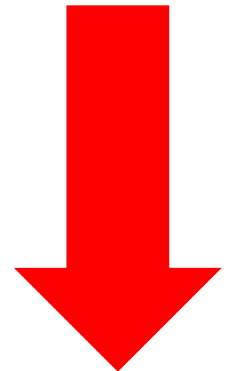
```
// noms de champs et noms d'attributs  
//identiques  
public Point ( int x, int y ) {  
    this.x = x;  
    this.y = y;  
}
```

Appel d'un constructeur dans un autre constructeur. (1/2)

Un constructeur peut appeler un autre constructeur de la même classe en utilisant le mot clé **this**. L'objectif majeur est la simplification du code et aussi pour des problèmes de sécurité.

```
public class Individu {  
    private String nom;  
    private String prenom;  
    private Compte c;  
    public Individu ( String lenom, String leprenom ) {  
        nom = lenom;  
        prenom = leprenom;  
    }  
    public Individu (String lenom, String leprenom, Compte c1) {  
        nom = lenom;  
        prenom = leprenom;  
        c = c1;  
    }  
}
```

Cette classe peut être écrite de façon plus sophistiquée comme suit ...

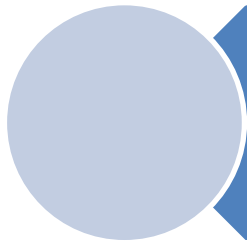


Appel d'un constructeur dans un autre constructeur. (2/2)

```
public class Individu2 {  
    private String nom;  
    private String prenom;  
    private Compte c;  
    public Individu2 ( String nom, String prenom ) {  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
    public Individu2 (String nom, String prenom, Compte c) {  
        // appel du constructeurs a deux arguments  
        this (nom, prenom);  
        this. c = c;  
    }  
}
```

ATTENTION :

L'appel `this (...)` doit nécessairement être la première instruction du constructeur appelant.

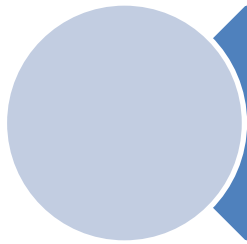


Objet membre

Objet membre = référence à un objet

```
public class Point {  
    private int x;  
    private int y;  
    public Point (int abs, int ord) {  
        x = abs;  
        y = ord;  
    }  
    public void affiche ( )  
    {  
        System.out.println(" Point :" +x +" " +y);  
    }  
}
```

```
public class Cercle {  
    private double r; //rayon du cercle  
    private Point p; // objet membre  
    public Cercle (double r, int x, int y) {  
        this.r = r;  
        p = new Point (x, y);  
    }  
    public void affiche ( )  
    {System.out.println("Cercle de rayon :"+r);  
        System.out.print(" et de centre:" );  
        p.affiche();  
    }  
}
```



Les classes internes

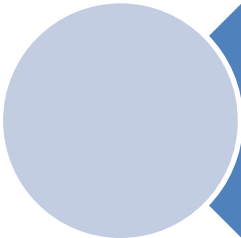
Une classe est dite interne lorsque que sa définition est située à l'intérieur de la définition d'une autre classe. **Les classes internes (inner classes) peuvent être situées à différent niveau d'une classe normale.**

Il existe quatre types de classes imbriquées :

- **les classes internes simples**, définies au niveau des classes,
- **les classes internes statiques**, représentant une classe de sommet intérieure,
- **les classes locales**, définies au niveau des méthodes,
- **les classes internes anonymes**, définies au niveau d'une instance.

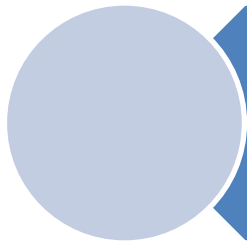
Les classes internes sont particulièrement utiles pour :

- *permettre de définir une classe à l'endroit ou une seule autre en a besoin*
- *définir des classes de type adapter (essentiellement à partir du JDK 1.1 pour traiter des évènements émis par les interfaces graphiques)*
- *définir des méthodes de type callback d'une façon générale.*



Classes internes simples(1/5)

```
package essai01;
public class ClasseParente {
    private int x = 10, static int y = 20;
    public int addition ( ) { return (x + y); }
    public class ClasseInterne    //DEBUT CLASSE INTERNE
    { static int p = 20;          //erreur de compilation,
      static final int k = 12;    //constante statique
      public int multiplier()
      {return x*y + addition( ); }
    } //FIN CLASSE INTERNE
    public static void main(String [] args) {
        ClasseParente ob_out = new ClasseParente();
        //ClasseInterne ob_in0 = new ClasseInterne(); IMPOSSIBLE
        ClasseInterne ob_in = ob_out.new ClasseInterne();
        System.out.println (ob_in.multiplier());
        // System.out.println (ob_out.multiplier()); //ERREUR
        // System.out.println (ob_in.addition ( )); IMPOSSIBLE
    }
}
```



Classes internes simples(2/5)

Quelques remarques importantes:

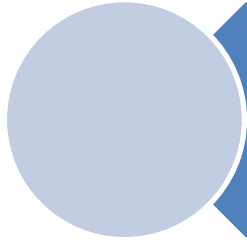
Une classe interne peut être déclarée avec n'importe quel modificateur d'accès (***public***, ***protected***, ***par défaut*** ou ***private***) et les modificateurs ***abstract***, ***final***, ***static***.

