Java Avancé

Cours 1 : Concepts

Hossein Khani

Slides d'après Florian Sikora

Hossein.Khani@dauphine.fr
Hossein.Khani@lamsade.dauphine.fr

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

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 × 1.5h
→ Exam : 60% de la note de l'UE

TD : 14 × 1.5h

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

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

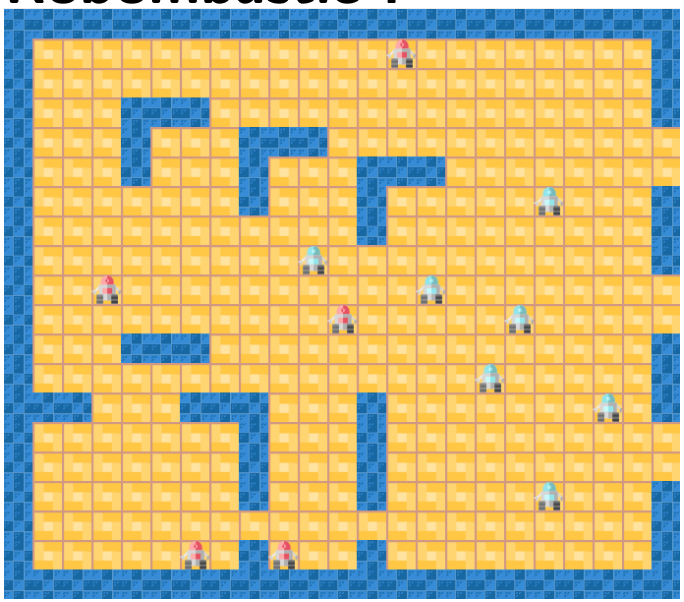
Projet :

- 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

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)`.
  - Charactère 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    void printEtuMoy() {
10     System.out.println("Moyenne " +
           moyenne) ; // this implicite
11   }
12 }
```

