

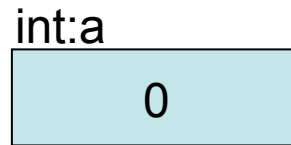
Notions de base

Notions de base

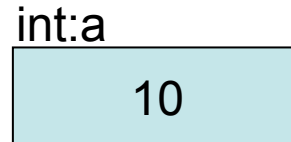
- Chaque Variable, Constante (valeur) est **typée**
- Types de bases:
 - int, byte, float, boolean, etc.
 - String, ...

Comprendre comment ça fonctionne

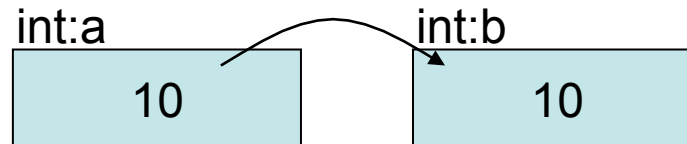
`int a;`



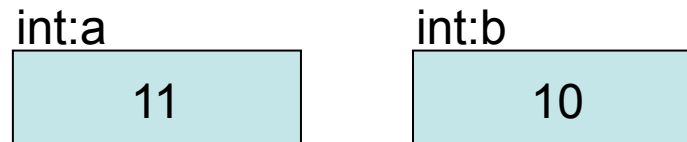
`a = 10;`



`int b = a;`



`a++;`



Classe comme type (plus de détails plus tard)

- Type objet défini par le programmeur
 - Définir une classe (type)
 - Référence de ce type
 - E.g.

```
public class Circle {...}
```

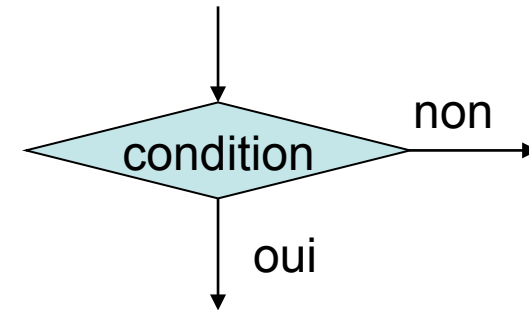
```
public class Rectangle {...}
```

```
Circle soleil; //déclaration de la var.
```

```
Rectangle box;
```
- ```
soleil = new Circle(); //créer une instance
```
- ```
box = soleil; // Erreur: types incompatibles
```
- ```
box = new Circle(); //Erreur
```

# Contrôle du programme

- if *condition* then *statement*  
else *statement* ;



- while *condition statement* ;
- do *statement* while (*condition*); //fait au moins une fois
- for (*Initialization; condition; update*) *statement* ;

e.g.

```
for (i=1; i<n; i++)
```

```
{
```

```
 double interest = balance * rate / 100;
```

```
 balance = balance + interest;
```

```
}
```

## Choix : if ...else switch

```
package tp1;
public class Exemple_If_Else{
int final MIN = 100;
int final Max = 1000;
int solde ;
public static void main(String args [])
{ if (solde < MIN)
 System.out.println("solde insuffisant");
 else
 if (solde == MAX)
 System.out.println("solde suffisant");
}
}
```

```
package tp1;
public class Exemple_Switch{
int final MIN = 100;
int final Max = 1000;
int choix , solde;
public static void main(String args [])
{ switch(choix)
 { case 1: solde = MIN;
 System.out.println("solde insuffisant");
 break;
 case 2: solde = MAX ;
 System.out.println("solde suffisant");
 break;
 default : break
 } }}
}
```

## Syntaxes : if ...else switch

**if (condition)**

**instruction\_1**

**[ else**

**instruction\_2 ]**

Condition booléenne ( true / false )

Expressions quelconques

Les **crochets** renferment des instructions facultatives.

**switch (expression)**

```
{ case constante_1 : [suite_d'instruction_1]
 case constante_2 : [suite_d'instruction_2]

 case constante_n : [suite_d'instruction_n]
 [default : suite_d'instructions]
}
```

Expression de type **byte**, **char**, **short** ou **int** .

