

# La programmation objet avec Java

**Sur la base d'un cours de Rémi Forax, amendé par Dominique Revuz et Etienne Duris**

**Université Paris-Est Marne-la-Vallée**

# Bonnes lectures

- Java Language and Virtual Machine Specifications  
<http://docs.oracle.com/javase/specs/>
- Java Code Convention  
<http://www.oracle.com/technetwork/java/codeconventions-150003.pdf>
- Doug Lea's coding convention  
<http://gee.cs.oswego.edu/dl/html/javaCodingStd.html>
- How To Write Unmaintainable Code  
<http://thc.org/root/phun/unmaintain.html>
- Effective Java 2nd edition (Josh Bloch)
- Crown sourced Java questions  
<http://stackoverflow.com/questions/tagged/java>

# Programmation Objet

- **Programmation objet**
- Encapsulation
- Immutabilité

# Différents styles de programmation

- **impératif** (Algol, FORTRAN, Pascal, C)  
séquences d'instructions indiquant comment on obtient un résultat en manipulant la mémoire
- **déclaratif** (Prolog, SQL)  
description ce que l'on a, ce que l'on veut, et non pas comment on l'obtient
- **applicatif ou fonctionnel** (LISP, Caml)  
évaluation d'expressions/fonctions où le résultat ne dépend pas de la mémoire (pas d'effet de bord)
- **objet** (modula, Objective-C, Self, C++)  
unités de réutilisation qui abstraient et contrôlent les effets de bord

# Pourquoi contrôler/éviter les effets de bord ?

- Un effet de bord est une modification de la mémoire (ou entrée/sortie) qui induit un changement de comportement d'un programme
    - Dur à débbugger car dur à reproduire
    - Ne fonctionne pas sans mécanisme de synchronisation externe si plusieurs processus accèdent à la même zone mémoire
- => Quand on peut, on essaye d'éviter les effets de bord ou du moins de les contrôler

# Le style de programmation objet

- On considère que des composants autonomes (les **objets**) disposent de ressources et de moyens de communication entre-eux
- Ces objets représentent des **données** qui sont modélisées par des **classes** (qui définissent des types)
  - un peu comme des `typedef struct` en C
- Les classes définissent en plus les **actions** que les objets peuvent prendre en charge et la manière dont elles affectent leur état (messages ou **méthodes**)

# Pourquoi la programmation objet ?

## **Abstraction**

- Séparation entre la définition et son implantation
- Réutilisation de code car basé sur la définition

## **Unification**

- Les données et le code sont unifiés en un seul modèle

## **Réutilisation**

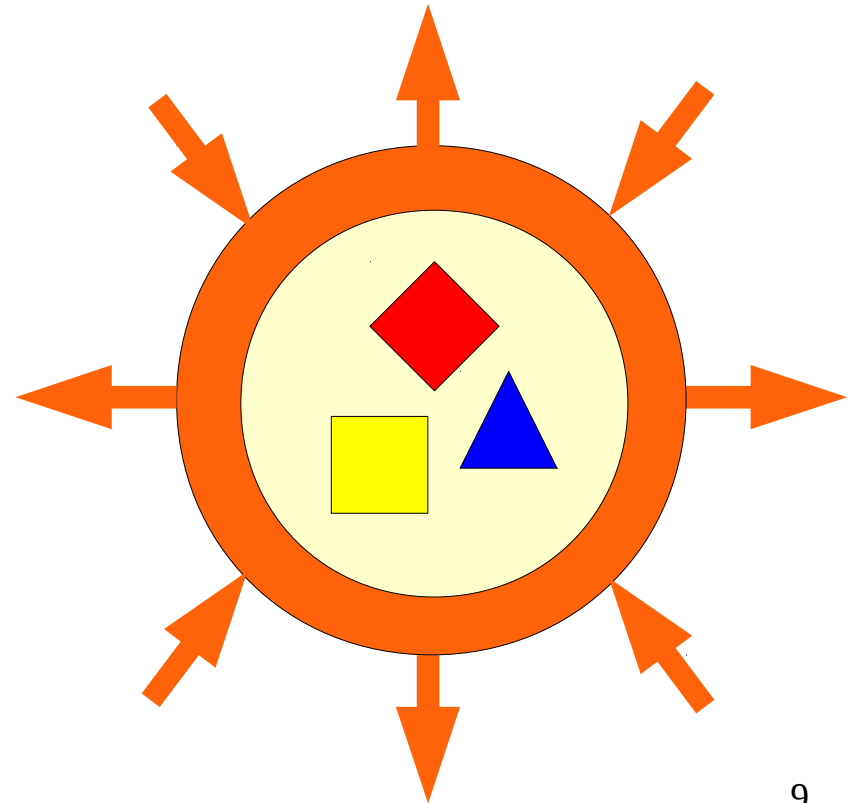
- La conception par classe conduit à des composants réutilisables (distingue et sépare les concepts)
- Cache les détails d'implantation

## **Spécialisation** (peu vrai dans la vraie vie en fait...)

- Le mécanisme d'héritage permet une spécialisation pour des cas particuliers

# Programmation modulaire

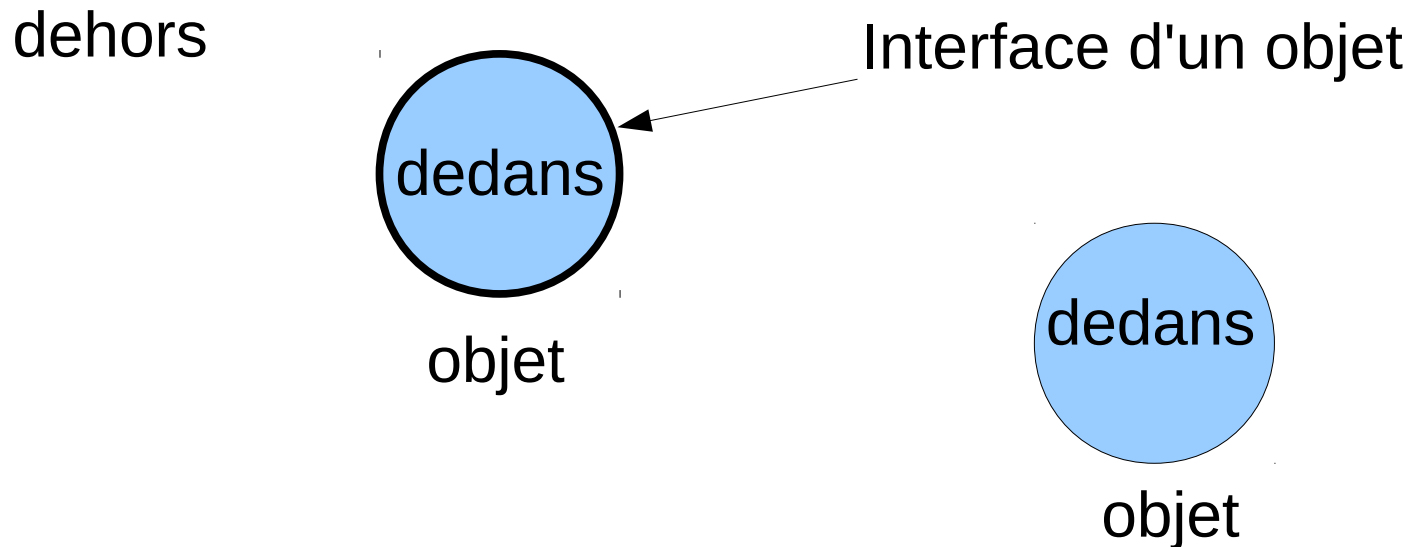
- La conception par classes, représentant à la fois les données, les actions et les **responsabilités** des objets de cette classe, permet de bien **distinguer et séparer les concepts**
- Le fait de définir des « **interfaces** », au sens « moyens et modalités de communication avec l'extérieur » permet de **cacher** les détails d'**implémentation** et d'éviter les dépendances trop fortes
- Tout ça favorise la réutilisabilité et la **composition / délégation**: l'assemblage des composants en respectant leurs responsabilités