```
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`

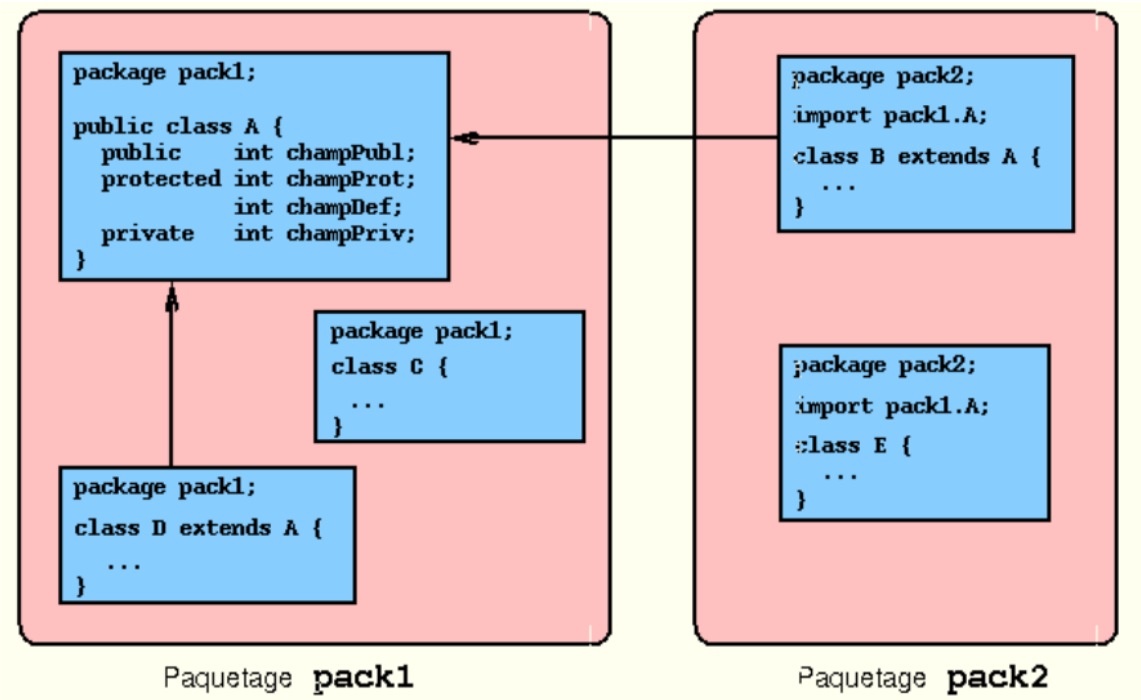
Visibilité

- 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

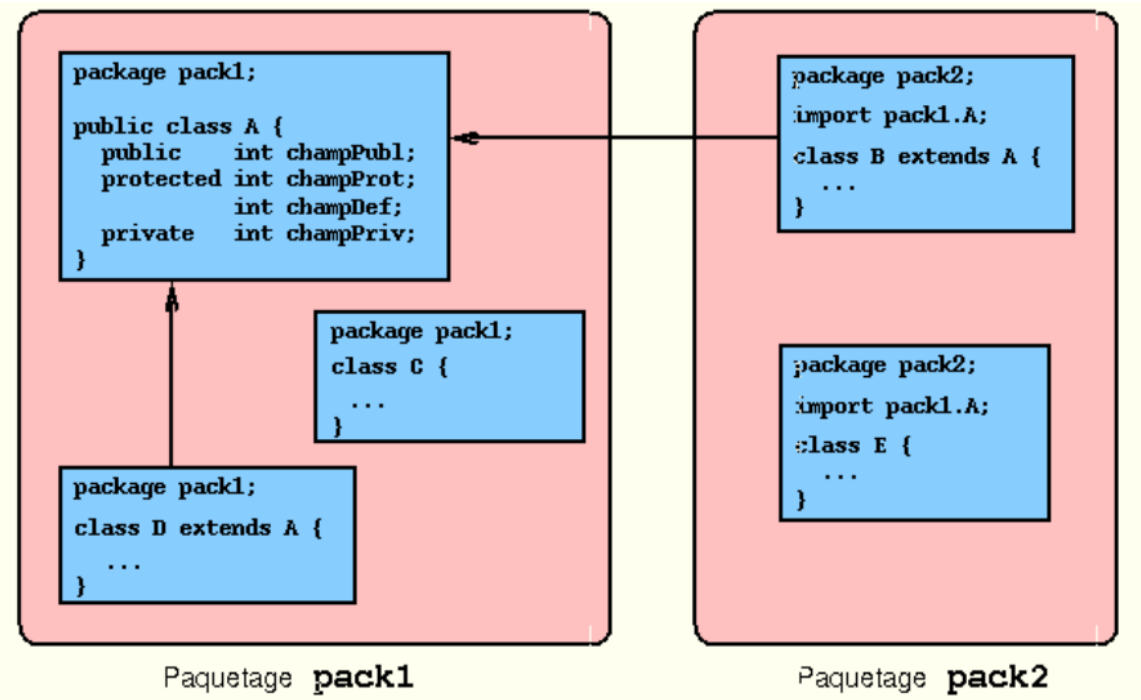
- 4 modificateurs de visibilité pour les membres d'une classe.
  - `private`
    - Visible que dans la classe.
  - Sans modificateur.
    - Visible par les classes du même package.
  - `protected`
    - Visible par les classes héritées et celles du même package.
  - `public`
    - Visible par tout le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