Elles sont membres à part entière de la classe qui les englobe et peuvent accéder à tous les membres de cette dernière.

Les classes internes ne peuvent pas être déclarées à l'intérieur d'initialiseurs statiques (blocs statiques).

Les classes internes ne doivent pas déclarer de membres statiques, sauf s'ils comportent le modificateur ***final***, dans le cas contraire, une erreur de compilation se produit. Toutefois, **les membres statiques de la classe externe peuvent être hérités sans problème par la classe interne.**

Les classes imbriquées sont **capables d'accéder à toutes les variables et méthodes de la classe parente**, y compris celles déclarées avec un modificateur ***private***.



Classes internes simples(3/5)

On retient:

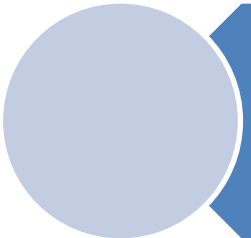
La notation particulière:

ClasseInterne `ob_in` = `ob_out.new` ClasseInterne();

spécifie que l'objet créé est une instance de la classe interne associée à l'objet résultant de l'instanciation d'une classe de plus haut niveau.

L'instanciation de la classe interne passe obligatoirement par une instance préalable de la classe d'inclusion.

La classe parente est d'abord instanciée, puis c'est au tour de la classe interne de l'être par l'intermédiaire de l'objet résultant de la première instance.



Classes internes simples(4/5)

Il est possible **d'utiliser une méthode de la classe parente pour créer directement une instance de la classe interne**. Toutefois, lors de l'appel de la méthode, il sera nécessaire de créer une instance de la classe d'inclusion.

```
package essai0;
public class ClasseParente02 {
    private int x = 10, int y = 20;
    public int addition ( )
    { ClasseInterne02 obj_in = new ClasseInterne02( );
      return (x + y)+ obj_in .multiplier ( );
    }
    public class ClasseInterne02
    { public int multiplier ( )
      { return x*y ;
      }
    }
}
```

Classes internes simples(5/5)

```
public class ClasseParente03 {  
    private int x = 10, int y = 20 ;  
    public int addition( )  
    { ClasseInterne03 obj_in= new  
      ClasseInterne03(10,10);  
      return (x + y)+ obj_in.multiplier();  
    }  
    public class ClasseInterne03  
    { private int x = 12; private int y = 14;  
      public ClasseInterne03 (int x, int y)  
      { this.x = x + ClasseParente03.this.x;  
        this.y = y + ClasseParente03.this.y;  
      }  
    }  
    public int multiplier( )  
    { return x*y ;  
    }  
}}
```

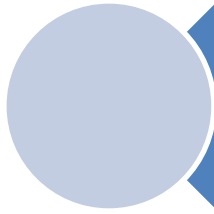
Parfois, il peut être nécessaire de distinguer les variables situées dans les classes interne et externe.

Classes internes statiques

Elles sont membres à part entière de la classe qui les englobe et peuvent accéder uniquement aux membres statiques de cette dernière.

```
public class Parente04 {  
    private static int x = 1, y = 2;  
    private int z = 3;  
    public int addition ( ) { return x + y + z;}  
    public static int multiplier( ) { return x*y;}  
    public static class Interne04{  
  
        private static int k = 1;  
        private int p = 2;  
        public void diviser()  
        { System.out.println (new Parente04( ).addition ( )/p+x+y);  
        }  
        public static void imprimer(){  
            System.out.println ( multiplier ( )/ x+y+k );} }  
        public static void main(String [ ] args) {  
            Parente04.Interne04( ).imprimer ( );  
            new Parente04.Interne04( ).diviser ( )} }  
}
```

Les classes internes statiques peuvent accéder à l'ensemble des membres statiques de leur classe parente, à l'instar des méthodes de classe.



Classes locales

Une classe locale est définie à l'intérieur d'une méthode ou un bloc, et agit librement et essentiellement au sein de cette dernière.

Elles peuvent être static ou non.

Il n'est possible de déclarer des classes locales, dont la portée est limitée au bloc, qu'avec les modificateurs *final* ou *abstract*. Les modificateurs suivants : *public*, *protected*, *private* et *static*, sont **interdits**.

Les données membres d'une classe externe peuvent être accédés par la classe locale.

Seules les variables locales et les paramètres de la méthode d'inclusion, déclarées avec le modificateur *final*, peuvent être exploitées par les classes internes locales, sinon une erreur se produit lors de la compilation. De plus, ces variables doivent être impérativement initialisées avant leur emploi dans la classe locale.