# Qu'est ce qu'un objet ?

- Un objet définit un **dedans** et un **dehors**
- L'idée est que le **dehors** ne doit pas connaître la façon dont le **dedans** fonctionne



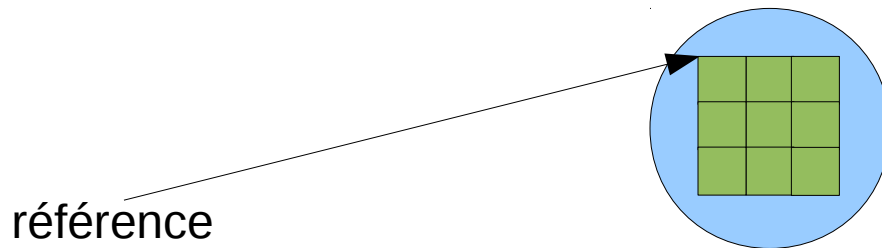
# Le **bien** et le *mal*

- Le bien (ce que l'on contrôle) est l'intérieur de l'objet
- Le mal (ce que l'on ne contrôle pas) est l'extérieur de l'objet
  - Vision relaxée pour les objets de la même sorte

On restreint les possibilités d'erreurs en  
**n'autorisant pas l'accès**

# Objet et mémoire

Un objet correspond à une zone en mémoire (une adresse); il connaît aussi sa taille



On manipule traditionnellement un objet par son adresse que l'on appelle une **référence**

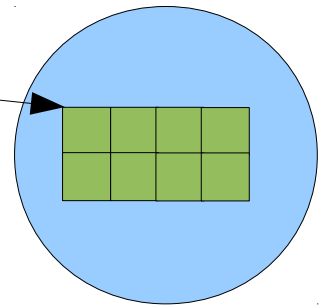
- On ne parle pas de pointeur car pas d'arithmétique

# Objet et classe

- La façon dont la mémoire est formatée à l'intérieur d'un objet est définie par la **classe** de l'objet (comme une struct en C)

```
class Point {  
    int x; // int => 32bits => 4 octets  
    int y; // int => 32bits => 4 octets  
} // taille totale 8 octets
```

référence



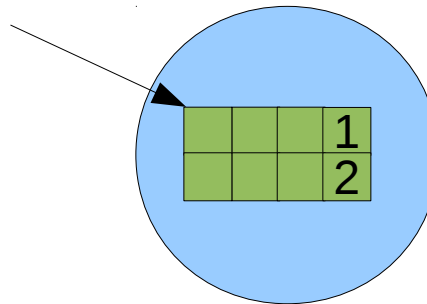
- la taille est souvent plus grosse dû à l'alignement et à des champs présents dans tous les objets
- l'ordre de déclaration n'est pas forcément l'ordre en mémoire

# Objet = instance d'une classe

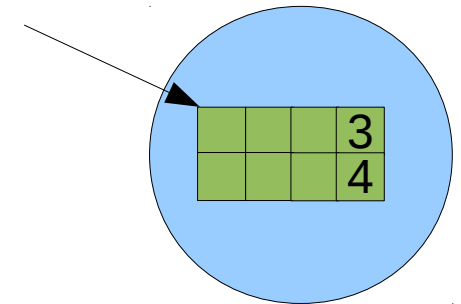
- Tous les objets d'une même classe sont organisés de la même manière (même type), mais ils ont un **état** qui leur est propre

```
class Point {  
  int x;  
  int y;  
}
```

p1



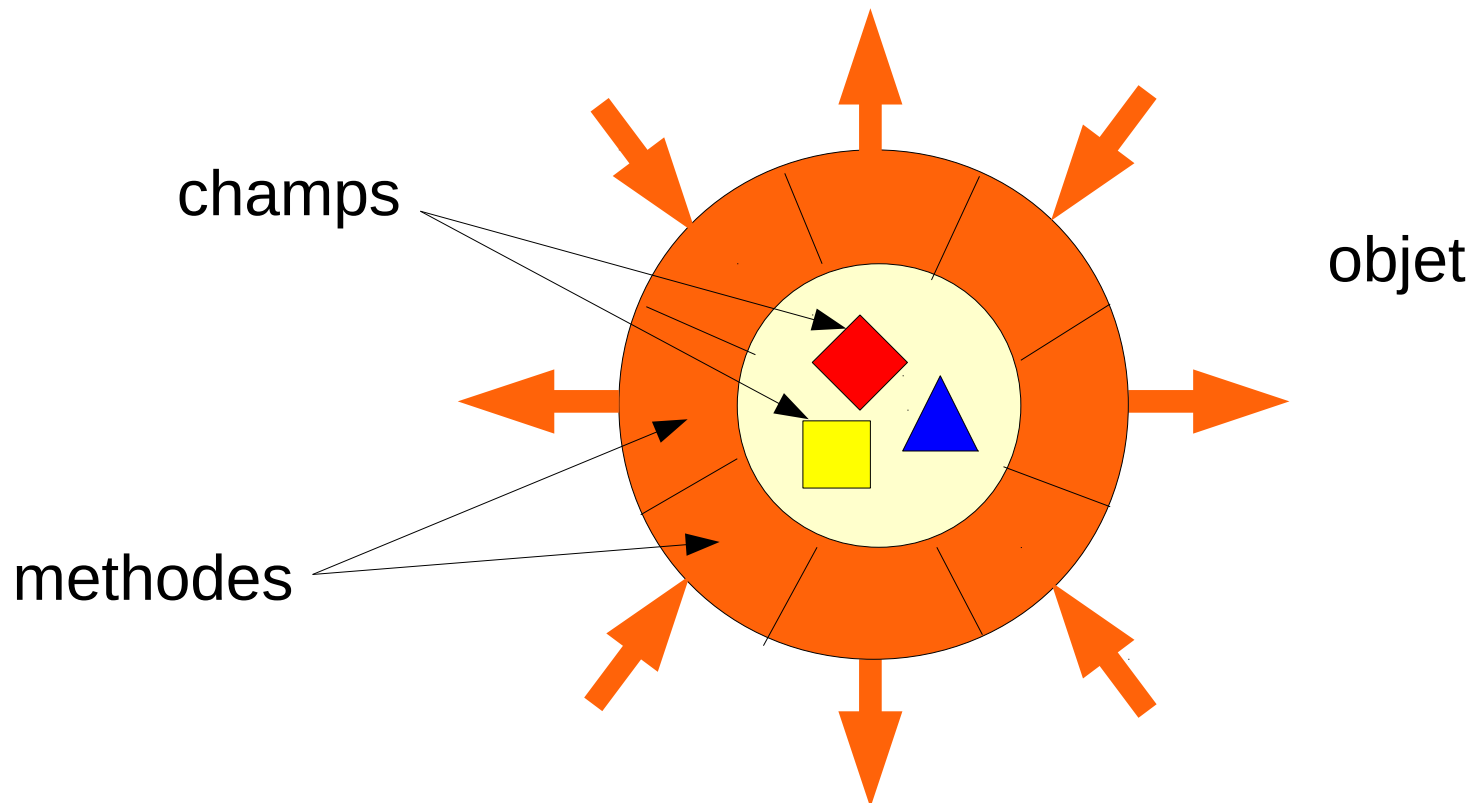
p2



- Les deux références, p1 et p2, permettent d'accéder à **deux instances** différentes de la **même classe** Point
  - Chaque objet a son propre état

# Méthodes

- Les méthodes représentent des points d'accès vis à vis de l'extérieur.
- Tout les champs d'un objet sont visibles (accessibles) dans les méthodes de **sa** classe.



