La programmation Objet

Rémi Forax

A lire absolument !

- Java Code Convention
  http://www.oracle.com/technetwork/java/codeconv-138413.html
- 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 2nde edition (Josh Bloch)
  http://java.sun.com/docs/books/effective/
- Crown sourced Java questions
  http://stackoverflow.com/questions/tagged/java

Programmation Objet

- **Programmation objet**
- Encapsulation
- Immutabilité

Différents styles de programmation

- style impérative (Algol, FORTRAN, Pascal, C)

  séquence d'instructions indiquant comment on obtient un résultat en manipulant la mémoire

- style déclarative (Prolog, SQL)

  description ce que l'on a, ce que l'on veut, pas comment on l'obtient

- style applicative ou fonctionnelle (LISP, Caml)

  évaluations d'expressions/fonctions où le résultat ne dépend pas de la mémoire (pas d'effet de bord)

- style objet (modula, Objective-C, Self, C++)

  unités de réutilisation qui abstrait et contrôle les effets de bord

Différents styles de programmation

Chaque style de programmation n'exclue pas forcément les autres

La plupart des languages les plus utilisés actuellement (c++, objective-c, python, ruby, java, PHP, scala, clojure) permettent de mixer les styles

- avec plus ou moins de bonheur :)

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

- Dure à débugger car dure à reproduire
- Ne fonctionne pas sans mécanisme de synchronization externe si plusieurs processeurs accède à la même zone mémoire

  donc on fuit les effets de bord en les évitant ou en les contrôlant

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

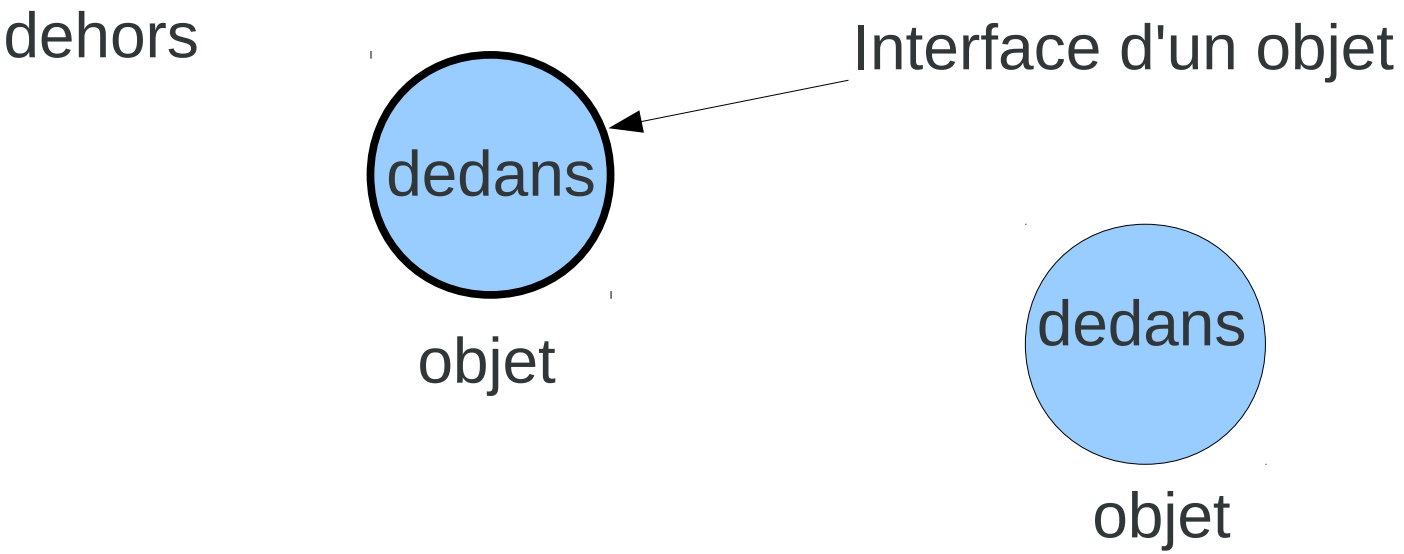
- La conception par classe conduit à des composant réutilisable
- Cache les détails d'implantation

Specialisation (peu vrai dans la vrai vie)

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

Qu'est ce qu'un objet ?

