

# Java Avancé

## Cours 1 : Concepts

Khitem BEN ALI BLIDAOU

Slides d'après Florian Sikora

khitem.blidaoui@dauphine.psl.eu

# Objectifs de l'UE

- ▶ Consolider les bases en programmation objet
- ▶ Donner des éléments de développement logiciel
  - ▶ Build systems : Maven
  - ▶ SCMs (Source Control Manager) : Git
  - ▶ Testing : Junit
- ▶ Introduire quelques nouveautés
  - ▶ Programmation multi-threads
  - ▶ Construction fonctionnelles (Java 8)

# Prérequis et thèmes abordés

## Prérequis :

- ▶ De bonne bases en programmation impérative !!
- ▶ Notions et vocabulaire de la programmation objet  
(e.g. « faites une classe qui hérite de ArrayList et implémentez la méthode getSize() »)

## Vu pendant le cours/TD :

- |                                  |                      |
|----------------------------------|----------------------|
| ▶ Concepts, objet, encapsulation | ▶ Maven              |
| ▶ Héritage, polymorphisme        | ▶ Git                |
| ▶ Classes internes, anonymes...  | ▶ Eclipse            |
| ▶ Collections                    | ▶ JUnit              |
| ▶ Types paramétrés               | ▶ Shell Unix         |
| ▶ Exceptions                     | ▶ Quelques notion    |
| ▶ Enumérations                   | d'architecture objet |
| ▶ Threads                        |                      |

# Bibliographie

- ▶ Effective Java 2nd Edition - J. Bloch (1ère éd. traduite mais vieille). TRES BIEN.
- ▶ Programmer en Java 6eme Edition - C. Delannoy.
- ▶ Java in a nutshell - D. Flanagan.
- ▶ Thinking in Java - B. Eckel.
- ▶ Programmation concurrente en Java - B. Goetz.
- ▶ Tête la première, Design Patterns - E. Freeman et al.

... Meilleure approche : la pratique !!

# Déroulement & évaluation

**Cours** :  $10 \times 1.5h$

→ Exam : 60% de la note de l'UE

**TD** :  $14 \times 1.5h$

- ▶ Individuels, à finir chez soit
- ▶ À mettre en ligne sur GitHub

→ Contrôle continu : 10% de la note de l'UE

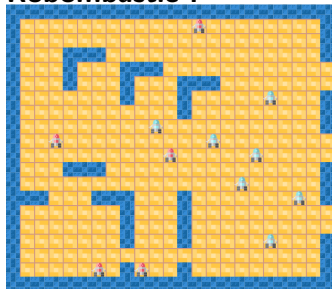
**Projet** :

- ▶ Projet de programmation en autonomie et en équipe
- ▶ Groupes de 2 ou 3
- ▶ Livrable : code + doc + rapport + démo

→ Évaluation du projet : 30% de la note de l'UE

# Exemple de projet (2016)

## Robombastic :



Auteurs : Moacdieh Tshilombo

- Conception objet
- Type paramétrés
- Multi-threading
- Interface graphique
- Chargement dynamique d'IAs pour contrôler les robots

# ATTENTION !!

- ▶ Le code remis est soumis a un outil de détection de plagiat
- ▶ L'examen de rattrapage compte 100% de la note de l'UE

# Cours 1 : Concepts

## Rappels...

Le paradigme de programmation objet

Concepts de la programmation objet

Import

Maven



# Java

- ▶ Orienté objet
- ▶ Indépendant de la plateforme (via VM).
- ▶ Semi-compilé / semi-interprété.
- ▶ Grosse API standard.

# Architecture en C

Code en ASCII → Compilateur → Fichiers objets → Éditeur de liens → Fichier binaire.

- ▶ Code source compilé en fichier objets.
- ▶ L'éditeur de liens lie les objets entre eux pour créer le fichier binaire (exécutable).
- ▶ Le binaire est exécuté directement sur le CPU

# Architecture en Java

Code en **Unicode** → Compilateur → Bytecode.

- ▶ Code compilé en représentation intermédiaire (bytecode)
- ▶ La machine virtuelle interprète le bytecode.

# Architecture en Java

Code en **Unicode** → Compilateur → Bytecode.

- ▶ Code compilé en représentation intermédiaire (bytecode)
- ▶ Un JIT (Just In Time Compiler) est appelé à l'exécution pour générer de l'assembleur depuis le bytecode.
- ▶ La machine hôte exécute l'assembleur.

# Avantages et inconvénients de la VM

- ▶ **Nécessite un VM pour exécuter le programme :**  
Performance moindre ou délai à l'exécution (JIT)
- ▶ **Permet la portabilité :** le même programme compilé peut s'exécuter sur n'importe quelle plat-forme disposant d'une VM Java
- ▶ **Facilite le développement :** e.g. le ramasse-miettes (GC) récupère les objets non utilisés (= 0 référence sur lui).
  - ▶ Déclenché périodiquement et lors d'un `new` si mémoire pleine.
  - ▶ Libère les objets qui ne sont plus référencés par aucune variable.