# Classe et méthodes

- En plus de définir les **champs**, une classe définit le code qui va pouvoir les manipuler  
=> des **méthodes**
- Une méthode est **une fonction liée à une classe**

```
class Point {  
    int x, y;
```

```
    double distance() {  
        return Math.sqrt(x * x + y * y);  
    }
```

Méthode



- En Java, le code est **toujours<sup>(\*)</sup>** dans une méthode
  - Et une méthode toujours dans une **classe**

# Méthode

- L'exécution d'une méthode est associée à une **instance** d'une classe
- Lors de l'appel d'une méthode, il faut spécifier l'instance sur laquelle on appelle la méthode

```
Scanner scanner = ...  
scanner.nextLine();
```

  - on dit qu'on appelle la méthode `nextLine()`  
« *sur* » (la référence de) l'objet `scanner`
- A l'intérieur de la méthode, celle-ci a accès aux valeurs des champs liés à l'instance sur laquelle est fait l'appel (uniquement à celle ci).

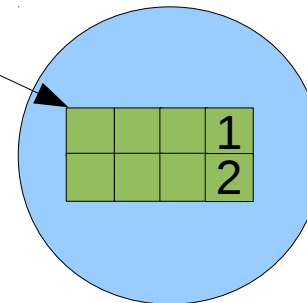


# Appel d'une méthode sur un objet

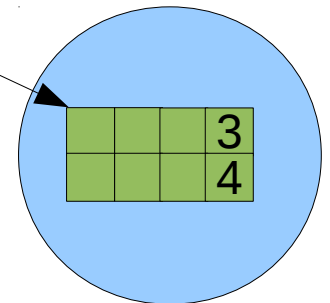
- Une même méthode, définie dans la classe `Point`, peut être appelée sur toutes les instances de cette classe
  - son exécution est liée à l'instance en question

```
class Point {  
    int x;  
    int y;  
    double distance() {...}  
}
```

p1



p2



- `p1.distance();`  
calcule `Math.sqrt(1*1 + 2*2);`
- `p2.distance();`  
calcule `Math.sqrt(3*3 + 4*4);`

# Une méthode est une fonction avec un paramètre caché

Ce paramètre caché est accessible dans la méthode appelée sous le nom de “**this**” en Java.

```
class AnotherClass {  
    void aMethod() {  
        Point p = ...  
        p.printXAndY();  
    }  
}
```

```
class Point {  
    int x;  
    int y;  
  
    void printXAndY() {  
        System.out.println(this.x + " " + this.y);  
    }  
}
```

Ici, **p** et **this** référencent  
le même objet



# this peut être sous-entendu

Le code généré pour les deux classes ci-dessous est identique

```
class Point {  
    int x;  
    int y;  
  
    void printXAndY() {  
        System.out.println(this.x + " "  
                             + this.y);  
    }  
}
```

```
class Point {  
    int x;  
    int y;  
  
    void printXAndY() {  
        System.out.println(x + " "  
                             + y);  
    }  
}
```

**this** est sous-entendu




# Appel inter-méthodes

- De la même façon que pour les champs, this peut être sous-entendu pour les méthodes

```
class Point {  
    int x;  
    int y;  
  
    void printX() { System.out.println(this.x); }  
    void printY() { System.out.println(this.y); }  
  
    void printXAndY() {  
        printX(); printY(); // est équivalent à this.printX(); this.printY();  
    }  
}
```

# Appel de fonction vs appel de méthode

Comme une méthode est liée à une classe, il n'est pas possible d'appeler une méthode sans envoyer un objet de cette classe

- En C, on écrit,  
Point p1 = ...; Point p2 = ...;  
distance(p1, p2)
- En Java, on écrit  
Point p1 = ...; Point p2 = ...;  
p1.distance(p2)  Il y a une bonne raison pour cette  
écriture que nous verrons plus tard  
(cf cours sur le sous-typage)

# Méthode statique

- En Java, il n'y a pas de fonction, le code est toujours dans une classe mais on peut dire que la méthode n'a pas besoin d'un objet (d'une instance de la classe) pour être appelée
- On déclare la méthode en utilisant le modificateur **static**
- On appelle alors la méthode sur la classe au lieu de l'appeler sur l'objet :  
`NomDeLaClasse.nomDeLaMethode ( )`


# Méthode static main

En Java, une classe qui possède une méthode `main()` qui prend en paramètre un tableau de chaînes de caractères peut être appelée à partir de la ligne de commande

```
public class Point {  
    public static void main(String[] args) {  
        ...  
    }  
}
```

`$java Point`

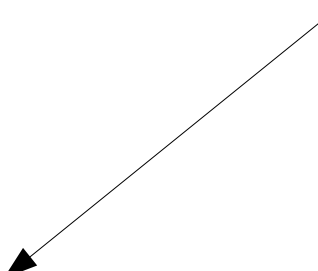
exécution  
depuis la  
ligne de  
commande



La signature doit être **exactement** celle-ci

# Exemple

La classe `Utils` n'est qu'un réceptacle pour les méthodes static



```
public class Utils {  
    private static void sum(int[] array) {  
        int sum = 0;  
        for(int value: array) {  
            sum += value;  
        }  
        return sum;  
    }  
    public static void main(String[] args) {  
        int[] array = new int[] { 1, 2, 3, 4, 5 };  
        System.out.println(Utils.sum(array));  
        System.out.println(sum(array));  
    }  
}
```



# Classe, champ, méthode et conventions

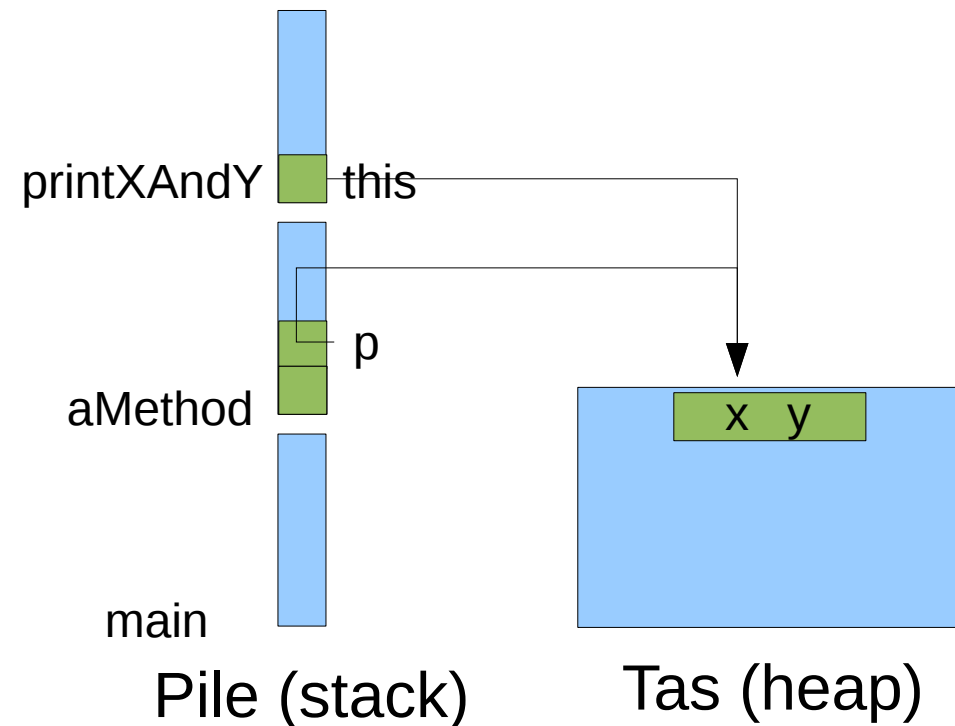
- En Java,
  - Une Classe commence par une Majuscule
  - Une méthode, un champ ou une variable locale commencent par une minuscule
- On utilise la convention *CamelCase*  
`CeciEstUneClasse`, `ceciEstUnChamp`,  
`ceciEstUneVariableLocaleOuUnParametre` et  
`ceciEstUneMethode()`
- On utilise `_` uniquement pour les constantes :  
`CECI_EST_UNE_CONSTANTE`

On écrit les noms en anglais !