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



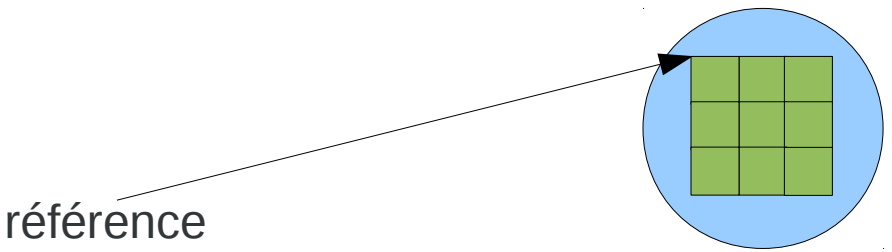
Le **bien** et le *mal*

- 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 relaxé pour les objets de la même sorte

On restreint les possibilités d'erreurs en ne permettant pas l'accès
(comme à la NSA normalement)

Objet et mémoire

Un objet correspond à une zone en mémoire (une addresse) et connait aussi sa taille

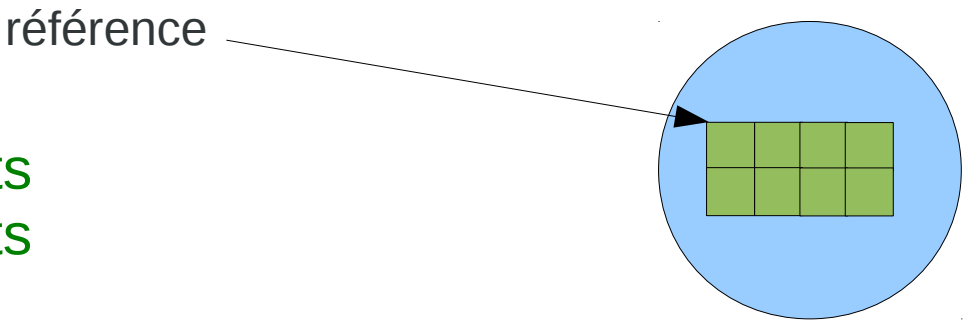


On manipule traditionnellement un objet par son adresse que l'on appel une référence
(pas un pointeur car pas d'arithmétique)

Objet et classe

La façon dont la mémoire est formattée à l'intérieur d'un objet est définie pas la classe d'un objet (comme une struct en C)

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



en fait la taille est souvent plus grosse due à l'alignement et à des champs présents dans tous les objets

Et s'il n'y a pas de classe

Javascript (Self en fait) est un langage objet sans classe

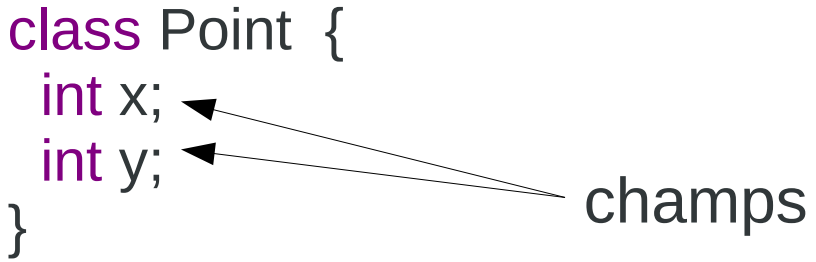
- On peut ajouter des champs à n'importe quel objet quand on veut

Astuce: la machine virtuelle à l'intérieur des browsers web crés des classes (hidden classes) non visible par l'utilisateur

Classe et champs

Une classe définie de la structure d'un objet en mémoire

Une classe des champs (ou attributs) dont le type indique la façon dont la mémoire est structurée



Contrairement au C, en Java, l'ordre des champs en mémoire n'est pas l'ordre des champs lors de la déclaration