# Avantages et inconvénients de la VM

- ▶ **Nécessite un VM pour exécuter le programme :**  
Performance moindre ou délai à l'exécution (JIT)
- ▶ **Permet la portabilité :** le même programme compilé peut s'exécuter sur n'importe quelle plat-forme disposant d'une VM Java
- ▶ **Facilite le développement :** e.g. le ramasse-miettes (GC) récupère les objets non utilisés (= 0 référence sur lui).
  - ▶ Déclenché périodiquement et lors d'un `new` si mémoire pleine.
  - ▶ Libère les objets qui ne sont plus référencés par aucune variable.  
Faut-il mettre les variables à `null` pour aider le GC ?

```
1 private void main(){  
2     Object a = new A() ;  
3     doSomethingWithA(A) ;  
4     a = null ;  
5 }
```

# Avantages et inconvénients de la VM

- ▶ **Nécessite un VM pour exécuter le programme :**  
Performance moindre ou délai à l'exécution (JIT)
- ▶ **Permet la portabilité :** le même programme compilé peut s'exécuter sur n'importe quelle plat-forme disposant d'une VM Java
- ▶ **Facilite le développement :** e.g. le ramasse-miettes (GC) récupère les objets non utilisés (= 0 référence sur lui).
  - ▶ Déclenché périodiquement et lors d'un `new` si mémoire pleine.
  - ▶ Libère les objets qui ne sont plus référencés par aucune variable.  
Faut-il mettre les variables à `null` pour aider le GC ?

```
1 private void main(){  
2     Object a = new A() ;  
3     doSomethingWithA(A) ;  
4     a = null ;  
5 }
```

**Pas nécessaire !**

# Avantages et inconvénients de la VM

- ▶ **Nécessite un VM pour exécuter le programme :**  
Performance moindre ou délai à l'exécution (JIT)
- ▶ **Permet la portabilité :** le même programme compilé peut s'exécuter sur n'importe quelle plat-forme disposant d'une VM Java
- ▶ **Facilite le développement :** e.g. le ramasse-miettes (GC) récupère les objets non utilisés (= 0 référence sur lui).
  - ▶ Déclenché périodiquement et lors d'un `new` si mémoire pleine.
  - ▶ Libère les objets qui ne sont plus référencés par aucune variable.  
Faut-il mettre les variables à `null` pour aider le GC ?

```
1     private void fastRemove(int index) {  
2         modCount++;  
3         int numMoved = size - index - 1;  
4         if (numMoved > 0)  
5             System.arraycopy(elementData, index+1, elementData, index,  
6                             numMoved);  
7         elementData[--size] = null; // utile !!  
8     }
```

Oui !



# Exemple de programme en Java

- Les programmes Java sont des compositions d'Objets

Exemple : Objet Vec2D

```
1 import java.lang.Math ;
2
3 class Vec2D {
4     private double x ;
5     private double y ;
6
7     public Vec2D(double x, double y){
8         this.x = x ;
9         this.y = y ;
10    }
11
12    public String toString(){
13        return "Vec2D " + x + ", " + y ;
14    }
15
16    public double norm(){
17        return Math.sqrt( x * x + y * y ) ;
18    }
19 }
```

# Types en Java

- ▶ Séparation entre les **types primitifs** (boolean, int...) et les types **Objets** (String, int[], Date...).
- ▶ Types primitifs manipulés par leur **valeur**, types objets par **référence**.

# Types primitifs

- **8 types primitifs** seulement.

# Types primitifs

- ▶ **8 types primitifs** seulement.
  - ▶ Valeur booléen : `boolean` (`true/false`).
  - ▶ Valeur numérique entière signée : `byte`(8 bits, de -128 à 127), `short`(16), `int`(32), `long`(64).
  - ▶ Valeur numérique flottante (représentation !) : `float`(32), `double`(64).
  - ▶ Caractère unicode ( $\neq$  ASCII) : `char`(16).
    - ▶ Caractères lus (fichier, réseau...) rarement en Unicode !  
Conversion par Java selon le Charset de la plateforme : source de bugs.

# Types primitifs

- Attention pour byte, short :

```
1 short s=1 ;  
2 short s2=s+s ; //compile pas
```

- Pourquoi ?

# Types primitifs

- Attention pour byte, short :

```
1 short s=1 ;  
2 short s2=s+s ; //compile pas
```

- Pourquoi ?
- Promotion entière de l'addition.

## Types primitifs - flottants

- ▶ Normes (IEEE 754) pour représenter des flottants.
- ▶ 3.0 : double, 3.0f : float.
- ▶ Arrondi au nombre représentable le plus proche à chaque opération !

```
1 for(double d=0.0 ; d!=1.0 ; d+=0.1) {  
2     System.out.println(d) ;  
3 }
```

**Was passiert ?**

## Types primitifs - flottants

- ▶ Normes (IEEE 754) pour représenter des flottants.
- ▶ 3.0 : double, 3.0f : float.
- ▶ Arrondi au nombre représentable le plus proche à chaque opération !

```
1 for(double d=0.0 ; d !=1.0 ; d+=0.1) {  
2     System.out.println(d) ;  
3 }
```

Was passiert ?

- ▶ Boucle infinie !



```
1 0.7999999999999999  
2 0.8999999999999999  
3 0.9999999999999999  
4 1.0999999999999999
```



## Types primitifs - flottants

- ▶ Normes (IEEE 754) pour représenter des flottants.
- ▶ 3.0 : double, 3.0f : float.
- ▶ Arrondi au nombre représentable le plus proche à chaque opération !

```
1 for(double d=0.0 ; d !=1.0 ; d+=0.1) {  
2     System.out.println(d) ;  
3 }
```

### Was passiert ?

- ▶ Boucle infinie !



```
1 0.7999999999999999  
2 0.8999999999999999  
3 0.9999999999999999  
4 1.0999999999999999
```

- ▶ Pour représenter de la monnaie, préférer un int pour les centimes.

# Types Objets

- ▶ Présents dans l'API du JDK (`Date`, `String`...).
- ▶ Définis par l'utilisateur.
  - ▶ Mot clef `class` pour la définir.
  - ▶ `new` pour l'instancier.
  - ▶ Champs initialisés avec zéro (0, 0.0, false, null).