रा.सू.॥ ॥श्रीगणेशायनमः॥ ॥अथरात्रिसूक्तं॥ ॥  
॥१॥ रात्रीव्यव्यदायतीपुरुत्रादेव्यक्षभिः॥विश्वा  
ऽअधिश्चियोधित॥ओर्वप्राऽअमर्त्यानिवतो  
देव्युद्धतः॥ज्योतिषाबाधतेतमः॥निरुस्वसार  
मस्तुतोषसदेव्यायती॥अपेदुहासतेतमः॥ ॥२॥  
सानोऽअयस्यावयंनितेयामुनविक्षमहि॥वृ  
क्षेनवसतिवयः॥नियामसोऽअविक्षतनि

# Ce que fait un appel de méthode

L'appel de méthode recopie les arguments dans les paramètres (p est recopié dans this)



```
public class AnotherClass {  
    void aMethod() {  
        Point p = ...  
        p.printXAndY();  
    }  
}  
  
public class Point {  
    int x;  
    int y;  
    void printXAndY() {  
        System.out.println(this.x  
                             + " " + this.y);  
    }  
}
```

# Passage par référence/par valeur ?

- On appelle passage **par valeur** un appel de fonction qui recopie sur la pile les arguments lors de l'appel de fonction
- On appelle passage **par référence** un appel de fonction qui copie sur la pile **l'adresse** des arguments lors de l'appel de fonction
  - par exemple en C
    - déclaration: `swap(int* a, int *b) { ... }`
    - appel: `swap(&i, &j);`
  - ou en C++,
    - déclaration: `swap(int& a, int& b) { ... }`
    - appel: `swap(i, j);`

# Appel de méthode en Java

- En Java, il n'y a pas de passage par référence, le passage se fait uniquement par valeur
- Comme un objet est manipulé par son adresse (sa référence), c'est donc sa référence qui est recopiée comme valeur
- Il n'est donc pas possible de passer l'adresse d'une valeur sur la pile en Java (pas de &) !

# En résumé, une classe c'est

- Une unité de compilation
  - La compilation d'un programme qui contient une classe Toto produira un fichier Toto.class
- La définition du type Toto
  - Il peut servir à déclarer des variables comme Toto t;
- Un moule pour la création d'objets de type Toto
  - Cela nécessite en général la définition d'un ensemble de **champs (fields)** décrivant l'état d'un objet de ce type et d'un ensemble de **méthodes** définissant son **comportement** ou ses **fonctionnalités**

# Structure d'une classe

- Une classe est définie par son nom complet
  - y compris package, ex: `java.lang.String`
  - En l'absence de directive, les classes sont dans un package dit « par défaut » (i.e., pas de package).
- Une classe peut contenir trois sortes de **membres**
  - Des champs (fields) ou attributs
  - Des méthodes (methods) et constructeurs
  - Des classes internes
- Les membres `static` sont dits membres de classe
  - Ils sont définis sur la classe et non sur les objets
  - Les membres non statiques (ou d'instance) ne peuvent exister sans un objet

# Ordre des membres

En Java, une classe est analysée par le compilateur en plusieurs passes on peut donc déclarer les méthodes, les champs, etc dans n'importe quel ordre

L'ordre usuel à l'intérieur d'une classe est:  
champs, constructeurs, getter/setter, méthode,  
méthodes statiques



```

public class Pixel {
    public final static int ORIGIN = 0;
    private int x;
    private int y;
    public Pixel(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public void reset() {
        x = ORIGIN;
        y = ORIGIN;
    }
    public void printOnScreen() {
        System.out.println("(" + x + ", " + y + ")");
    }
    public static boolean same(Pixel p1, Pixel p2) {
        return (p1.x == p2.x) && (p1.y == p2.y);
    }
    public static void main(String[] args) {
        Pixel p0 = new Pixel(0,0);
        Pixel p1 = new Pixel(1,3);
        p1.printOnScreen(); // (1,3)
        System.out.println(same(p0,p1)); // false
        p1.reset();
        System.out.println(same(p0,p1)); // true
    }
}

```

Constante

Champs

Constructeur

Méthodes  
d'instances

Méthode  
de classe

Variables locales  
à la méthode  
main et  
objets de la  
classe Pixel

# Objets, références et méthodes

Lorsque la méthode

```
p1.reset();
```

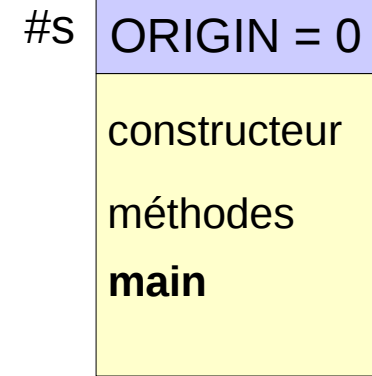
est appelée, le code de la méthode défini dans la classe comme

```
public void reset() {  
    x = ORIGIN;  
    y = ORIGIN;  
}
```

est exécuté sur la pile avec les références #1 à la place de p1 et #s à la place de ORIGIN

```
public void reset() {  
    #1.x = #s;  
    #1.y = #s;  
}
```

## Classe Pixel

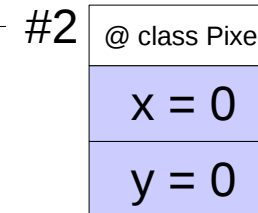
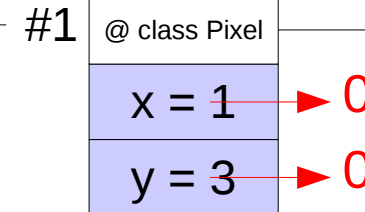


Pixel p1

#1

Pixel p2

#2

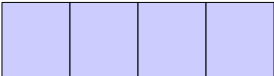


Ce qui a pour effet de mettre **p1.x** et **p1.y** à 0

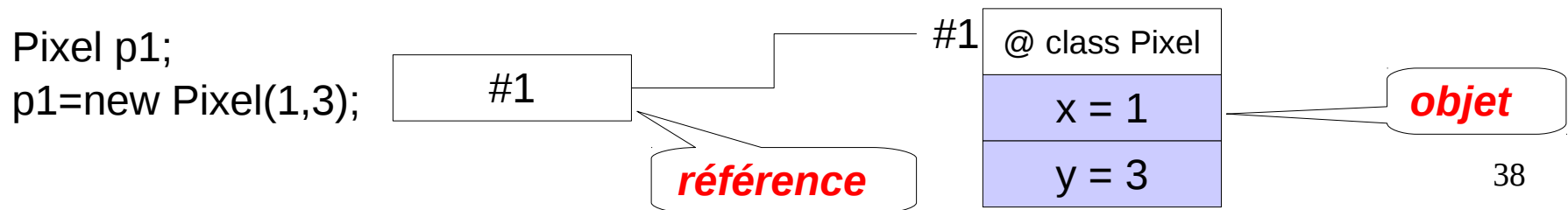
Chaque objet connaît sa classe

# La nature des variables en Java

- Les **variables locales** comme les **champs** des classes et des objets ne peuvent être que de deux natures
- De **type « primitif »**
  - Dans ce cas, la déclaration de la variable réserve la place mémoire pour stocker sa valeur (qui dépend de son type)

int entier;  long entierLong; 

- De **type « objet », ou référence**
  - Dans ce cas, la déclaration de la variable ne fait que réserver la place d'une **référence** qui permettra d'accéder à l'endroit en mémoire où est effectivement stocké l'objet en lui-même (vaut **null** si référence inconnue)



# Les types primitifs

- Types **entiers signés** (représentation en complément à 2)
  - **byte** (octet) sur 8 bits: [-128 .. 127]
  - **short** sur 16 bits [-32768 .. 32767]
  - **int** sur 32 bits [-2147483648 .. 2147483647] (défaut pour entiers)
  - **long** sur 64 bits [-9223372036854775808 .. 9223372036854775807]
- Type **caractère non signé** (unités de code UTF-16)
  - **char** sur 16 bits ['\u0000' .. '\uffff']
- Types à **virgule flottante** (représentation IEEE 754)
  - **float** sur 32 bits
  - **double** sur 64 bits (défaut pour flottants)
- Type **booléen**: **boolean** (**true** ou **false**)