Classes locales

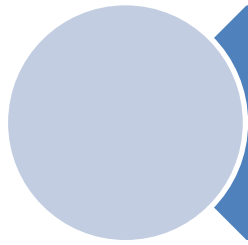
```
public class ClasseExterne {  
    private int x, y; private static int z;  
    public void addition( int p){  
        final int k = 9;  
        int u = 121; // inutilisable dans ClasseLocale  
        class ClasseLocale {  
            boolean verif()  
            {if (x+ y+ k == z) return true;  
            else return false;  
            }  
        }  
    } // fin bloc de méthode  
}
```

p et **u** ne peuvent pas être utilisés dans la ClasseLocale. Par contre, **k** est déclarée finale donc on peut l'utiliser.

REMARQUE:

Lorsqu'une classe est déclarée dans une méthode **statique**, alors les variables d'instances de la classe externe ne sont plus accessibles pour la classe imbriquée.

L'utilisation d'une classe locale ne dépend pas de l'instanciation d'une classe externe.



Classes anonymes

Les classes anonymes (anonymous classes) sont déclarées immédiatement après l'expression d'instanciation d'une classe, permettant directement d'étendre ou d'implémenter respectivement la classe ou l'interface instanciée.

Elles sont définies et instanciées à la volée sans *posséder de nom*.

```
new Classe ([Liste d'arguments]) { // Instructions de la classe anonyme... };
```

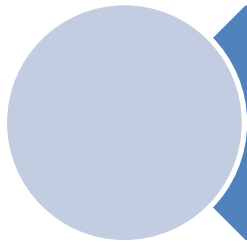
```
new Interface ( ) { // Instructions de la classe anonyme... };
```

Les classes anonymes **obéissent aux mêmes restrictions** que les **classes locales** et de plus, ne peuvent **ni être abstraites** (*abstract*) **ni être statiques** (*static*).

Par contre, **elles portent toujours implicitement le modificateur *final***.

En fait, **aucun modificateur n'est permis** dans une déclaration de classe anonyme

On verra l'utilité des classes anonymes en programmation événementielle.



Gros plan sur les packages

Un package regroupe un ensemble de classes sous un même espace de nommage.

Les noms des packages suivent le schéma : **name.subname...**

Une classe **Watch** appartenant au package **time.clock** doit se trouver obligatoirement dans le fichier **time/clock/Watch.class**.

Les packages permettent au compilateur et à la JVM de localiser les fichiers contenant les classes à charger.

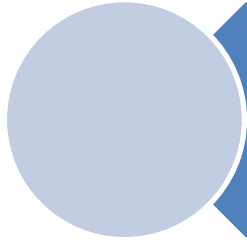
L'instruction package indique à quel paquetage appartient la ou les classe(s) de l'unité de compilation (le fichier).

Les répertoires contenant les packages doivent être présents dans la variable d'environnement **CLASSPATH**.

En dehors du package, les noms des classes sont: **packageName.className**.

L'instruction **import packageName** permet d'utiliser des classes sans les préfixer par leur nom de package.

Les API sont organisées en package (**java.lang, java.io, javax.swing,....**)



Droits d'accès et paquetage

En Java, il y a quatre types de droits aux méthodes et aux champs d'un objet d'une classe. Autrement dit, la portée de la visibilité des méthodes et champs est assurée par les mots clés: **private**, **protected**, **vide (droit de paquetage)**, et **public**.

Nous décrivons ici, la notion de droit d'accès parallèlement à la notion de paquetage.

Nous séparerons la visibilité des champs et celle des méthodes.

Paquetage et visibilité des champs (encapsulation des membres)

```
class c1 {  
    public    int a;  
            int b;  
    protected int c;  
    private  int d; }
```

package A

```
class c2 extends c1 {  
    ....  
    a b c  
}
```

```
class c3 {  
    ....  
    a b c  
}
```

```
class c4 extends c1{  
    ....  
    a c  
}
```

package B

```
class c5 {  
    ....  
    a  
}
```

Exemple d'accès aux membres

sow/classes/ graph/2D/ Circle.java

```
package graph.2D;  
public class Circle{  
.....  
}
```

sow/classes/ graph/3D/ Sphere.java

```
package graph.3D;  
public class Sphere{  
.....  
}
```

sow/classes/ testpackage/ MainClass.java

```
package testpackage;
```

```
import graph.2D.*;
```

```
public class MainClass{
```

```
public static void main (String args []){
```

```
graph.2D.Circle c1= new graph.2D.Circle(50);
```

```
Circle c2 = new Circle (80);
```

```
graph.3D.Sphere s1 = new graph.3D.Sphere (100); //OK
```

```
Sphere s2 = new Sphere (50); // error: class testpackage.Sphere not found
```

import graph.3D.Sphere, *//nécessaire*

Paquetage et visibilité des méthodes

```
class c1 {  
    public    int f();  
            int g();  
    protected int h();  
    private  int k();  
}
```

package A

```
class c2 extends c1 {  
    .... f() g() h()  
}
```

```
class c3 {  
    f() g() h()  
}
```

```
class c4 extends c1 {  
    .... f() h()  
}
```

package B

```
class c5 {  
    .... f()  
}
```