# Types Objets

- Membres (champ, méthode, classe...) **non statiques** d'une classe ont une référence implicite vers l'**instance courante**, noté `this`.

```
1 class Etudiant{
2     int id ;
3     int moyenne ;
4
5     void printEtuId() {
6         System.out.println("Etu No " + this.id
7             );
8     }
9
10    void printEtuMoy() {
11        System.out.println("Moyenne " +
12            moyenne); // this implicite
13    }
14 }
```

```
1 class EtuTest{
2     void test() {
3         Etudiant e = new Etudiant() ;
4         e.printEtuId() ;
5     }
6 }
```

- `this` (à gauche) et `e` (à droite) font référence au même objet.
- `this` implicite dans `printEtuMoy()`

# Constructeurs

```
1 public class Trex {  
2     private int a = 5;  
3  
4     public static void main(String[] args) {  
5         Trex t = new Trex();  
6         System.out.println(t.a);  
7     }  
8 }
```

► Compile ?

# Constructeurs

```
1 public class Trex {  
2     private int a = 5;  
3  
4     public static void main(String[] args) {  
5         Trex t = new Trex();  
6         System.out.println(t.a);  
7     }  
8 }
```

- ▶ Compile ?
- ▶ Oui, existe un constructeur par défaut (ne prenant aucun argument).

# Constructeurs

```
1 public class Trex {  
2     private int a = 5;  
3  
4     public Trex(int a) {  
5         this.a = a;  
6     }  
7  
8     public static void main(String[] args) {  
9         Trex t = new Trex();  
10        System.out.println(t.a);  
11    }  
12 }
```

► Compile ?

# Constructeurs

```
1 public class Trex {  
2     private int a = 5 ;  
3  
4     public Trex(int a) {  
5         this.a = a ;  
6     }  
7  
8     public static void main(String[] args) {  
9         Trex t = new Trex() ;  
10        System.out.println(t.a) ;  
11    }  
12 }
```

- ▶ Compile ?
- ▶ Non, plus de constructeur par défaut !

# Constructeurs

```
1 public class Trex {  
2     private int a = init();  
3  
4     public Trex() {  
5         this.a = 24;  
6     }  
7     private int init() {  
8         System.out.println("init");  
9         return 42;  
10    }  
11    public static void main(String[] args) {  
12        Trex t = new Trex();  
13        System.out.println(t.a);  
14    }  
15 }
```

► Compile ?



# Constructeurs

```
1 public class Trex {  
2     private int a = init();  
3  
4     public Trex() {  
5         this.a = 24;  
6     }  
7     private int init() {  
8         System.out.println("init");  
9         return 42;  
10    }  
11    public static void main(String[] args) {  
12        Trex t = new Trex();  
13        System.out.println(t.a);  
14    }  
15 }
```

► Compile ?

► Oui.

# Constructeurs

```
1 public class Trex {  
2     private int a = init();  
3  
4     public Trex() {  
5         this.a = 24;  
6     }  
7     private int init() {  
8         System.out.println("init");  
9         return 42;  
10    }  
11    public static void main(String[] args) {  
12        Trex t = new Trex();  
13        System.out.println(t.a);  
14    }  
15 }
```

- ▶ Compile ?
- ▶ Oui.
- ▶ Affiche ?

# Constructeurs

```
1 public class Trex {  
2     private int a = init();  
3  
4     public Trex() {  
5         this.a = 24;  
6     }  
7     private int init() {  
8         System.out.println("init");  
9         return 42;  
10    }  
11    public static void main(String[] args) {  
12        Trex t = new Trex();  
13        System.out.println(t.a);  
14    }  
15 }
```

- ▶ Compile ?
- ▶ Oui.
- ▶ Affiche ?
- ▶ init 24.
  - ▶ `init()` est d'abord appelé, mais le code du constructeur écrase la valeur de `a`.

# Visibilité

- ▶ 4 modificateurs de visibilité pour les membres d'une classe.
  - ▶ `private`

# Visibilité

- ▶ 4 modificateurs de visibilité pour les membres d'une classe.
  - ▶ `private`
    - ▶ Visible que dans la classe.

# Visibilité

- ▶ 4 modificateurs de visibilité pour les membres d'une classe.
  - ▶ `private`
    - ▶ Visible que dans la classe.
  - ▶ Sans modificateur.

# Visibilité

- ▶ 4 modificateurs de visibilité pour les membres d'une classe.
  - ▶ `private`
    - ▶ Visible que dans la classe.
  - ▶ Sans modificateur.
    - ▶ Visible par les classes du même package.

# Visibilité

- ▶ 4 modificateurs de visibilité pour les membres d'une classe.
  - ▶ `private`
    - ▶ Visible que dans la classe.
  - ▶ Sans modificateur.
    - ▶ Visible par les classes du même package.
  - ▶ `protected`



# Visibilité

- ▶ 4 modificateurs de visibilité pour les membres d'une classe.
  - ▶ `private`
    - ▶ Visible que dans la classe.
  - ▶ Sans modificateur.
    - ▶ Visible par les classes du même package.
  - ▶ `protected`
    - ▶ Visible par les classes héritées et celles du même package.