Classe et méthodes

En plus de définir les champs, une classe définie le code qui va pouvoir manipuler les champs dans 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);
  }
}
```

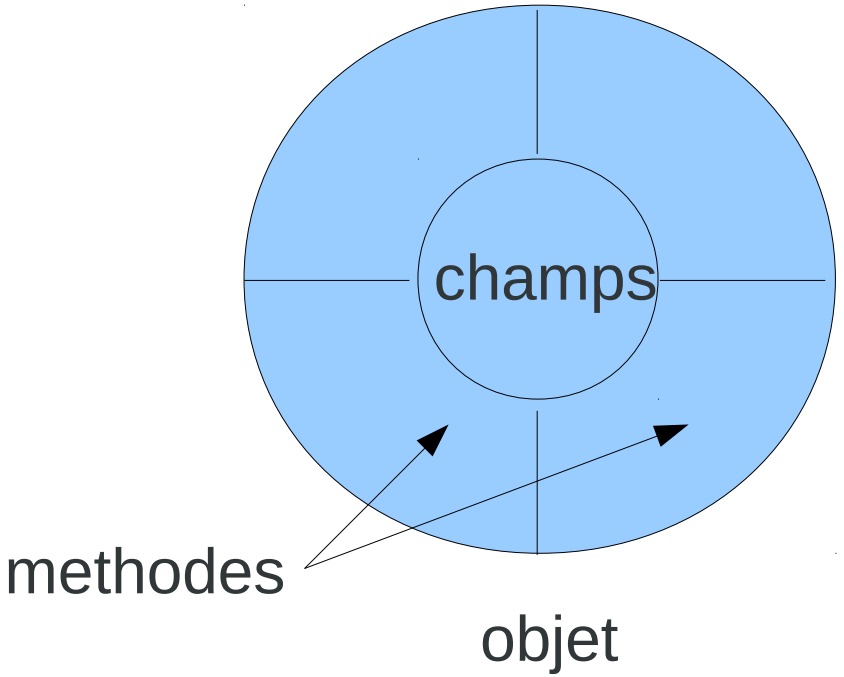
méthode

En Java, le code est **toujours** dans une méthode

Methodes & prog. objet

Une méthode représente un point d'accès vis à vis de l'extérieur d'un objet

Le code d'une méthode a accès à l'intérieur d'un objet



Méthode

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 appel la méthode

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

A l'intérieur de la méthode, celle-ci à accès aux valeurs des champs liés à l'instance passée lors de l'appel

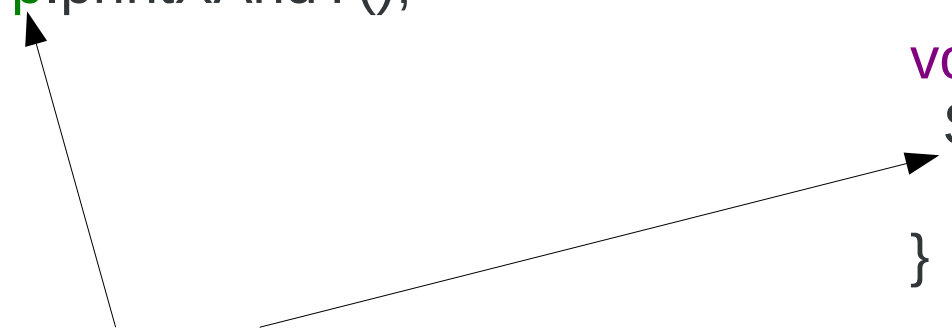
Une méthode est une fonction avec un paramètre caché

Ce caché paramètre 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 référence le même objet

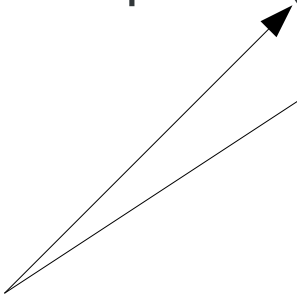


this peut être sous-entendu

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-méthode

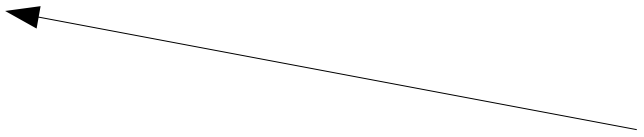
- De la même façon que pour les champs, this peut être sous-entendu pour les méthodes