Expression **constante** d'un type compatible par affectation avec le type de **expression**

## L'instruction do while

```
package tp1;
import java.util.Scanner ; // importation de classe de l' API
public class Exemple_Do_While{
public static void main (String args [])
{ Scanner clavier = new Scanner (System.in) ;
do
 { System.out.println ("saisir un entier strictement positif ") ;
 n = clavier.nextInt () ; // saisir à partir du clavier
 if (n < 0) System.out.println ("la saisie est invalidée: recommencez") ;
 }
while ((n < 0) || (n == 0));
}}
```

**do instruction**

**Expression quelconque**

**while (condition) ;**

**Condition booléenne**



## L'instruction while

```
package tp1;
public class Exemple_While{
public static void main(String args [])
{ while (n <= 0)
 { System.out.println ("saisir un entier strictement positif ");
 n = clavier.nextInt(); // saisir à partir du clavier
 if (n < 0) System.out.println ("la saisie est invalidée: recommencez");
 }
}
}
```

**while (condition) ;**

**Condition booléenne**

**instruction**

**Expression quelconque**

## L'instruction for

```
package tp1;
public class Exemple_For{
public static void main (String args [])
{ int tab [] = new int [100] ; // tableau d'entiers de taille 100
 for(int i = 0 ; i < 100 ; i ++)
 {

 tab [i] = i + 1;

 }
}
```

**for ( [initialisation] ;[condition] ; [incrémentation])**

**instruction**

## Branchement inconditionnel break / continue

Ces instructions s'emploient principalement au sein des boucles.

**break**

Elle sert à interrompre le déroulement de la boucle, en passant à l'instruction suivant la boucle.

```
package tp1;
public class Exemple_Break{
public static void main (String args [])
{ int tab [] = new int [10] ; // tableau d'entiers de taille 10
 for(int i = 0 ; i < 10 ; i ++)
 { if (i == 5) break ; // initialiser seulement les 5 premiers elts du tableau
 tab [i] = i + 1 ;
 } // ← le break nous branche à la sortie du for pour continuer
 for (int i = 0 ; i < 10 ; i ++)
 System.out.println (" éléments du tableau:" + " " + tab [i]);
}
```

|                      |   |   |   |   |   |   |   |   |   |   |
|----------------------|---|---|---|---|---|---|---|---|---|---|
| éléments du tableau: | 1 | 2 | 3 | 4 | 5 | 0 | 0 | 0 | 0 | 0 |
|----------------------|---|---|---|---|---|---|---|---|---|---|

## break avec imbrication de boucles

```
package tp1;
public class Essai_Break_Imbr {
public static void main (String args [])
{ int tab [] = new int [10]; // tableau d'entiers de taille 100
 for(int i = 0 ; i < 10 ; i ++)
 {
 for (int j =i;j <10;j++)
 { if (j == 3 || j == 4 || j == 5) break ;
 tab [j] = j+1;
 } // ← le break branche ici
 }
 for (int i = 0 ; i < 10 ; i ++)
 System.out.println (" éléments du tableau:" + " " +tab [i]);
}
```

|                      |   |   |   |   |   |   |   |   |   |    |
|----------------------|---|---|---|---|---|---|---|---|---|----|
| éléments du tableau: | 1 | 2 | 3 | 0 | 0 | 0 | 7 | 8 | 9 | 10 |
|----------------------|---|---|---|---|---|---|---|---|---|----|

En cas de boucles imbriquées, l'instruction break fait sortir uniquement de la boucle la plus interne.

## break avec étiquette

```
package tp1;
public class Essai_Break_Etiq {
public static void main (String args [])
{ int tab [] = new int [10]; // tableau d'entiers de taille 100
 repeat: for(int i = 0 ; i < 10 ; i ++)
 {
 for (int j =i;j <10;j++)
 { if (j == 3 || j == 4 || j == 5) break repeat;
 tab [j] = j+1;
 }
 } // ← cette fois le break branche ici
 for (int i = 0 ; i < 10 ; i ++)
 System.out.println (" éléments du tableau:" + " " +tab [i]);
}
```

Étiquette : pour sortir  
de deux boucles imbriquées

|                      |   |   |   |   |   |   |   |   |   |   |
|----------------------|---|---|---|---|---|---|---|---|---|---|
| éléments du tableau: | 1 | 2 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|----------------------|---|---|---|---|---|---|---|---|---|---|