# Visibilité

- ▶ 4 modificateurs de visibilité pour les membres d'une classe.
  - ▶ `private`
    - ▶ Visible que dans la classe.
  - ▶ Sans modificateur.
    - ▶ Visible par les classes du même package.
  - ▶ `protected`
    - ▶ Visible par les classes héritées et celles du même package.
  - ▶ `public`

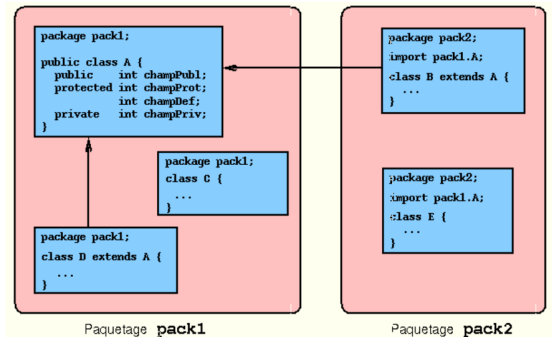
# Visibilité

- ▶ 4 modificateurs de visibilité pour les membres d'une classe.
  - ▶ `private`
    - ▶ Visible que dans la classe.
  - ▶ Sans modificateur.
    - ▶ Visible par les classes du même package.
  - ▶ `protected`
    - ▶ Visible par les classes héritées et celles du même package.
  - ▶ `public`
    - ▶ Visible par tout le monde.

# Visibilité

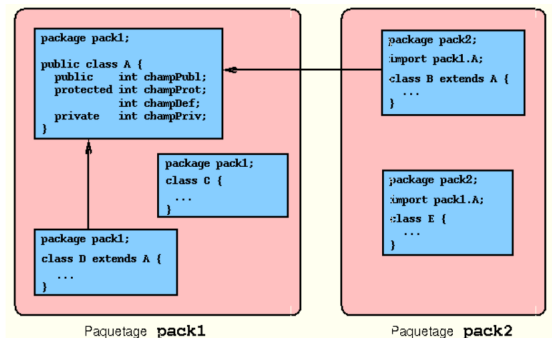
- ▶ 4 modificateurs de visibilité pour les membres d'une classe.
  - ▶ `private`
    - ▶ Visible que dans la classe.
  - ▶ Sans modificateur.
    - ▶ Visible par les classes du même package.
  - ▶ `protected`
    - ▶ Visible par les classes héritées et celles du même package.
  - ▶ `public`
    - ▶ Visible par tout le monde.
- ▶ `private < ' ' < protected < public`

# Visibilité



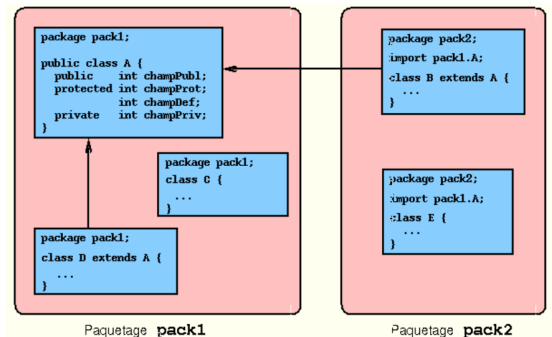
Dans :	A	B	C	D	E
champPubl					
champProt					
champ					
champPriv					

# Visibilité



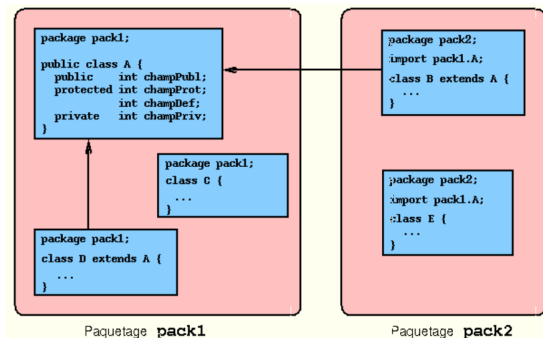
Dans :	A	B	C	D	E
champPubl	Oui	Oui	Oui	Oui	Oui
champProt					
champ					
champPriv					

# Visibilité



Dans :	A	B	C	D	E
champPubl	Oui	Oui	Oui	Oui	Oui
champProt	Oui	Oui	Oui	Oui	Non
champ					
champPriv					

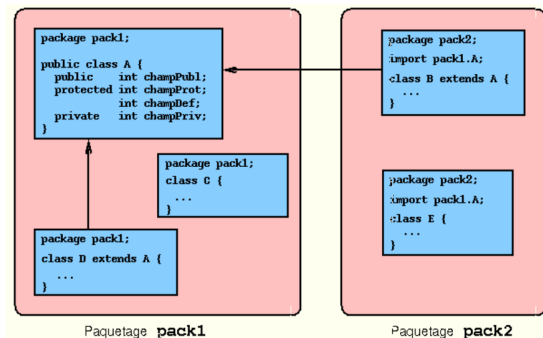
# Visibilité



Dans :	A	B	C	D	E
champPubl	Oui	Oui	Oui	Oui	Oui
champProt	Oui	Oui	Oui	Oui	Non
champ	Oui	Non	Oui	Oui	Non
champPriv					



# Visibilité



Dans :	A	B	C	D	E
champPubl	Oui	Oui	Oui	Oui	Oui
champProt	Oui	Oui	Oui	Oui	Non
champ	Oui	Non	Oui	Oui	Non
champPriv	Oui	Non	Non	Non	Non

# Visibilité - Règles

- ▶ Pas de champ en `public` ou `protected` (sauf constantes).
- ▶ Utilisation de la visibilité de package (`rien`) si une classe partage des détails d'implémentation avec une autre (classe interne..).
- ▶ Méthode en `public` uniquement si nécessaire.