```
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 function vs appel de méthode

Comme une méthode est liée à une classe, il n'est pas possible d'appeler une méthode sans envoyé 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 pour être appelée

On déclare la méthode en utilisant le modificateur **static**

On appel alors la méthode sur la classe
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) {
    ...
  }
}
```

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

Exemple

```
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));
  }
}
```

La classe Util**s** n'est qu'un receptacle pour les méthodes static

Classe, champ, méthode et convention

En Java,

- Une classe commence par une majuscule
- Une méthode, un champs ou une variable locale commence par une minuscule

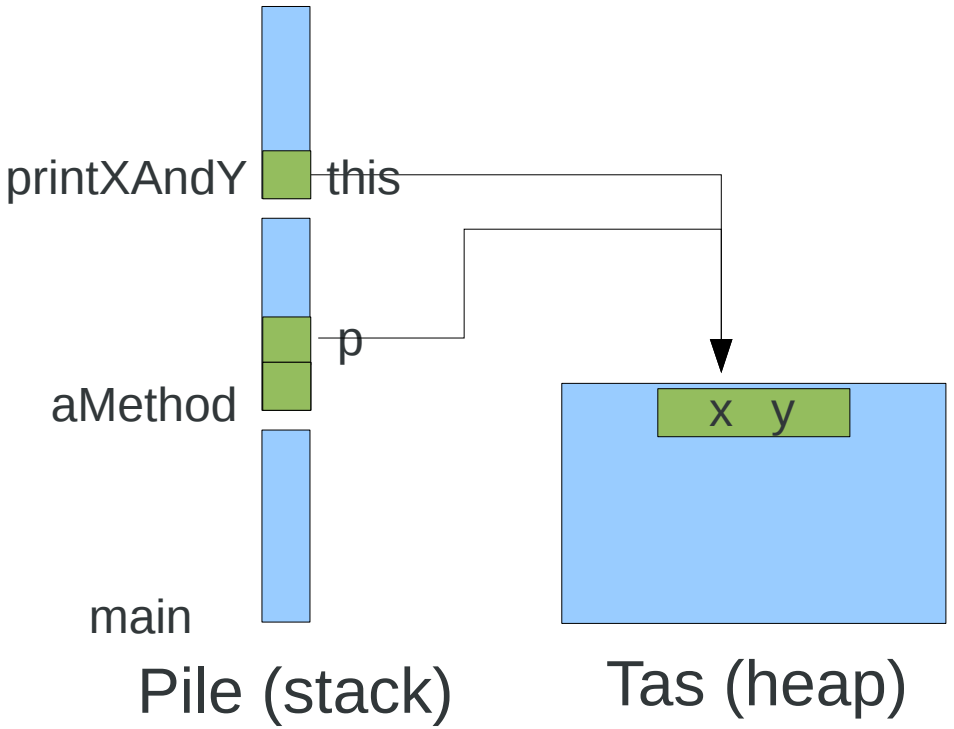
On utilise la convention CamelCase
CeciEstUneClasse, ceciEstUnChamp, ceciEstUneVariableLocalOuUnParametre et ceciEstUneMethode()

On utilise _ uniquement pour les constantes
CECI_EST_UNE_CONSTANTE

On écrit les noms en anglais !

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 appel passage par valeur un appel de fonction qui recopie les arguments lors de l'appel de fonction

on appel passage par référence un appel de fonction qui copie l'adresse sur la pile 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 réference) donc sa référence est recopié comme valeur

Il n'est donc pas possible de passer l'adresse d'une valeur sur la pile en Java (pas de &) !

Ordre des membres