## Continue ordinaire

### continue

L'instruction continue permet de passer *prématurément* au tour de boucle suivant.

```
package tp1;
public class Exemple_Continue_Ord{
public static void main (String args [])
{ int tab [] = new int [10] ; // tableau d'entiers de taille 10
 for(int i = 0 ; i < 10 ; i ++) // ← ici
 { if (i == 5) continue ; // on poursuit la boucle for
 tab [i] = i + 1 ;
 }
 for (int i = 0 ; i < 10 ; i ++)
 System.out.println (" éléments du tableau:" + " " + tab [i]);
}
}
```

éléments du tableau: 1 2 3 4 5 0 6 7 8 9 10

## Continue avec étiquette

```
package tp1;
public class Essai_Continue_Etiq {
public static void main(String args [])
{ int tab [] = new int [10]; // tableau d'entiers de taille 100
 again: for(int i = 0 ; i < 10 ; i ++) // ← cette fois le continue branche ici
 {
 for (int j =i;j <10;j++)
 { if (j == 3 || j == 4 || j == 5) continue;
 tab [j] = j+1;
 }
 } for(int i = 0 ; i < 10 ; i ++)
 System.out.println(" éléments du tableau:" + " " +tab [i]);
}
```

**ATTENTION:** on ne mentionne pas le nom de l' étiquette avec continue.

|                      |   |   |   |   |   |   |   |   |   |    |
|----------------------|---|---|---|---|---|---|---|---|---|----|
| éléments du tableau: | 1 | 2 | 3 | 0 | 0 | 0 | 7 | 8 | 9 | 10 |
|----------------------|---|---|---|---|---|---|---|---|---|----|

# boolean et test

- Combinaison de conditions (test booléens)

- &&      ET
- ||        OU
- !        NON
- !=       DIFFERENT
- <        INFÉRIEUR
- <=      INFÉRIEUR OU ÉGALE
- >        SUPÉRIEUR
- >=      SUPÉRIEUR OU ÉGALE

e.g. if (0 < amount && amount < 1000) . . .



# Rappel: Classe

- Classe = modèle (moule) à objet
- Définition d'une nouvelle classe
  - Identificateur (nom de classe)
  - Attributs (ou variables)
  - Méthodes (dont constructeur)

- E.g.

```
public class Rectangle
{
 public Rectangle(int nx, int ny, int nwidth, int nheight)
 {
 x=nx; y=ny; width=nwidth; height=nheight;
 }
 int x, y, width, height;
 public int getWidth() {return width; }
}
```

The diagram illustrates the components of the provided Java code snippet and their roles:

- Nom de classe**: Points to the class name `Rectangle`.
- constructeur**: Points to the constructor method `public Rectangle(int nx, int ny, int nwidth, int nheight)`.
- attributs**: Points to the instance variables `int x, y, width, height;`.
- méthodes**: Points to the method `public int getWidth() {return width; }`.

## Définition d'une classe

```
package tp1;
public class Point
{
 private int x ; // champ x d'un objet Point
 private int y ; // champ y d'un objet Point
 public Point (int abs, int ord)
 { x = abs ;
 y = ord ;
 }
} // fin de la classe
```

**private**: Les champs x et y ne sont visibles q'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 (accesseurs).

**Permet d'attribuer des valeurs initiales aux champs de l'objet.**  
**Cette méthode est ce qu'on appelle un constructeur.**

## 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.

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 P
 // mais simplement une référence à un objet de type Point.
```

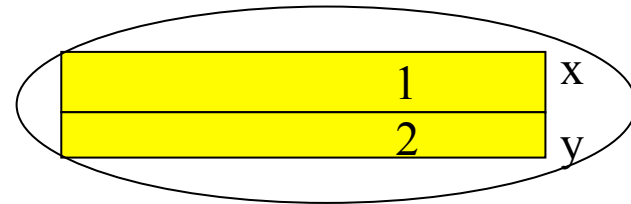
La création d'un objet (on parle d'instanciation ) se fait avec l'opérateur **new** :

on pourra écrire :

```
a = new Point (1, 2) ; // crée un emplacement mémoire pour un objet de
 //type Point et on met sa référence dans la variable a
```

référence

a



objet  
Point

## Utilisation d'une classe

*Lorsqu'une classe est définie, il ne reste qu'à l'utiliser, en instanciant des objets et en appelant les méthodes décrivant les fonctionnalités. Mais attention, l'accès DIRECT aux champs encapsulés est impossible en dehors de leur classe.*