# Tous les autres types sont « objets » et sont manipulés via des références

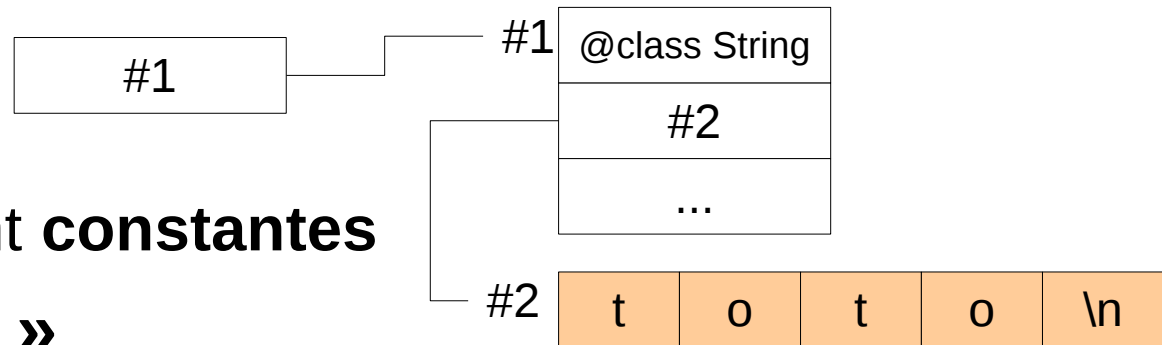
- soit des **types définis dans les APIs**

- java.lang.Object, java.lang.String, java.util.Scanner, etc.

- `String chaine = "toto";`

- Différent du C!

- En plus, les String sont **constantes**



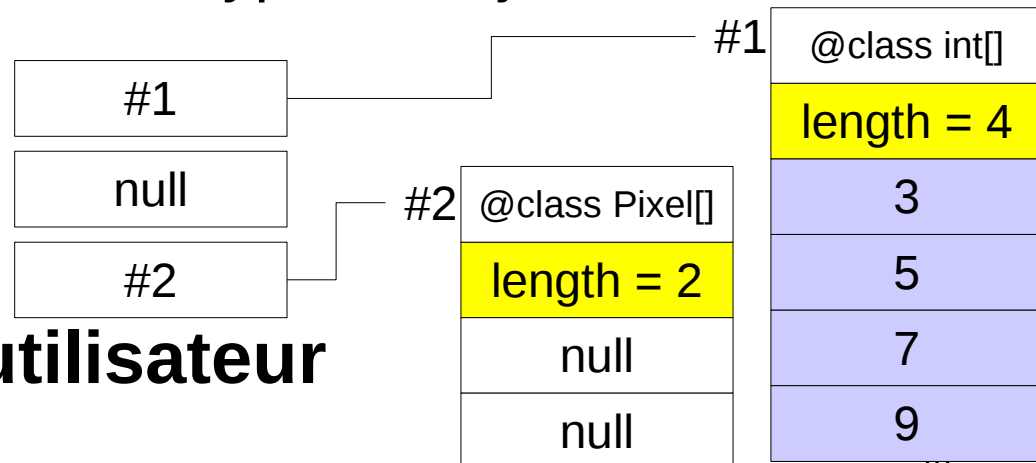
- soit des **types « cachés »**

- Tableau de types primitifs ou d'autres types « objets »

- `int[] tab = {3, 5, 7, 9};`

- `String[] args;`

- `Pixel[] array = new Pixel[2];`



- soit des **types définis par l'utilisateur**

- `Pixel p = new Pixel(0,0);`

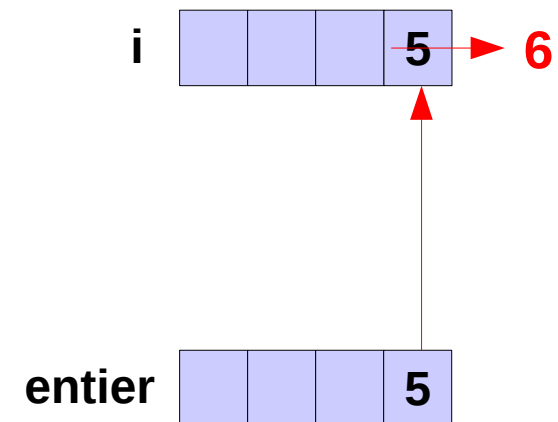
# Retour sur le passage de paramètre

- Lors des appels de méthode, les **arguments** sont toujours **passés par valeur**
- Dans le cas des **types primitifs**, c'est la **valeur** de l'argument qui est copiée dans le paramètre de la méthode
  - **Les modifications sur le paramètre de la méthode sont sans effet sur l'argument**
- Dans le cas des **types « objet »**, c'est la valeur de la variable, (la **référence** à l'objet) qui est transmise à la méthode
  - Les **modifications effectuées en suivant cette référence** (e.g. modification d'un champ de l'objet) sont **répercutés dans la mémoire** et sont donc visibles sur l'argument
  - En revanche, **la modification de la référence elle-même est sans effet sur l'argument** (c'en est une copie)

# Passage de paramètre: type primitif

- Dans le cas des **types primitifs**, c'est la **valeur** de l'argument qui est recopiée dans le paramètre de la méthode
  - **Les modifications sur le paramètre de la méthode sont sans effet sur l'argument**

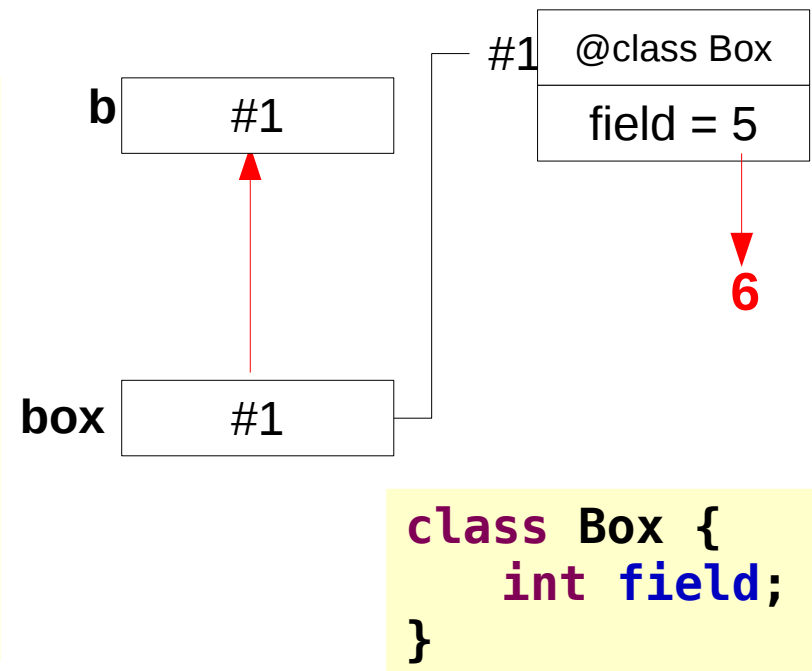
```
public static void m1(int i) {  
    i++;  
}  
  
public static void main(String[] args) {  
    int entier = 5;  
    m1(entier);  
    System.out.println(entier); // 5  
}
```



# Passage de paramètre: type référence

- Dans le cas des **types « objet »**, c'est la valeur de la variable, (la **référence** à l'objet) qui est transmise à la méthode
  - Les **modifications effectuées en suivant cette référence** (e.g. modification d'un champ de l'objet) sont **répercutés dans la mémoire** et sont donc visibles sur l'argument

```
public static void m2(Box b) {  
    b.field++;  
}  
  
public static void main(String[] args) {  
    Box box = new Box();  
    box.field = 5;  
    m2(box);  
    System.out.println(box.field); // 6  
}
```