Visibilité



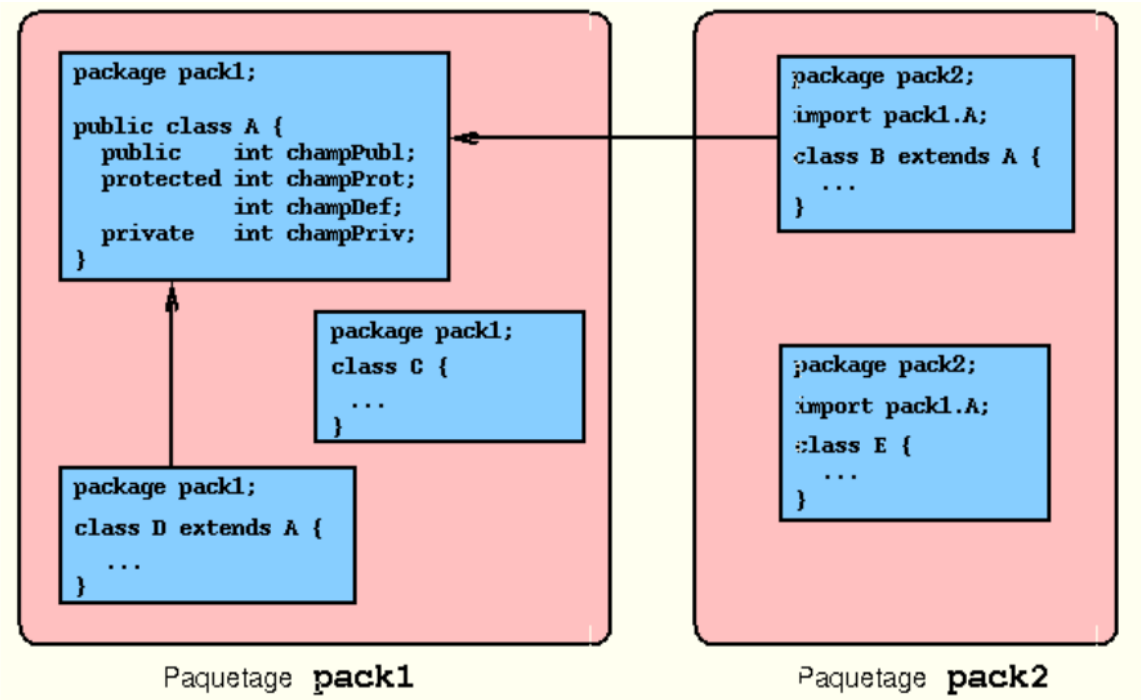
Dans :	A	B	C	D	E
champPubl					
champProt					
champ					
champPriv					

Visibilité



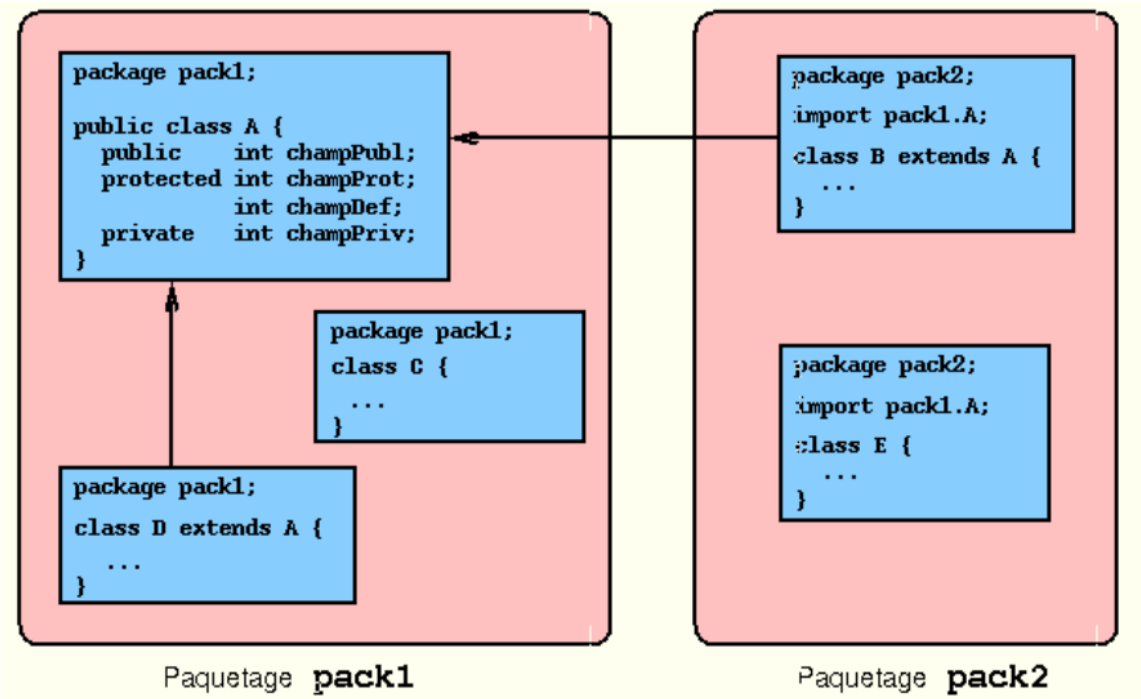
Dans :	A	B	C	D	E
champPubl	Oui	Oui	Oui	Oui	Oui
champProt					
champ					
champPriv					