```
package tp1;
public class TestPoint
{ public static void main(String args [])
 { Point a = new Point (1,2) ;
 int abs=a.x; // tentative d'accès au CHAMP x déjà bien encapsulé dans sa classe
 int ord=a.y; // tentative d'accès au CHAMP y déjà bien encapsulé dans sa classe
 }
} // fin de la classe TestPoint
```

*Lorsque vous définissez une classe, vous devez aussi y définir toutes les méthodes utiles. En particulier, songez à définir les ACCESSEURS ou MUTATEURS. Il s'agit de méthodes (que l'on nomme aussi getter et setter) servant à accéder et à modifier individuellement chaque champ de l'objet.*

## ACCESSEURS: **getter** et **setter**

*Voici les accesseurs à insérer dans notre classe Point précédente.*

```
public class Point
{
 //code manquant

 public int getX ()
 { return x;
 }
 public int getY ()
 { return y;
 }
 public void setX (int abs)
 { x=abs;
 }
 public void setY (int ord)
 { y=ord;
 }
} // fin de la classe Point
```

**Remarque importante:** Java a introduit une convention importante sur le nommage des getter et setter qui stipule que leur nom doit commencer par **get** (pour un getter) et **set** (pour un setter) suivi du nom du champ et que **l'initiale** du nom du champ doit commencer par une lettre MAJUSCULE.

## Autres fonctionnalités

*A part les accesseurs, les autres fonctionnalités de la classe doivent y être insérées sous forme de méthodes d'instances ou méthodes de classe.*

*Voici par exemple, une méthode pour déplacer un point et une méthode pour afficher à la fois les coordonnées d'un point.*

```
class Point
{
 // code manquant

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

 public void affiche()
 { out.println("Point de coordonnées" + x + " et " + y);
 }
} //fin de la classe Point
```

Pour utiliser ces méthodes, il suffit de disposer d'une instance de la classe:

**Point z = new Point (23,-12)**

Et de faire: **z.deplace (-3,-4); // on dit que l'instance z accède à SA méthode deplace**  
**z.affiche ( );**

## Le constructeur

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, le constructeur se chargeait d'initialiser correctement les champs d'un objet de type Point.

**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 par défaut: **pseudo-constructeur**

**En Java, vous n'êtes pas obligé de créer effectivement un constructeur explicite lors de la définition d'une classe. Dans ce cas, Java vous fournit un constructeur par défaut appelé pseudo-constructeur.**

Il s'agit d'un constructeur sans paramètre ne faisant aucun traitement. Il sert à créer des objets avec une initialisation par défaut des champs aux valeurs « nulles » par défaut.

*Si on a :*

```
class Point
{
}
```

*Cela correspond à:*

```
class Point
{ public Point () { }
}
```

*Et on peut toujours écrire:*

```
Point a = new Point ();
```

**Mais ATTENTION:**

**Si vous créez explicitement un constructeur dans votre classe, le pseudo-constructeur n'existe plus.**

## 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 disposer d'un constructeur sans arguments qui est bien différent du pseudo-constructeur.**

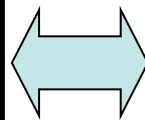
**Un constructeur peut appeler un autre constructeur de la même classe (A VOIR).**

**Un constructeur peut être déclaré **public** ou **privé**.**

## Autoréférence : **this** ( 2/3)

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

```
public class Point {
 //code manquant
 public Point (int abs, int ord) {
 x = abs;
 y = ord;
 }
 public void affiche()
 { out.println("Point de
 coordonnées" + x + " et " + y);
 }
```



```
public class Point {
 //code manquant
 public Point (int abs, int ord) {
 this.x = abs;
 this.y = ord;
 }
 public void affiche()
 { out.println("Point de coordonnées"
 + this.x + " et "
 + this.y);
 }
```

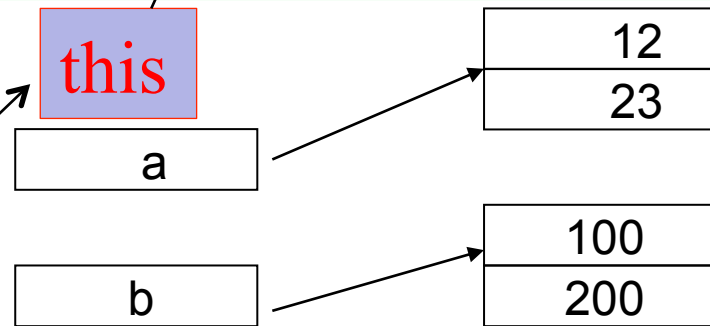
## Autoréférence : **this** ( 3/3)