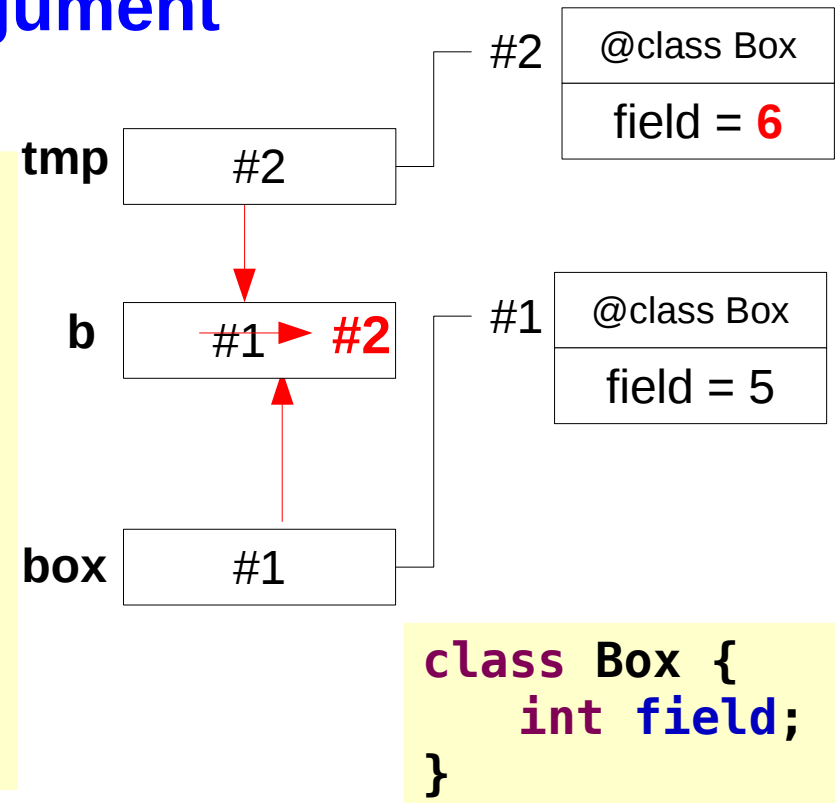


# Passage de paramètre: type référence

- Dans le cas des **types « objet »**, c'est la valeur de la variable, (la **référence** à l'objet) qui est transmise à la méthode

- En revanche, **la modification de la référence elle-même est sans effet sur l'argument** (c'en est une copie)

```
public static void m3(Box b) {  
    Box tmp = new Box();  
    tmp.field = b.field+1;  
    b = tmp;  
}  
public static void main(String[] args) {  
    Box box = new Box();  
    box.field = 5;  
    m3(box);  
    System.out.println(box.field); // 5  
}
```



# Type référence et valeur null

- Lorsqu'on déclare une variable de type objet, seule la place de la référence est réservée sur la pile d'exécution (registre)
  - 32 ou 64 bits, et ce quelque soit le type de l'objet référencé
  - Par défaut, cette référence vaut une valeur particulière, **null**.
  - Il est interdit de tenter d'y accéder, de la déréréferencer
- Le compilateur vérifie ce qu'il peut

```
public static void main(String[] args) {  
    Box b;  
    System.out.println(b.field); // Variable b might not have been initialized  
}
```

- On peut « forcer » pour que ça compile => lève une **exception**

```
public static void main(String[] args) {  
    Box b = null;  
    System.out.println(b.field); // lève NullPointerException  
}
```

# Allocation mémoire

- Pour qu'une variable objet prenne une autre valeur que null, il faut être capable d'y affecter une référence
  - Elle peut être produite (retournée) par l'opérateur d'allocation **new**
  - Cet opérateur a besoin de connaître la taille de l'objet qu'il doit réserver en mémoire
    - Le nom du type / de la classe suit immédiatement l'opérateur
      - `new Box();` // j'ai besoin de stocker 1 int (field)
      - `new int[10];` // stocker 10 int + la taille du tableau
    - La zone mémoire allouée doit être initialisée (affectation des valeurs)
      - `new Pixel(1,3);` // utilise un « constructeur »
      - Le terme de **constructeur** est mal choisi: **initialiseur** serait mieux
  - Ce que retourne l'opérateur new est **la référence** qui permet d'accéder à l'objet alloué en mémoire

# Désallocation mémoire

- Elle n'est pas gérée par le programmeur, mais par un **GC (Garbage Collector)**
- Les objets qui ne sont plus référencés peuvent être « récupérés » par le GC, pour « recycler » l'espace mémoire qu'ils occupent
- Un même objet peut être référencé par plusieurs variables
- Il faut qu'aucune variable ne référence plus un objet pour qu'il soit réclamé
- Les variables cessent de référencer un objet
  - Quand on leur affecte une autre valeur, ou null
  - Quand on quitte le bloc où elles ont été définies: elles meurent, disparaissent... (sur la pile)

# Références et Garbage Collector

- La quantité de mémoire disponible dans le tas de la VM est fixé à l'avance (paramétrable):
  - `java -Xms<size> MyAppli`
- C'est le gestionnaire de la mémoire qui se charge de
  - L'allocation des nouveaux objets
    - Demandée par l'opérateur new
  - La récupération de la place occupée par les objets morts (devenus inaccessibles)
    - Lorsqu'il y a besoin de place supplémentaire ou quand il le décide
  - De la manière d'organiser les objets
    - Pour éviter une « fragmentation » de la mémoire, il « déplace » les objets en mémoire (zone des « vieux » des « récents », etc.)
    - Les références ne sont pas des adresses (indirection)

# Programmation Objet

- Programmation objet
- **Encapsulation**
- Immutabilité

# Encapsulation

**“la seule façon de modifier l'état d'un objet est d'utiliser les méthodes de celui-ci”**

- Restreint l'accès/modification d'un champs à une quantité finie de code
  - Les méthodes de la classe
  - En Java, toutes les méthodes d'une classe sont dans un seul fichier (pas en C++, C#, Go)
- Permet de garantir des invariants
  - par ex, le champ x est toujours positif
- Permet de contrôler les effets de bord

# Encapsulation

Principe fondateur de la programmation orientée objet

- Aide la conception  
(1 responsabilité / 1 objet)
- Aide le debuggage  
le code de modification est localisé
- Aide l'évolution et la maintenance  
code localisé, abstraction par rapport au code



# Et en pratique ?

On déclare tous les champs “private”  
donc pas d'accès hors de la classe

On déclare les méthodes “public” si le code est accessible de l'extérieur ou “private” sinon

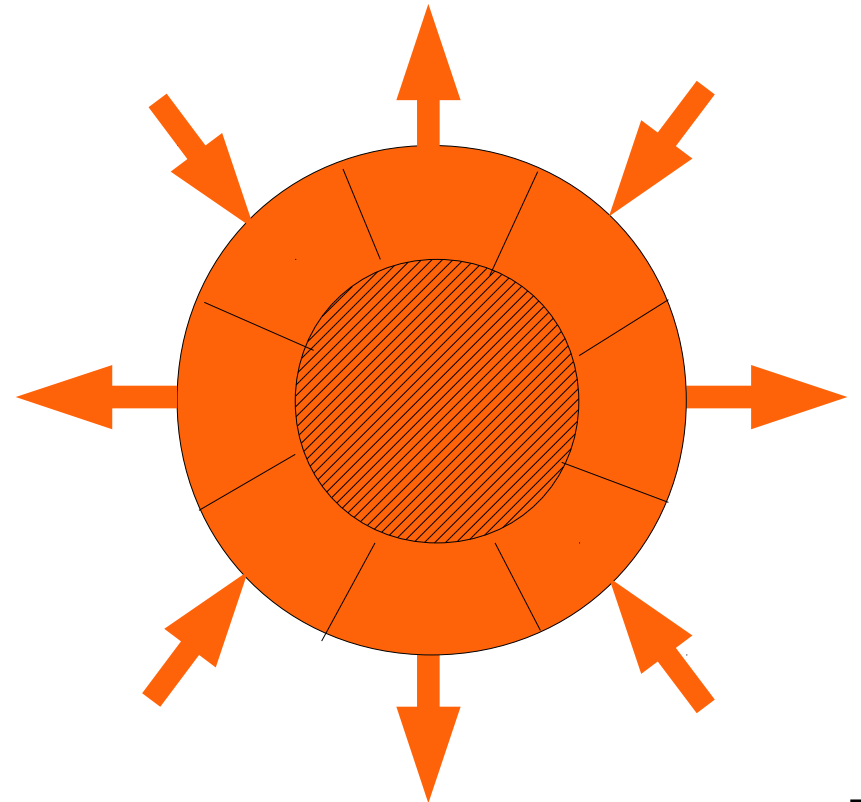
```
public class Point {  
    private int x;  
    private int y;  
  
    public double distance() {  
        return Math.sqrt(x * x + y * y);  
    }  
}
```