# Nommage

- ▶ Par convention :
  - ▶ Classe commence par une majuscule.
  - ▶ Une méthode, un champ, une variable locale par une minuscule.
  - ▶ Majuscule pour chaque mot suivant (sauf constantes).
  - ▶ En anglais...
- ▶ `ThisIsAClass`
- ▶ `thisIsAMethod`
- ▶ `thisIsAField`
- ▶ `thisIsAVariable`
- ▶ `THIS_IS_A_CONSTANT`

# Contexte static

- ▶ Mot clef `static` pour définir des membres liés à la classe et non à une instance.
  - ▶ Champs.
  - ▶ Méthodes.
  - ▶ Classes...
  - ▶ Bloc d'initialisation.
- ▶ Utilisés sans instance dans la classe.

## Contexte static – variables

- Champ statique pas propre à un objet..

```
1 public class Static {  
2     private int value ;  
3     private static int staticValue ;  
4  
5     public static void main(String[] args) {  
6         Static st1=new Static() ;  
7         Static st2=new Static() ;  
8         System.out.println(st1.value++) ;  
9         System.out.println(Static.staticValue++) ;  
10        System.out.println(st2.value++) ;  
11        System.out.println(Static.staticValue++) ;  
12    }  
13 }
```

- Sortie ?

## Contexte static – variables

- Champ statique pas propre à un objet..

```
1 public class Static {  
2     private int value ;  
3     private static int staticValue ;  
4  
5     public static void main(String[] args) {  
6         Static st1=new Static() ;  
7         Static st2=new Static() ;  
8         System.out.println(st1.value++) ;  
9         System.out.println(Static.staticValue++) ;  
10        System.out.println(st2.value++) ;  
11        System.out.println(Static.staticValue++) ;  
12    }  
13 }
```

- Sortie ?

- 0
- 0
- 0
- 1

## Contexte static – variables

- ▶ Accès au champ avec le nom de la classe.
- ▶ Attention, compilateur l'autorise avec un objet (warning).

```
1 public class Static {  
2     private int value ;  
3     private static int staticValue ;  
4  
5     public static void main(String[] args) {  
6         Static st1=new Static() ;  
7         System.out.println(st1.value++) ;  
8         System.out.println(Static.staticValue++) ;  
9         System.out.println(st1.staticValue++) ;  
10    }  
11 }
```

## Contexte static – constantes

- ▶ Constantes en C : `#define`
- ▶ En java : `static` et `final`.
- ▶ Par convention, en majuscule, mots séparés par des `_`.

```
1 public class Cst {  
2     private final static int MAX_SIZE = 1024 ;  
3     public static void main(String[] args) {  
4         int[] t = new int[MAX_SIZE] ;  
5     }  
6 }
```



## Contexte static – méthodes

- ▶ Méthode static : peut-être appelée sans instance d'un objet (comme une fonction en C).
- ▶ Méthode qui n'utilise aucune variable d'instance (variables non statiques)
- ▶ Le mot clé `this` ne peut pas être utilisé
- ▶ Appelé par le nom de la classe.

```
1 public class Point {  
2     private int x,y;  
3     private static double value ;  
4  
5     private static int test() {  
6         int v=value ; // ok  
7         return x+y ; // ko, pourquoi ?  
8     }  
9 }
```

# Contexte static - blocs

- ▶ Bloc exécuté **une seule fois** lors de l'initialisation de la classe.
  - ▶ En java, classes chargées que si nécessaire (si appel).
- ▶ Initialisation de champs statiques complexes.

```
1 public class Colors {  
2     private static final HashMap<String,Color> colorMap ;  
3     static {  
4         colorMap = new HashMap<String,Color>() ;  
5         colorMap.put("Rouge",Color.RED) ;  
6         colorMap.put("Vert",Color.GREEN) ;  
7         ...  
8     }  
9     public static Color getColorByName(String name) {  
10         return colorMap.get(name) ;  
11     }  
12 }
```

# Cours 1 : Concepts

Rappels...

**Le paradigme de programmation objet**

Concepts de la programmation objet

Import

Maven

# Paradigmes de programmations

Paradigme de programmation = Modèle de programmation

# Paradigmes de programmations

Paradigme de programmation = Modèle de programmation

- ▶ Impératif (Fortran, C...)
  - ▶ Variables mutables + séquence d'instructions qui manipulent les variables.

# Paradigmes de programmations

Paradigme de programmation = Modèle de programmation

- ▶ Impératif (Fortran, C...)
  - ▶ Variables mutables + séquence d'instructions qui manipulent les variables.
- ▶ Fonctionnel (LISP, OCaml, Haskell, Scala...)
  - ▶ Évaluation d'expressions sans dépendance de la mémoire (pas d'effet de bords).

# Paradigmes de programmations

Paradigme de programmation = Modèle de programmation

- ▶ Impératif (Fortran, C...)
  - ▶ Variables mutables + séquence d'instructions qui manipulent les variables.
- ▶ Fonctionnel (LISP, OCaml, Haskell, Scala...)
  - ▶ Évaluation d'expressions sans dépendance de la mémoire (pas d'effet de bords).
- ▶ Objet (Java, C++...)
  - ▶ Réutilisation d'unités abstraites qui remplissent un rôle spécifique

# Paradigmes de programmations

Paradigme de programmation = Modèle de programmation

- ▶ Impératif (Fortran, C...)
  - ▶ Variables mutables + séquence d'instructions qui manipulent les variables.
- ▶ Fonctionnel (LISP, OCaml, Haskell, Scala...)
  - ▶ Évaluation d'expressions sans dépendance de la mémoire (pas d'effet de bords).
- ▶ Objet (Java, C++...)
  - ▶ Réutilisation d'unités abstraites qui remplissent un rôle spécifique
- ▶ Non exclusif : la plupart des langages sont *Multi-paradigme*



# Exemples de paradigmes

## Différents paradigmes ont différents objectifs

- ▶ Simplicité du code source
- ▶ Expressivité
- ▶ Réutilisabilité du code
- ▶ Facilité de compilation/optimisation
- ▶ Facilité de parallélisation/distribution
- ▶ ...