```
public void affiche()
{ out.println("Point de coordonnées" + this.x + " et " + this.y);
}
```

Si on a deux objets a et b de type Point:

Point a = new Point (12, 23);

Point b = new Point (100, 200);



Dans un appel de la forme **a.affiche()**, la méthode reçoit une information lui permettant d'identifier l'objet ayant fait l'appel. Si l'information est transmise, la méthode pourra agir sur les champs spécifiques de l'objet.

*Cette transmission est gérée automatiquement par le compilateur.*

Mais on peut vouloir agir sur l'objet globalement au lieu d'agir sur les champs.

Dans ce cas Java utilise le mot clé **this**.

## Autoréférence : **this** : remarque

*Le mot clé **this** peut être utilisé pour simplifier l'écriture du constructeur.  
En clair, on peut utiliser les noms des champs identiques aux noms des arguments.*

```
public class Point {
 private int X ; // champ x d'un objet Point
 private int y ; // champ y d'un objet Point

 public Point (int X, int y) {
 this.X = X;
 this.y = y;
 }
}
```

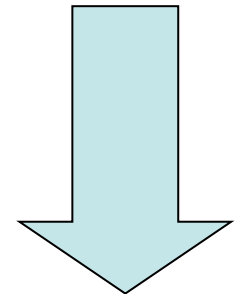
Le this est obligatoire ici.

# 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)

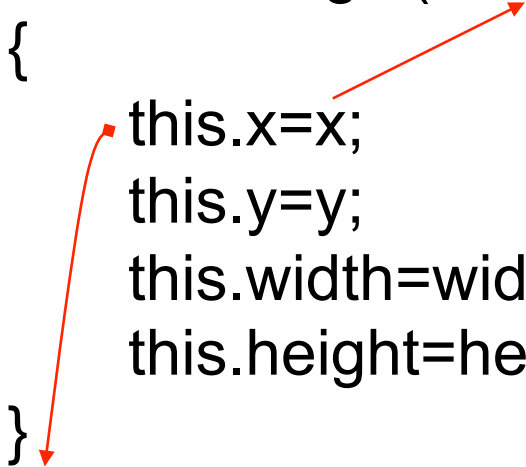
```
public class Individu {
 private String nom;
 private String prenom;
 private Compte c;
 public Individu (String nom, String prenom) {
 this.nom = nom;
 this.prenom = prenom;
 }
 public Individu (String nom, String prenom, Compte c) {
 // appel du constructeurs a deux arguments
 this (nom, prenom);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

ATTENTION :

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

# This: résoudre le conflit

```
public class Rectangle
{
 public Rectangle(int x, int y, int width, int height)
 {
 this.x=x;
 this.y=y;
 this.width=width;
 this.height=height;
 }
 int x, y, width, height;
 public int getWidth() {return width; }
}
```



The diagram illustrates how the 'this' keyword resolves variable conflicts in a constructor. Two red arrows originate from the 'this' keyword in the constructor body. One arrow points to the 'x' parameter in the constructor signature 'public Rectangle(int x, int y, int width, int height)'. The other arrow points to the 'x' field in the class field declaration 'int x, y, width, height;'. This visualizes the distinction between the local parameter and the class attribute.