# L'interface d'un objet

L'interface d'un objet correspond à l'ensemble des points d'entrée d'un objet qui sont *visibles* depuis l'extérieur

=> on parle d'**accessibilité**

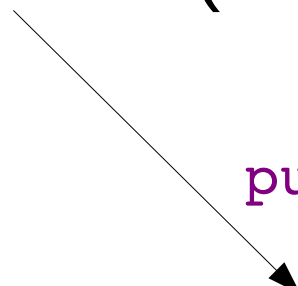
donc en Java,  
l'interface d'un objet  
est l'ensemble  
des méthodes publiques  
de celui-ci



# Autres accessibilités

Java possède 4 modificateurs d'accessibilité, donc 2 autres que “public” et “private”

L'accessibilité “par défaut” (rien) **n'est pas** private ou public



```
public class Point {  
    int x;  
    int y;  
}
```

de plus on déclare aussi la classe “public”  
(cf cours sur les packages)

# La classe en elle-même: accessibilité

- **class MyClass** (par défaut)
  - La classe sera **accessible** depuis toutes les classes du même paquetage qu'elle (on parle quelque fois de **visibilité de paquetage**)
- **public class MyClass**
  - La classe sera **accessible** de n'importe où (pourvu qu'on indique son nom de paquetage complet ou qu'on utilise la directive **import**)
- Cette notion d'accessibilité sert à définir des « composants » de plus grande granularité que les classes,
  - Permet de **limiter l'accès**
  - Peut éviter des **conflits de noms**

# Accessibilité des membres

- Tous les membres ont une **accessibilité** qui est spécifiée à la déclaration par un « modificateur »
- De manière **complémentaire à celui de la classe**, il permet de déterminer qui, parmi ceux qui ont accès à la classe A, ont accès à ce membre
  - **private** : accessible uniquement depuis l'intérieur de la classe A
  - **Par défaut** (pas de modificateur) : accessible depuis toutes les classes qui sont dans le même paquetage que A
  - **protected** : accessible depuis toutes les classes qui sont dans le même paquetage que A, et également depuis celles qui ne sont pas dans le même paquetage mais qui héritent de la classe A
  - **public** : accessible depuis n'importe où

# Accès aux champs et méthodes

- Avec le point « . » sur une **référence** à un objet:

**p.x, p0.sameAs(p1);**

- Le compilateur regarde le **type déclaré** de la variable (ici **p** ou **p0**) et vérifie que le membre (ici **x** ou **sameAs**) existe.
- Le compilateur vérifie également les droits d'accessibilité
- Un champ et une méthode peuvent avoir le même nom
- **this** représente l'instance courante

```
public class Pixel {
    private int x;
    private int y;
    public Pixel(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public boolean sameAs(Pixel p) {
        return (this.x==p.x) && (this.y==p.y);
    }
    public static void main(String[] args) {
        Pixel p0 = new Pixel(0,0);
        Pixel p1 = new Pixel(1,3);
        boolean b = p0.sameAs(p1); // false
    }
}

class OtherClass {
    public static void main(String[] args) {
        Pixel p0 = new Pixel(0,0);
        p.sameAs(p); // true
        p0.x = 1; // error: x has private
    }                // access in Pixel
}
```

# Les champs, par rapport aux variables?

- Leur existence et leur durée de vie sont **associées aux objets** (ou au pire à la classe elle-même)
  - Tandis que les variables locales sont associées à une exécution de la méthode qui les utilise... sur la pile!
- Les champs possèdent une **valeur par défaut** qui leur est affectée lors de la création d'un objet
  - **0** pour les types numériques primitifs
  - **false** pour les booléens
  - **null** pour les types référence
  - Tandis que les variables locales doivent nécessairement être initialisées avant d'être utilisées

# Création d'instances

- Une instance, ou un objet, d'une classe est créée en 3 temps

```
Pixel p1 = new Pixel(1,3);
```

- **new** est l'opérateur d'instanciation: comme il est suivi du nom de la classe, il sait quelle classe doit être instanciée
  - Il initialise chaque champ à sa valeur par défaut
  - Il peut exécuter un bloc d'initialisation éventuel
  - Il retournera la référence de l'objet ainsi créé
- **Pixel** est le nom de la classe
- **(1,3)** permet de trouver une fonction d'initialisation particulière dans la classe, qu'on appelle un constructeur



# Constructeur

- Un constructeur est une **méthode particulière**, qui sert à **initialiser** un objet une fois que la mémoire est réservée par **new**
  - Permet de garantir des **invariants** sur un objet sont conservés, par exemple pour initialiser un objet avec des valeurs particulières
    - Par exemple, on veut qu'un Pixel soit en (1,1)
  - Le faire avec une méthode d'initialisation « normale » ne garantirait pas qu'on accède pas à l'état de l'objet avant son initialisation

```
public class AClass {  
    public void aMethod() {  
        Calc calc = new Calc();  
        // ici, l'invariant « divisor!=0 » est faux!  
        calc.init(3);  
        int result = calc.multiply(4); // 12  
    }  
}
```

```
public class Calc {  
    private int divisor;  
    // invariant souhaité : « divisor!=0 »  
    public void init(int divisor) {  
        if (divisor == 0) ... // kaboom  
        this.divisor = divisor;  
    }  
    public double divide(int value) {  
        return value / divisor;  
    }  
}
```

# Constructeur (suite)

- L'initialisation « dans » le constructeur permet de garantir l'invariant

```
public class Calc {
    private int divisor;

    // invariant garanti dans le constructeur
    // « divisor!=0 »
    public Calc(int divisor) {
        if (divisor == 0) ... // kaboom
        this.divisor = divisor;
    }

    public double divide(int value) {
        return value / divisor;
    }
}
```

```
public class AClass {
    public void aMethod() {
        Calc calc = new Calc(3);
        // ici, l'invariant « divisor!=0 » est garanti
        int result = calc.multiply(4); // 12
    }
}
```

# Constructeur (fin)

- Le constructeur a le même nom que la classe et pas de type de retour
  - Il ne peut pas être appelé autrement qu'avec new
  - Si on met un type de retour, le compilateur croit que c'est une méthode:(
- En l'absence de constructeur explicitement défini, le compilateur ajoute un constructeur public sans paramètre

```
class Box {  
    private int field;  
    public static void main(String[] a){  
        Box b = new Box(); // OK  
    }  
}
```

- Si au moins un constructeur explicite est défini, le compilateur n'en ajoute pas

```
class Box {  
    private int field;  
    Box(int field) {this.field = field;}  
    public static void main(String[] a){  
        Box b = new Box(); // undefined  
    }  
}
```

# Appel à un autre constructeur

- Plusieurs constructeurs peuvent cohabiter dans la même classe
  - Ils ont typiquement des rôles différents et offrent des « services » complémentaires à l'utilisateur, par exemple:
    - `new Pixel(1,3)` crée un pixel avec les coordonnées explicites
    - `new Pixel()` crée un pixel avec les coordonnées à l'origine
    - `new Pixel(1)` crée un pixel sur la diagonale ( $x == y$ ), etc.
  - Quand c'est possible, il est préférable qu'il y en ait un « **le plus général** » et que **les autres y fassent appel**
    - Plutôt que de dupliquer le code dans plusieurs constructeurs
    - L'appel à un autre constructeur de la même classe se fait par `this(...)`

```
public class Pixel {  
    private int x;  
    private int y;  
    public Pixel(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public Pixel() {  
        this(0,0);  
    }  
    public Pixel(int v) {  
        this(v, v);  
    }  
}
```

# Champ constant

- Il est possible de déclarer un champ avec le modificateur **final**.
  - Cela signifie qu'il doit avoir une **affectation unique (une et une seule)**
  - Le compilateur vérifie qu'il a **bien été initialisé**, et ce **quelque soit le constructeur**, mais également qu'il n'a été **affecté qu'une seule fois**

```
public class Pixel {  
    private final int x;  
    private int y;  
    public Pixel(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public Pixel() {  
        // error: final field x may not  
        // have been initialized  
    }  
    public Pixel(int v) {  
        this(v, v);  
    }  
    public static void main(String[] a){  
        Pixel p = new Pixel(1);  
        p.x = 0; // error: final field x  
                // cannot be assigned  
    }  
}
```

# Constructeur comme point d'entrée

- En Java, si un constructeur existe, il n'est plus possible de créer un objet sans passer par un constructeur
- Le constructeur devient un point de passage obligé.
- On écrira dans le constructeur les tests garantissant que les valeurs envoyées respectent les invariants de la classe
  - Par exemple: un age est toujours positif.