# Exemples de paradigmes

## Différents paradigmes ont différents objectifs

- ▶ Simplicité du code source
- ▶ Expressivité
- ▶ Réutilisabilité du code
- ▶ Facilité de compilation/optimisation
- ▶ Facilité de parallélisation/distribution
- ▶ ...

La POO favorise le **découplage** et la **réutilisabilité** du code

- ▶ Concepts de base : objet, héritage, délégation, polymorphisme

# Pourquoi l'objet ?

- ▶ Abstraction.
  - ▶ Séparation entre définition et implémentation.
- ▶ Réutilisation.
  - ▶ Conception par classe pour réutilisation.
  - ▶ Cache des détails d'implémentation.
- ▶ Extension / Spécialisation.  
(Propriété de *tolérance à l'élaboration* (J. McCarthy))
  - ▶ Via l'héritage pour des cas particuliers.

# Programmation Objet - Bonnes pratiques

- ▶ Responsabilité : 1 par objet.
- ▶ Encapsulation : protection des données de l'extérieur.
- ▶ Localité : une fonction à un seul endroit.

# Programmation Objet - A éviter

- ▶ L'effet papillon :
  - ▶ Une petite modification entraîne un gros problème.
- ▶ Le copier/coller :
  - ▶ Si bug dans le code de départ ?
- ▶ L'objet Dieu :
  - ▶ Fait tout mais...
- ▶ Les spaghettis
  - ▶ Les lasagnes sont mieux.

# Cours 1 : Concepts

Rappels...

Le paradigme de programmation objet

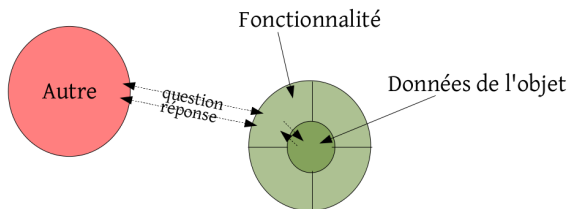
**Concepts de la programmation objet**

Import

Maven

# Encapsulation

- ▶ Suppose l'existence de l'intérieur et de l'extérieur.
- ▶ Permet de contrôler l'accès aux composants critiques de l'objet (variables membres)
- ▶ L'objet n'expose que certaines **fonctionnalités** à l'extérieur (méthodes).



# Champs privés

- ▶ Les champs d'un objet définissent l'état d'un objet
- ▶ Seule une méthode de l'objet peut en changer l'état : **champs privés**.



# Champs privés

- ▶ Les champs d'un objet définissent l'état d'un objet
- ▶ Seule une méthode de l'objet peut en changer l'état : **champs privés**.

```
1 public class Point {  
2     private int x; //privé !  
3     public void setX(int x){  
4         this.x=x;  
5     }  
6 }
```

```
1 public class Main {  
2     public static void main(String[] args) {  
3         Point point = new Point();  
4         point.x = 3; // compile pas  
5         point.setX(3); //ok  
6     }  
7 }
```

Pareil en mieux

- ▶ -1 point par champ non privé dans le projet (si dans une classe publique).
- ▶ Les méthodes sont public si on veut y accéder de l'extérieur, private sinon.

# Encapsulation

- ▶ Est le principe de la POO.
  - ▶ Aide à la conception.
    - ▶ 1 objet = 1 responsabilité.
  - ▶ Aide au debug.
    - ▶ On sait où un objet est modifié.
  - ▶ Aide à l'évolution/maintenance.
    - ▶ Abstraction du code (slide suivante).

# Abstraction

- ▶ Suppose l'existence d'un **développeur** et d'un **utilisateur**
- ▶ Permet à l'utilisateur d'utiliser un objet sans **connaître** ou **dépendre** son de son fonctionnement interne
- ▶ Le développeur :
  - ▶ décide d'un ensemble de fonctionnalités pour l'objet
  - ▶ définit une interface (méthodes publiques)
  - ▶ implémente et teste ces fonctionnalités
  - ▶ fait en sorte que l'utilisateur ne corrompe pas accidentellement l'objet
- ▶ L'utilisateur :
  - ▶ Utilise l'objet pour implémenter des fonctionnalités de plus haut niveau

Le développeur et l'utilisateur peuvent être la même personne

# Abstraction (exemple)

```
1 public class Point {  
2     private double x ; //passe  
3     double  
4     public void setX(int x){  
5         this.x=x ;  
6     }  
7     public void setX(double x){  
8         this.x=x ;  
9     }  
}
```

```
1 public class Main {  
2     public static void main(String[] args)  
3     {  
4         Point point = new Point() ;  
5         point.setX(3) ; //setX(int) appelé  
6     }  
}
```

Pas besoin de modifier le main

- Pour l'**utilisateur** : Possibilité d'utiliser setX() sans connaître les détails de l'implémentation (sans connaître le type de x)
- Pour le **développeur** : Possibilité de changer la représentation interne sans changer l'interface

# Etat d'un objet

- ▶ Un objet doit toujours être dans un état valide.
  - ▶ Une méthode qui modifie l'objet doit le laisser valide.
  - ▶ Un constructeur initialise un objet valide.
- ▶ Exemple : classe représentant un point en coordonnées polaires : **distance toujours positive.**