# 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();
 }
}
```

## 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.

## Référence nulle: le mot clé **null**

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

**Les variables locales doivent  
toujours être initialisées avant toute  
utilisation.**

## Référence nulle: le mot clé **null**

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

**Les variables locales doivent  
toujours être initialisées avant toute  
utilisation.**

## Comparaison d'objets (1/2): == versus equals

```
public class Point
{ private int x ; // champ x d'un objet Point
 private int y ; // champ y d'un objet Point
 public Point (int abs, int ord) // un constructeur à deux arguments
 { x = abs ;
 y = ord ;
 }
 public static void main(String args [])
 {
 Point a = new Point (1,1);
 Point b = new Point (1,1);
 System.out.println ("avec == : " + a == b);
 System.out.println ("avec equals :" + a.equals (b));
 }
} //fin de la classe Point
```

Résultat

avec == : false

avec equals : false

## Comparaison d'objets (2/2)

**`==`** teste s'il s'agit du même objet ( pas d'une copie ).  
**`equals`** teste l'égalité de contenu de deux objets .

### ATTENTION :

dans l'exemple précédent la méthode **`equals`** dont il s'agit est celle de la classe **`Object`** (la super classe de toutes les classes en Java).

Souvent, vous serez emmené à *redéfinir* cette méthode.

Elle a pour en-tête:

```
public boolean equals (Object o)
```

L'opérateur **`!=`** s'applique également à des références d'objet pour tester la différence.

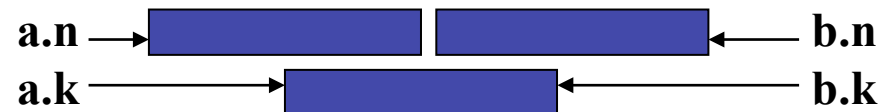
# Champs et méthodes de classe: le mot clé **static** .

## 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.k** et **b.k** peuvent être remplacés par **ChampStatic.k** . Mais si **k** est privé, on ne peut faire ceci.

## 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 que 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 nombre
 public MethodeStatic(long k) // d'objets créés
 { nombreInstanceCrees++;
 n = k ;
 }
 public void affiche ()
 {
 System.out.println ("nombre d'objets créés :" + nombreObjet());
 }
 public static long nombreObjet()
 {
 return nombreInstanceCrees;
 }
}
```

## Surdéfinition 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 changé.

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

## Exemple de surdéfinition 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.

# Surdéfinition de constructeurs

Les constructeurs peuvent être surdéfinis 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**.

# Protection

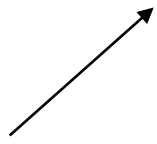
- **public**: permet un accès de n'importe qui
- **private**: accessible seulement par une méthode de la même classe
- **Sans spécification**: accessible du même package (à étudier plus tard)
- **protected** (package, plus tard)
- **Conseils**:
  - Protéger le plus possible les attributs d'un objet
  - Créer des méthodes d'accès de l'extérieur
    - Lire la valeur (*accessor*)
    - Modifier la valeur
  - Accessibilité des méthodes
    - **public**: permet à l'utiliser de l'extérieur
    - **private**: ne peut être utilisée que par une méthode définie dans la même classe
- Q: Peut-on créer un constructeur private?

# Conversion automatique de types

- Widening

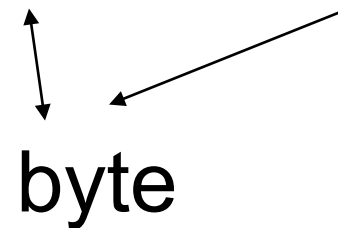
byte → short → int → long → float → double

char



- Narrowing (casting)

double → float → long → int → char ← → short





# Package

# Notion de Package

- Un package regroupe un ensemble de classes qui ont des fonctionnalités similaires.
  - Par exemple, le package `awt` regroupe toutes les classes reliées aux traitements de l'interface graphiques.
- Une autre hiérarchie pour les classes, mais selon leurs fonctionnalité
- Utilité: Faciliter l'importation des classes appropriées
- `import <package>.<sous-package>.<classe>;`

# Définir son propre package

- Inclure une classe dans un package: spécifier le package pour la classe comme suit

```
package <nom-package>;
class <nom-classe> {...}
```

- Pour définir un sous-package dans un package, il faut préciser tout le chemin pour arriver à ce sous-package comme suit:

```
package <package>.<sous-package>;
class <nom-classe> {...}
```

# Hiérarchie de package

- Package contient des sous-packages, ...
- Hiérarchie arborescente
- Deux aspects dans la définition de la hiérarchie:
  - Logique: spécifiée par « package » avant la définition de la classe;
  - Physique: le stockage des classes doit suivre la même hiérarchie de répertoire/sous-répertoire
- Exemple

```
package P1.P2;
class C {... }
```

ceci implique qu'il doit exister un répertoire nommé "P1", et dans lequel il y a un sous-répertoire "P2". Dans P2, il doit avoir la classe C.class.