Visibilité



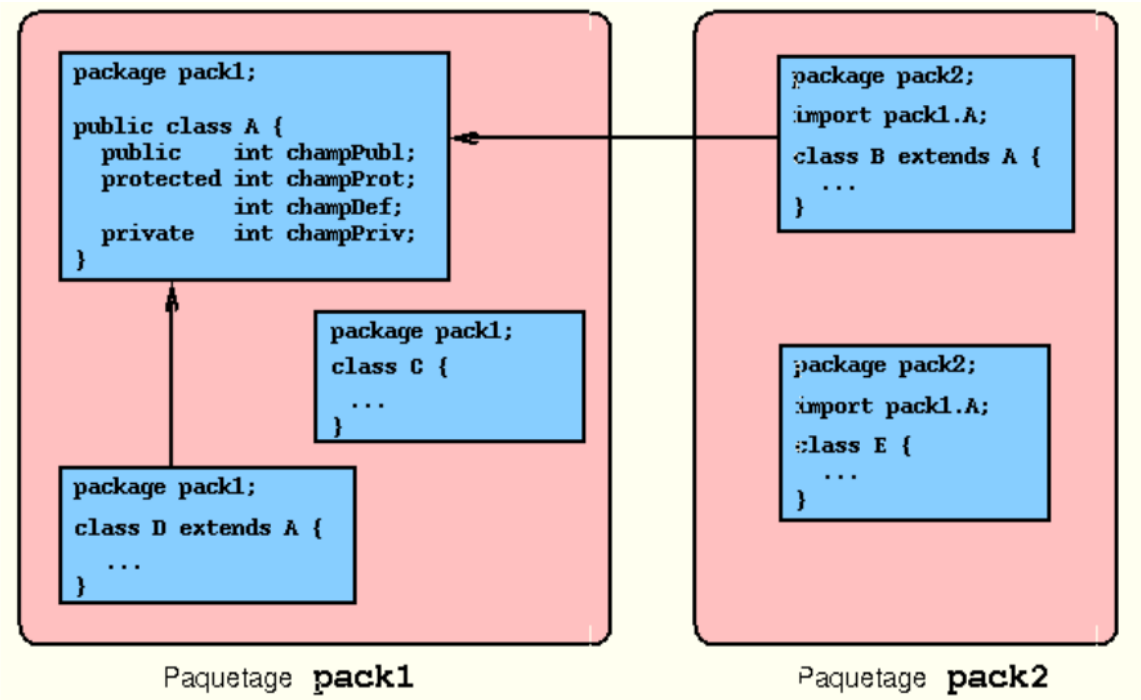
Dans :	A	B	C	D	E
champPubl	Oui	Oui	Oui	Oui	Oui
champProt	Oui	Oui	Oui	Oui	Non
champ					
champPriv					

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 - Régles

- 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

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 *tolérence 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.