# Constructeur privé

- Certaines classes n'ont pas de champ et servent juste de conteneur pour des méthodes statiques ou des constantes
- Dans ce cas, il est usuel de créer un constructeur privé pour empêcher de pouvoir créer un objet de cette classe

```
public class Utils {  
    private Utils() { /* garbage class */ }  
    public static int sum(int[] array) { ... }  
}
```

# Constructeur et déboggage

- Un constructeur **ne doit pas faire de calcul**, exécuter un algorithme, etc.
- Il doit juste faire des initialisations (et des tests si besoin)
- sinon, cela veut dire qu'il va falloir débogger le constructeur et débogger un objet à moitié initialisé n'est pas une bonne idée !



# Factory method to the rescue

Au lieu de:

```
public class A {  
    private int result;  
    public A(int value) {  
        // a complex code that uses value // oh no !!  
        result = ...  
    }  
}
```

... on écrit plutôt:

```
public class A {  
    private int result;  
    private A(int result) {  
        this.result = result; // cool  
    }  
  
    // factory method  
    public static A createA(int value) {  
        // a complex code that uses value  
        int result = ...  
        return new A(result);  
    }  
}
```

# Programmation Objet

- Programmation objet
- Encapsulation
- **Immutabilité**

# Effet de bord et encapsulation

L'effet de bord permet de violer le principe d'encapsulation

```
public class Point {  
    private int x;  
    private int y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public void setX(int x) {  
        this.x = x;  
    }  
}
```

Change l'état du cercle  
sans passer par une méthode  
du cercle

```
public class Circle {  
    private Point center;  
  
    public Circle(Point center) {  
        this.center = center;  
    }  
}  
  
public class AClass {  
    public void aMethod() {  
        Point p = new Point(2, 3);  
        Circle c = new Circle(p);  
        p.setX(4); // ahhhhh  
    }  
}
```

# Corriger le problème ?

Où est le problème ?

```
public class Point {  
    private int x;  
    private int y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public void setX(int x) {  
        this.x = x;  
    }  
}
```

là ?

```
public class Circle {  
    private Point center;  
  
    public Circle(Point center) {  
        this.center = center;  
    }  
}  
  
public class AClass {  
    public void aMethod() {  
        Point p = new Point(2, 3);  
        Circle c = new Circle(p);  
        p.setX(4);  
    }  
}
```

ici ?

# Pas d'effet de bord ?

Résoudre le problème des effets de bord

=> Ne pas en avoir !

c'est un principe de base de la programmation applicative

Traduction, dans le monde objet :

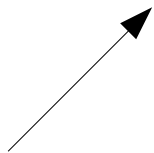
**“La modification de l'état d'un objet doit entraîner la création d'un nouvel objet”**

# Corriger le problème ? (2)

La classe Point ne doit pas permettre de modifier ses champs

```
public class Point {  
    private int x;  
    private int y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public Point setX(int x) {  
        return new Point(x, this.y);  
    }  
}
```

Astuce !



```
public class Circle {  
    private Point center;  
  
    public Circle(Point center) {  
        this.center = center;  
    }  
}  
  
public class AClass {  
    public void aMethod() {  
        Point p = new Point(2, 3);  
        Circle c = new Circle(p);  
        p.setX(4); // petit problème ici !  
                  // => p = p.setX(4) ;  
    }  
}
```

# Classe Immutable

Une classe immutable (ou non-mutable) est une classe qui ne permet pas la modification de son état

En Java, malheureusement, il n'y a pas de moyen de dire qu'une classe est non mutable

- Il faut donc l'écrire **explicitement dans la doc**
- On peut dire que les valeurs des champs ne doivent pas être modifiés et ce pour tous les champs

# java.lang.String est immutable !

java.lang

## Class String

Extrait de la javadoc

java.lang.Object  
    java.lang.String

### All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

---

```
public final class String  
extends Object  
implements Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:



# Tableaux toujours mutable

En Java, les tableaux sont toujours mutables !

```
public class Stack {  
    private final int[] array;  
  
    public Stack(int capacity) {  
        array=new int[capacity];  
    }  
    public int[] asArray() {  
        return array ;  
    }  
    public static void main(String[] args) {  
        Stack s=new Stack(3);  
        s.asArray()[2]=-30;  // impact array[2] !!!  
    }  
}
```

# Tableaux toujours mutable

En Java, les tableaux sont toujours mutables !

```
public class Stack {
    private final int[] array;

    public Stack(int capacity) {
        array=new int[capacity];
    }
    public int[] asArray() {
        return array.clone(); // defensive copy...
    }
    public static void main(String[] args) {
        Stack s=new Stack(3);
        s.asArray()[2]=-30; // OK
    }
}
```

# Pour avoir une classe immutable

Il faut déclarer tous les champs **final**

Le type des *arguments* passés à la construction doit être

- soit un type primitif
- soit une classe immutable
- soit une classe mutable dont on a fait une **copie défensive**

Si la valeur d'un champ typé avec une classe est *publié* vers l'extérieur, il faut faire une copie défensive

# Protection si un objet non mutable utilise un objet mutable

- Pour une méthode, lorsque l'on envoie ou reçoit un objet mutable, on prend le risque que le code extérieur modifie l'objet pendant ou après l'appel de la méthode
- Pour palier cela
  - Passer une copie/effectuer une copie de l'argument
  - passer/accepter un objet non mutable

# Corriger le problème ? (2)

Si la classe Point est mutable, la classe Cercle doit faire des copies défensives des paramètres qu'elle accepte...

```
public class Point {  
    private int x;  
    private int y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public void setX(int x) {  
        this.x = x ;  
    }  
}
```

```
public class Circle {  
    private Point center;  
  
    public Circle(Point center) {  
        this.center = center.clone();  
    }  
}  
  
public class AClass {  
    public void aMethod() {  
        Point p = new Point(2, 3);  
        Circle c = new Circle(p);  
        p.setX(4); // OK...  
    }  
}
```

# Corriger le problème ? (3)

Mais aussi des valeurs de retour qu'elle «publie» à l'extérieur...

```
public class Point {  
    private int x;  
    private int y;
```

Mutable

```
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public void setX(int x) {  
        this.x = x ;  
    }  
}
```

```
public class Circle {  
    private Point center;
```

Non-mutable

```
    public Circle(Point center) {  
        this.center = center.clone();  
    }  
    public Point getCenter() {  
        return center.clone();  
    }  
}
```

```
public class AClass {  
    public void aMethod() {  
        Point p = new Point(2, 3);  
        Circle c = new Circle(p);  
        p.setX(4); // OK...  
        p.getCenter().setX(4); // OK...  
    }  
}
```

# Alors mutable ou pas ?

## En pratique

- Les petits objets sont non-mutables, le GC les recycle facilement
- Les gros (tableaux, listes, table de hachage, etc) sont mutables pour des questions de performance

Et comment créer un objet non-mutable si l'un des champs est mutable ?

=> Copie défensive