# (Im)mutabilité

## Illustration :

### ► Sortie ?

```
1 public class Out {  
2     public void someMethod() {  
3         Point p = new Point(1,0);  
4         if(distanceToOrigin() == 1)  
5             System.out.println(p  
6                 + " is on the unit circle ")  
7         else{  
8             ...  
9         }  
10    }  
11 }
```

```
1 public class Point {  
2     private double x;  
3     private double y;  
4     ...  
5     public Point(double x, double y) {  
6         this.x = x;  
7         this.y = y;  
8     }  
9  
10    public double distanceToOrigin(){  
11        return Math.sqrt(x*x + y*y);  
12    }  
13  
14    public String toString() {  
15        return x+", "+y;  
16    }  
17 }
```

# (Im)mutabilité

## Illustration :

### ► Sortie ?

```
1 public class Out {  
2     public void someMethod() {  
3         Point p = new Point(1,0) ;  
4         if(distanceToOrigin() == 1)  
5             System.out.println(p  
6                 + " is on the unit circle ")  
7         else{  
8             ...  
9         }  
10    }  
11 }
```

```
1 public class Point {  
2     private double x ;  
3     private double y ;  
4     ...  
5     public Point(double x, double y) {  
6         this.x = x ;  
7         this.y = y ;  
8     }  
9  
10    public double distanceToOrigin(){  
11        return Math.sqrt(x*x + y*y) ;  
12    }  
13  
14    public String toString() {  
15        return x+", "+y ;  
16    }  
17 }
```

1,0 is on the unit circle

# (Im)mutabilité

## Illustration :

- Sortie ?
- Et là ?

```
1 public class Out {  
2     public void someMethod() {  
3         Point p = new Point(1,0);  
4         if(distanceToOrigin() == 1)  
5             System.out.println(p  
6                 + " is on the unit circle ")  
7         else{  
8             ...  
9         }  
10    }  
11 }
```

```
1 public class Point {  
2     private double x;  
3     private double y;  
4     ...  
5     public Point(double x, double y) {  
6         this.x = x;  
7         this.y = y;  
8     }  
9     public double distanceToOrigin(){  
10        return Math.sqrt(x*x + y*y);  
11    }  
12    public String toString() {  
13        x=3;  
14        return x+", "+y;  
15    }  
16 }
```



# (Im)mutabilité

## Illustration :

- Sortie ?
- Et là ?

```
1 public class Out {  
2     public void someMethod() {  
3         Point p = new Point(1,0);  
4         if(distanceToOrigin() == 1)  
5             System.out.println(p  
6                 + " is on the unit circle ")  
7         else{  
8             ...  
9         }  
10    }  
11 }
```

```
1 public class Point {  
2     private double x;  
3     private double y;  
4     ...  
5     public Point(double x, double y) {  
6         this.x = x;  
7         this.y = y;  
8     }  
9     public double distanceToOrigin(){  
10        return Math.sqrt(x*x + y*y);  
11    }  
12    public String toString() {  
13        x=3;  
14        return x+", "+y;  
15    }  
16 }
```

3,0 is on the unit circle (**Faux !**)

# Moralité de l'exemple

- ▶ Vous avez considéré `Point` comme ne pouvant pas changer de valeur après utilisation (**non mutable**).
- ▶ Une méthode peut modifier un objet **mutable**.
- ▶ Ce qui était vrai à l'instant  $t$  n'est pas forcément vrai à l'instant  $t + 1$ .
- ▶ Choix mutable/non mutable important.

# Final

- ▶ Champ non mutable : `final`.
- ▶ A faire par défaut !

```
1 public class Point {  
2     private final double x ;  
3     private final double y ;  
4     public Point(double x, double y) {  
5         this.x = x ;  
6         this.y = y ;  
7     }  
8     public String toString() {  
9         x=3 ; //compile pas  
10        return x+", "+y ;  
11    }  
12 }
```

# Final - Non mutable

- Tous les champs avec `final` : insuffisant pour considérer l'objet non mutable.

```
1 public class Circle {  
2     private final Point center ;  
3     public Circle(Point center, int r) {  
4         this.center = center ;  
5     }  
6     public void translate(int dx, int dy) {  
7         center.translate(dx,dy) ;  
8     }  
9 }
```

- Pourquoi ?

# Final - Non mutable

- Tous les champs avec `final` : insuffisant pour considérer l'objet non mutable.

```
1 public class Circle {  
2     private final Point center ;  
3     public Circle(Point center, int r) {  
4         this.center = center ;  
5     }  
6     public void translate(int dx, int dy) {  
7         center.translate(dx,dy) ;  
8     }  
9 }
```

- Pourquoi ?
- Objets référencés doivent aussi être non mutables.

# Non mutable et modification

- Pour modifier un objet non mutable, il faut en créer un nouveau et remplacer la référence.

```
1 public class Point {  
2     private double x ;  
3     private double y ;  
4     public Point(double x, double y) {  
5         this.x = x ;  
6         this.y = y ;  
7     }  
8     public void translate(double dx,  
9         double dy) {  
10         x += dx ;  
11         y += dy ;  
12     }  
}
```

**Mutable**

```
1 public class Point {  
2     private final double x ;  
3     private final double y ;  
4     public Point(double x, double y) {  
5         this.x = x ;  
6         this.y = y ;  
7     }  
8     public Point translate(double dx,  
9         double dy) {  
10         return new Point(x+dx,y+dy) ;  
11     }  
}
```

**Non mutable**

# Non mutable et modification

- ▶ Par exemple, String est non mutable.
- ▶ Problème, impossible de dire au compilateur de récupérer la valeur de retour.
- ▶ Erreur classique du débutant :

```
1 String s = "M1 miage";  
2 s.toUpperCase();  
3 System.out.println(s); //M1 miage
```

# Mutable ou pas ?

- ▶ En pratique :
  - ▶ Les petits objets sont non mutables, le GC les recycle facilement.
  - ▶ Les gros objets (tableaux, listes...) sont mutables pour des questions de perfs.