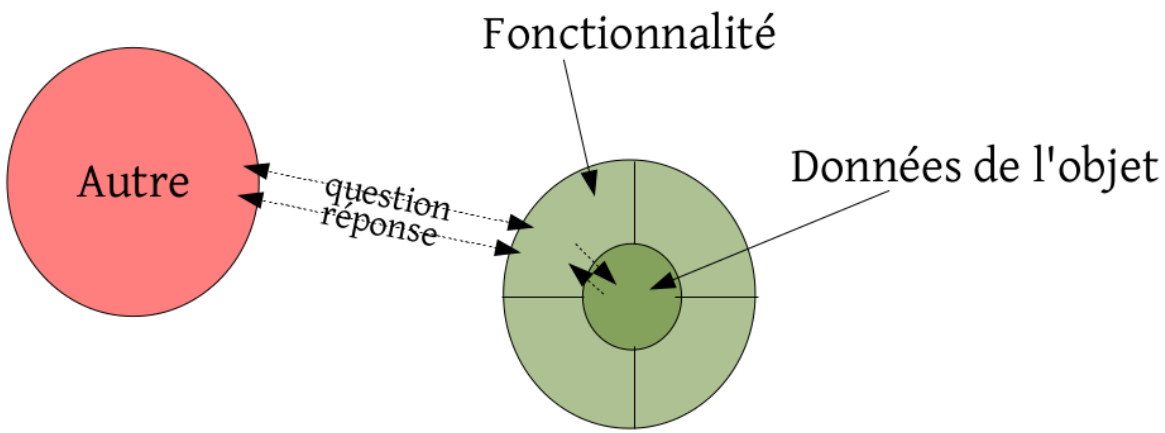
Programmation Objet - A éviter

- 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

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**.

```java
public class Point {
  private int x ; //privé !
  public void setX(int x){
    this.x=x ;
  }
}
```

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

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)

```java
public class Point {
  private double x ; //passe
        double
  public void setX(int x){
    this.x=x ;
  }
  public void setX(double x){
    this.x=x ;
  }
}
```

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

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,
           double dy) {
9      x += dx ;
10     y += dy ;
11   }
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,
           double dy) {
9      return new Point(x+dx,y+dy) ;
10   }
11 }
```

Non mutable

Non mutable et modification

- Par exemple, `String` est non mutable.
- Problème, impossible de dire au compilo 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 récuperée 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

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 trouvé.
- 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 }
```

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 2 : Polymorphisme et compagnie

Cours 2 : Polymorphisme et compagnie

Types

- Typage à la compilation et/ou l'exécution.
- Seulement à la compilation :
  - C++, OCaml...
- Seulement exécution :
  - PHP, Javascript, Python, Ruby...
- Les deux :
  - Java, C#...

Types et Objet

```
1 Object o = new Integer(3) ;
```

- Qu'est-ce qui est le type de l'objet, qu'est-ce qui est la classe de l'objet ?

Types et Objet

```
1 Object o = new Integer(3) ;
```

- Qu'est-ce qui est le type de l'objet, qu'est-ce qui est la classe de l'objet ?
- **Type** de o : son interface : ensemble des méthodes pouvant être appelées (ici `Object`).
  - Connu et vu par le compilateur.
- **Classe** de o : ensemble des propriétés et méthodes utilisées pour créer l'objet (ici `Integer`).
  - Connu par la VM.

Types et Objet

```
1 Object o = new Integer(3) ;
```

- Qu'est-ce qui est le type de l'objet, qu'est-ce qui est la classe de l'objet ?
- **Type** de o : son interface : ensemble des méthodes pouvant être appelées (ici `Object`).
  - Connu et vu par le compilateur.
- **Classe** de o : ensemble des propriétés et méthodes utilisées pour créer l'objet (ici `Integer`).
  - Connu par la VM.
  - Opérations dynamiques en ont besoin.
    - Création, `getClass()`, `instanceof`...

Object

```
1 Miage m = new Miage(2014) ;
2 System.out.println(m) ; //Miage@40f92a41
```

- Comment peut-on appeler une méthode avec un objet de type inconnu lors de son écriture ?
- Signature de println : `public void println(Object x) ;`

Object

```
1 Miage m = new Miage(2014) ;
2 System.out.println(m) ; //Miage@40f92a41
```

- Comment peut-on appeler une méthode avec un objet de type inconnu lors de son écriture ?
- Signature de println : `public void println(Object x) ;`
- Miage **hérite** d'Object.
- Objets de la classe Miage peuvent être manipulés par des références sur Object : `Object o = new Miage(2014)`.
- Miage est un sous-type d'Object, Object est un super-type de Miage.

Object

- Toutes les classes héritent directement (ajouté par le compilateur quand on définit une classe) ou indirectement (héritage d'une classe) de `java.lang.Object`.
- 3 méthodes universelles :

Object

- Toutes les classes héritent directement (ajouté par le compilateur quand on définit une classe) ou indirectement (héritage d'une classe) de `java.lang.Object`.
- 3 méthodes universelles :
  - `toString()` : affichage debug de l'objet.
  - `equals()` : renvoie vrai si deux objets sont égaux.
  - `hashCode()` : "résumé" de l'objet sous forme d'entier.