# Rendre un package visible au compilateur

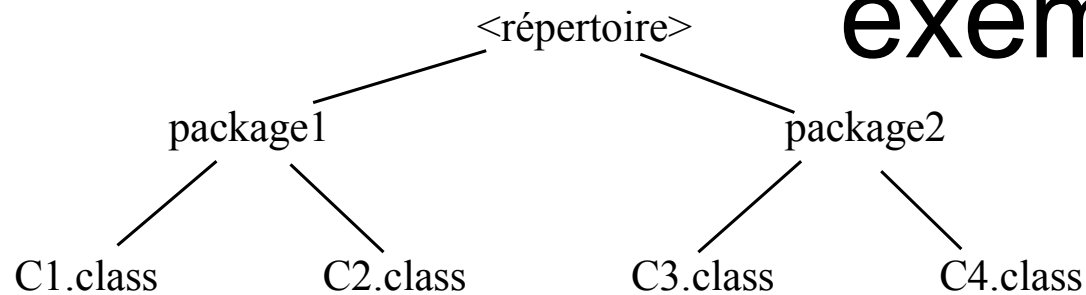
- Quand on compile un programme qui "import" des packages, il faut que ces packages soient visibles au compilateur.
- La visibilité est déterminée par la variable d'environnement CLASSPATH.
  - Par défaut, la CLASSPATH inclut le chemin pour accéder les packages prédéfinis du langage JAVA, et le répertoire courant (.).
- Pour indiquer au compilateur d'autres points où il doit aussi commencer sa recherche des packages, il faut étendre cette variable d'environnement avec d'autres chemins.
  - Les différents chemins dans CLASSPATH sont séparés par ";" » sous windows et par ":" sous linux ou mac.
  - `setenv CLASSPATH ./u/thiongam/JAVA`

# Accessibilité liée au package

- Classes/attributs private, public: faciles
- Par défaut: aucune spécification de protection
- protected

|            |                                                     |     |
|------------|-----------------------------------------------------|-----|
| Par défaut | Dans le même package                                | Oui |
|            | Dans un diff. package                               | Non |
| protected  | Dans le même package                                | Oui |
|            | Dans un autre package et à partir d'une sous-classe | Oui |
|            | Dans un autre package et en dehors de la classe     | Non |

# exemple



## C1.java:

```
package package1;
public class C1 {
 protected int X = 1;
 int y = 2;
}
```

## C2.java:

```
package package1;
public class C2 {
 C1 rc1 = new C1();
 void P2() {
 rc1.x = 3; // OK:
 rc1.y = 4; // OK.
 }
}
```

## C3.java:

```
package package2;
import package1.*;
public class C3 extends C1 {
 void P3() {
 x = 2; // OK
 y = 3; // erreur
 }
 void P33(C1 ref) {
 ref.x = 1; //erreur.
 ref.y = 2; //erreur;
 }
}
```

## C4.java:

```
package package2;
import package1.*;
public class C4 {
 void P4(C1 ref) {
 ref.x = 3; //erreur.
 ref.y = 4; // erreur;
 }
}
```

# Cas particulier pour "protected" dans un constructeur

- Pour une variable "protected" de la super-classe:
  - Si dans un constructeur d'une sous-classe, on crée une instance de cette même sous-classe ou d'une sous-sous-classe, alors on peut accéder à une variable (ou méthode) "protected" à partir de cette instance.
  - Si l'instance créée est de la super-classe, alors, on ne peut pas accéder à cette variable.



# Exemple

## **C1.java:**

```
package package1;
public class C1 {
 protected int X = 1;
 int y = 2;
}
```

## **C3.java:**

```
package package2;
import package1.*;
public class C3 extends C1 {
 C3() {
 C3 rc3 = new C3();
 rc3.x = 3; // OK car rc3
 est de la même sous-classe;
 }
}
```

Supposons une sous-classe C5  
de C3:

```
C3() {
 C5 rc5 = new C5();
 rc5.x = 3; // OK car rc5
 est d'une sous-sous-classe;
}
```

```
C3() {
 C1 rc1 = new C1();
 rc1.x = 3; // erreur.
}
```