En Java, une classe est analysée par le compilateur en plusieurs passe 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, méthode statique

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 un nombre fini 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 orienté 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 pratiquement ?

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

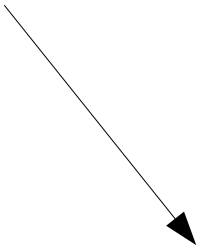
L'interface d'un objet correspond à l'ensemble des points d'entrée d'un objet qui sont visibles depuis l'extérieur

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

Autres visibilités

Java possède 4 modificateurs de visibilité, donc 2 autres que “public” et “private”

La visibilité “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)

Creation d'objets

La création d'objet (instanciation) se fait en utilisant le mot-clé new sur la classe de l'objet qui sera créé (l'instance de la classe)

Définition de la classe

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

Instanciation de la classe Point

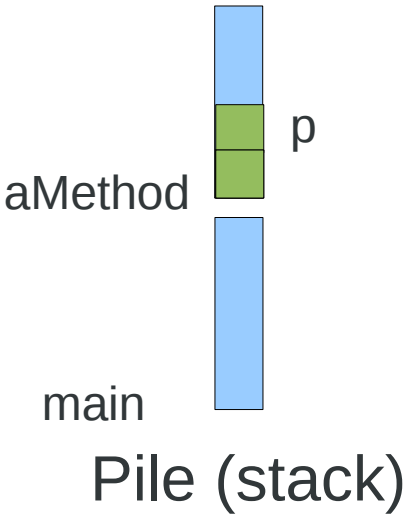
```
public class AnotherClass  {
  private void aMethod() {
    Point p = new Point();
  }
}
```

type

classe

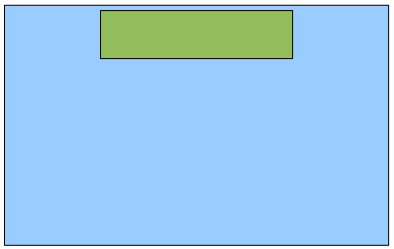
Ce que fait new ?

En mémoire chaque appel de méthode réserve la place nécessaire pour ses variable locales dans la pile



1) new utilise la taille de la classe pour réserver l'espace mémoire dans le tas

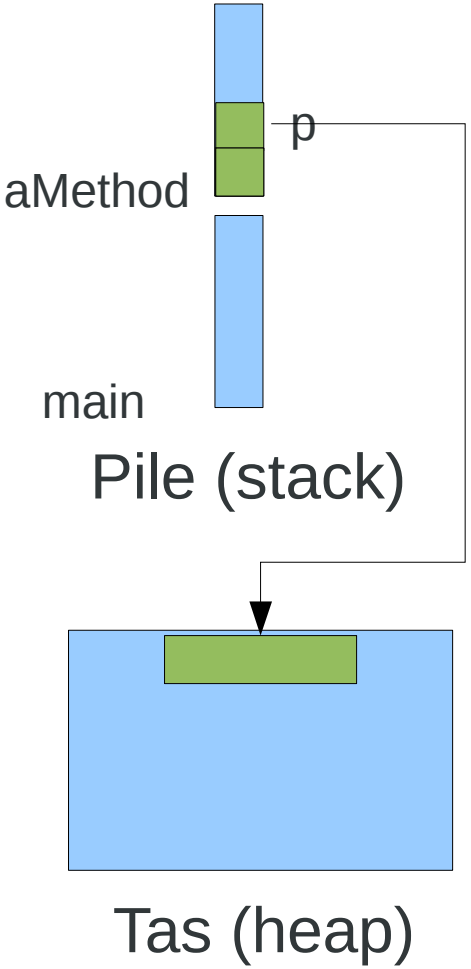
2) new initialise l'espace avec des 0 (si cela n'est pas fait globalement)