Object

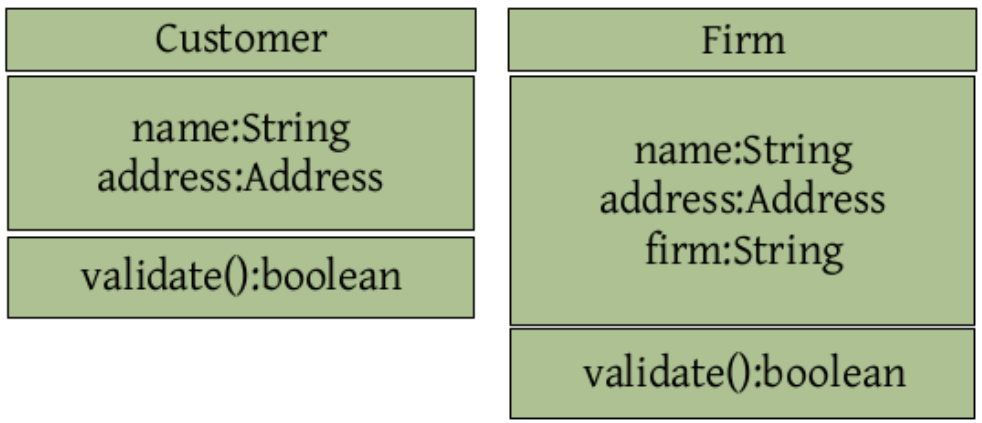
- Une implémentation par défaut des 3 méthodes.

```
1  public class Miage {
2    private final int year ;
3    public Miage(int year) {
4      this.year = year ;
5    }
6    public static void main(String[] args) {
7      Miage m = new Miage(2014) ;
8      System.out.println(m.toString()) ; //Miage@264532ba : class+@+hashCode()
9      System.out.println(m.equals(new Miage(2014))) ; // false car ==
10     System.out.println(m.equals(m)) ; //true car ==
11     System.out.println(m.hashCode()) ; //random sur 24bits
12   }
13 }
```

Cours 2 : Polymorphisme et compagnie

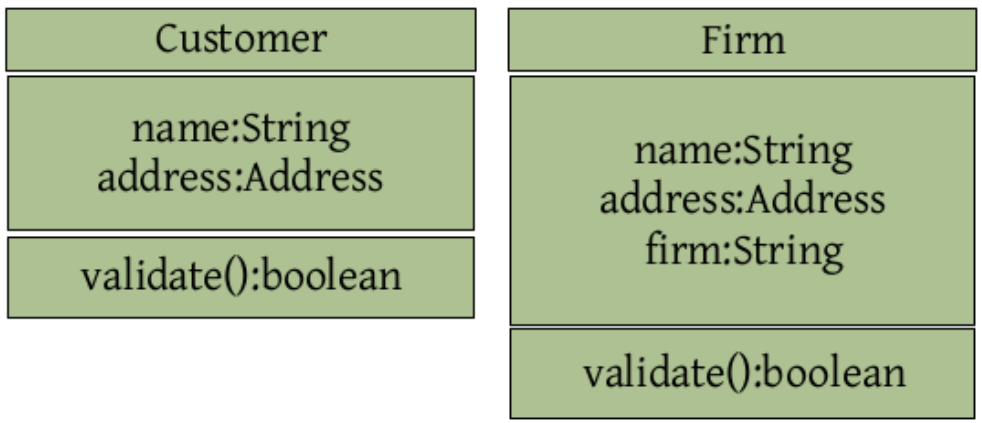
Héritage - Exemple

- Application de commerce en ligne.
- Deux types de clients :
  1. Des particuliers.
  2. Des entreprises.
- Design possible :