# Non mutable avec champ mutable

- Comment créer un objet non mutable si on utilise un champ mutable ?

```
1 public class Dog {  
2     private final StringBuilder name; //SB mutable  
3     public Dog(StringBuilder name) {  
4         this.name = name;  
5     }  
6     public StringBuilder getName() {  
7         return name;  
8     }  
9     public static void main(String[] args) {  
10        StringBuilder name= new StringBuilder("Milou");  
11        Dog dog = new Dog(name);  
12        name.reverse();  
13        System.out.println(dog.getName()); //uoliM : meme reference  
14    }  
15 }
```

# Non mutable avec champ mutable

- Comment créer un objet non mutable si on utilise un champ mutable ?
- Copie défensive.

```
1 public class Dog {  
2     private final StringBuilder name ; //SB mutable  
3     public Dog(StringBuilder name) {  
4         this.name = new StringBuilder(name) ; //copie def  
5     }  
6     public StringBuilder getName() {  
7         return name ;  
8     }  
9     public static void main(String[] args) {  
10         StringBuilder name= new StringBuilder("Milou") ;  
11         Dog dog = new Dog(name) ;  
12         name.reverse() ;  
13         System.out.println(dog.getName()) ; //Milou  
14     }  
15 }
```

# Non mutable avec champ mutable

- ▶ Comment créer un objet non mutable si on utilise un champ mutable ?
- ▶ Copie défensive.
- ▶ Pas qu'à la création !

```
1 public class Dog {  
2     private final StringBuilder name ; //SB mutable  
3     public Dog(StringBuilder name) {  
4         this.name = new StringBuilder(name) ;  
5     }  
6     public StringBuilder getName() {  
7         return name ;  
8     }  
9     public static void main(String[] args) {  
10        StringBuilder name= new StringBuilder("Milou") ;  
11        Dog dog = new Dog(name) ;  
12        dog.getName().reverse() ; //référence récupérée et modifiée  
13        System.out.println(dog.getName()) ; //uoliM  
14    }  
15 }
```

# Non mutable avec champ mutable

- ▶ Comment créer un objet non mutable si on utilise un champ mutable ?
- ▶ Copie défensive.
- ▶ Pas qu'à la création !
- ▶ Aussi à l'envoi de références.

```
1 public class Dog {  
2     private final StringBuilder name ; //SB mutable  
3     public Dog(StringBuilder name) {  
4         this.name = new StringBuilder(name) ;  
5     }  
6     public StringBuilder getName() {  
7         return new StringBuilder(name) ;  
8     }  
9     public static void main(String[] args) {  
10        StringBuilder name= new StringBuilder("Milou") ;  
11        Dog dog = new Dog(name) ;  
12        dog.getName().reverse() ;  
13        System.out.println(dog.getName()) ; //Milou  
14    }  
15 }
```

# Non mutable et méthode

- ▶ Méthode utilisant en paramètre un objet non mutable : risque de modification par l'extérieur pendant ou après l'exécution.
- ▶ Solution :
  - ▶ Recevoir / créer une copie défensive.
  - ▶ Deal with it...

# Cours 1 : Concepts

Rappels...

Le paradigme de programmation objet

Concepts de la programmation objet

**Import**

Maven

# Import

- Pour utiliser une classe, il faut son nom complet, avec son package (lourd).

```
1 java.util.Date d = new java.util.Date() ;
```

# Import

- Pour utiliser une classe, il faut son nom complet, avec son package (lourd).

```
1 java.util.Date d = new java.util.Date() ;
```

- Utilisation de `import` pour que le compilateur comprenne que `Date` a pour vrai nom `java.util.Date`.
- Même code généré.

```
1 import java.util.Date ;  
2 Date d = new Date() ;
```



# Import\*

- ▶ Compilateur peut regarder dans le package si classe non trouvée.
- ▶ Si deux packages possèdent une classe avec le même nom et que les deux sont importés avec `import*`, ambiguïté.
- ▶ On la lève en important explicitement une des classes.

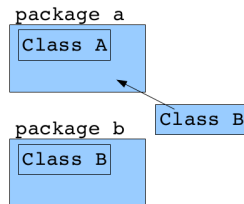
```
1 import java.util.* ;
2 import java.awt.* ;
3 public class Foo {
4     public void m() {
5         List l = ... //KO
6     }
7 }
```

```
1 import java.util.* ;
2 import java.awt.* ;
3 import java.util.List ;
4 public class Foo {
5     public void m() {
6         List l = ... //OK
7     }
8 }
```

# Import\* et maintenance

- Problème potentiel si ajout de classes dans un package.

```
1 import a.*;  
2 import b.*;  
3  
4 public class Foo {  
5     public void m() {  
6         A a = new A();  
7         B b = new B();  
8     }  
9 }
```



# Sources

- ▶ Classes dans des **packages**.
- ▶ Package par défaut (si rien indiqué) à ne pas utiliser sauf pour des tests.
- ▶ Package `a.b.c` placé dans le répertoire `a/b/c`.

# Javadoc

- ▶ Située entre `/**` et `*/`.
- ▶ Tags :
  - ▶ `@see` lien vers une méthode, une classe ou un champ.
  - ▶ `@param x` : description du paramètre `x`.
  - ▶ `@return` : description de la valeur de retour.
  - ▶ `@throws E` : description des cas où `E` est levée.
- ▶ A utiliser !!

# Cours 1 : Concepts

Rappels...

Le paradigme de programmation objet

Concepts de la programmation objet

Import

**Maven**