```
public class AnotherClass  {
  private void aMethod() {
    Point p = new Point();
    p.printXandY();
  }
}
```

Ce que fait new ? (suite)

Chaque champs est initialisé avec sa valeur par défaut (0 pour les int, false pour les boolean, etc)



3) p = ... assigne l'adresse en mémoire réservé par new dans la case p de la pile

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

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

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

Invariant et initialisation

Problème! entre le moment où l'instance est crée et où init() est appelée, l'invariant est faux !

```
public class AClass  {
  public void aMethod() {
    Calc calc = new Calc();
    // problème avant l'appel à init
    calc.init(3);
    int result = calc.multiply(4);  // 12
  }
}
```

```
public class Calc  {
  private int divisor;

  public void init(int divisor) {
    if (divisor == 0) ...  // kaboom
    this.divisor = divisor;
  }

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

Le constructeur

est une méthode d'initialisation qui est appelée automatiquement à la suite d'un new

- Remarque! En fait il ne construit rien, c'est le new qui construit, il initialise juste

En Java, un constructeur possède le même nom que la classe et ne spécifie pas de type de retour (c'est toujours void)

- si on spécifie un type de retour, le compilo croit que c'est une méthode :(

Invariant et constructeur

Les arguments du new sont passés au constructeur

L'invariant est garantie !

pas de type de retour !

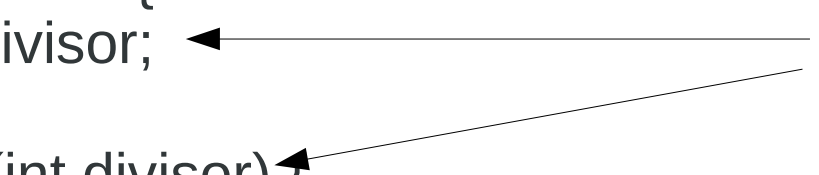
```
public class AClass  {
  public void aMethod() {
    Calc calc = new Calc(3);
    int result = calc.divide(4);  // 12
  }
}
```

```
public class Calc  {
  private int divisor;

  public Calc(int adivisor) {
    if (adivisor == 0) ...  // kaboom
    this.divisor = adivisor;
  }

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

Constructeur et initialisation de variable

Il est usuel d'utiliser le même nom pour les paramètres et le champs dans un constructeur

```
public class Calc  {
  private int divisor;

  public Calc(int divisor) {
    if (divisor == 0) ...  // kaboom
    this.divisor = divisor;
  }
}
```



dans ce cas on utilise la notation “this.” pour savoir de qui on parle

Constructeur comme point d'entrée

En Java, si un constructeur existe, il n'est pas possible de créer un objet sans passer par le constructeur

Le constructeur devient un point de passage obligé donc on écrira dans le constructeur les tests garantissant que les valeurs envoyées respectent les invariants

Constructeur privée

Certaine classe n'ont pas de champs et servent juste de conteneur pour des méthodes statiques ou des constantes

Dans ce cas, il est usuel de créer un constructeur privé pour empécher de pouvoir créer un objet de cette classe

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

Constructeur & compilateur

Si une classe ne possède pas de constructeur, un constructeur publique et sans paramètre est automatiquement ajouté par le compilateur

```
public class Car {
  private int numberOfWheels;

  public static void main(String[] args) {
     Car car = new Car();  // ok !
  }
}
```

Constrairement au C++, le constructeur sans paramètre n'est pas obligatoire !!

Constructeur et déboggage

Un constructeur ne doit pas faire des calculs, exécuter un algorithme, etc.

Il doit juste faire des initializations (et des tests si besoin)

sinon, cela veut dire qu'il va falloir débugger le constructeur et débugger 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:

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

Programmation Objet

- Programmation objet
- Encapsulation
- **Immutabilité**

Effet de bord et encapsulation

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;
  }
}
```

```
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;
  }
}
```

ici ?

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

Pas d'effet de bord ?

Résoudre le problème des effets de bord

=> Ne pas en créer

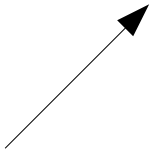
c'est un principe de base de la programmation applicative

Traduction, dans le monde objet :
“La modification de l'état d'un objet entraine 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 !
  }
}
```

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 modifié et ce pour tout 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:

Tableau toujours mutable

En Java, les tableaux sont toujours mutables !

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

Une classe immutable

Il faut déclarer tous les champs final

Le type des arguments passés à la construction doit être

- soit un type primitifs
- soit une classe immutable
- soit une classes 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

Alors mutable ou pas ?

En pratique

- Les petits objets sont non-mutable,
  le GC les recycle facilement
- Les gros (tableaux, list, 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