Héritage - Exemple

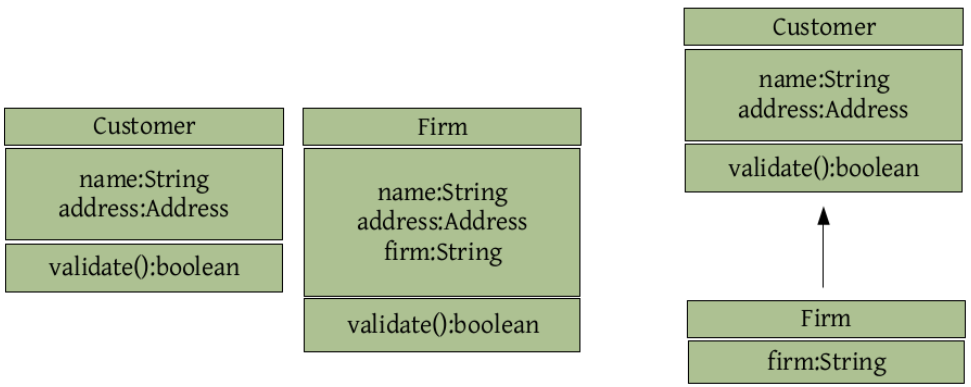
- Application de commerce en ligne.
- Deux types de clients :
  1. Des particuliers.
  2. Des entreprises.
- Design possible :



- Similarités ?

Héritage - Exemple

- Firm est comme un Customer, mais avec un champ en plus.



- Customer super-classe.
- Firm sous-classe.

Héritage - code

- Sans héritage.

```
1  public class Customer {
2    private final String name ;
3    private final Address address ;
4
5    public Customer(String name,
           Address address) {
6      this.name = name ;
7      this.address = address ;
8    }
9  }
```

```
1  public class Firm {
2    private final String name ;
3    private final Address address ;
4    private final String firm ;
5
6    public Firm(String name, Address
           address, String firm) {
7      this.name = name ;
8      this.address = address ;
9      this.firm = firm ;
10   }
11 }
```

Héritage - code

- Avec héritage.

```
1 public class Customer {
2   private final String name ;
3   private final Address address ;
4
5   public Customer(String name,
          Address address) {
6     this.name = name ;
7     this.address = address ;
8   }
9 }
```

```
1  //mot clef extends
2  public class Firm extends Customer {
3    private final String firm ;
4    //autres champs hérités
5
6    public Firm(String name, Address
           address, String firm) {
7      this.name = name ;
8      this.address = address ;
9      //compile pas
10     this.firm = firm ;
11   }
12 }
```

Héritage - code

- Avec héritage.

```
1 public class Customer {
2   private final String name ;
3   private final Address address ;
4
5   public Customer(String name,
          Address address) {
6     this.name = name ;
7     this.address = address ;
8   }
9 }
```

```
1 public class Firm extends Customer {
2   private final String firm ;
3
4   public Firm(String name, Address
          address, String firm) {
5     //appel au constructeur de la
            super-classe
6     super(name, address) ;
7     this.firm = firm ;
8   }
9 }
```

Héritage - code

- Champs et méthodes hérités !

```
1  public class Customer {
2    private final String name ;
3    private final Address address ;
4
5    public Customer(String name,
           Address address) {
6      this.name = name ;
7      this.address = address ;
8    }
9
10   public boolean validate() {
11     return DB.exists(address) &&  !
           name.isEmpty() ;
12   }
13 }
```

```
1 public class Firm extends Customer {
2   private final String firm ;
3
4   public Firm(String name, Address
        address, String firm) {
5     super(name, address) ;
6     this.firm = firm ;
7   }
8   //bug potentiel !
9 }
```

- Pourquoi ?

Héritage - code

- Champs et méthodes hérités !

```
1  public class Customer {
2    private final String name ;
3    private final Address address ;
4
5    public Customer(String name,
           Address address) {
6      this.name = name ;
7      this.address = address ;
8    }
9
10   public boolean validate() {
11     return DB.exists(address) &&  !
           name.isEmpty() ;
12   }
13 }
```

```
1 public class Firm extends Customer {
2   private final String firm ;
3
4   public Firm(String name, Address
        address, String firm) {
5     super(name, address) ;
6     this.firm = firm ;
7   }
8   //bug potentiel !
9 }
```

- Pourquoi ?
- On hérite de validate()...qui ne vérifie pas le champ firm !

```
1 Firm f = new Firm(..) ;
2 if(firm.validate()){...
```

Héritage - code

- Bug ?

```java
public class Customer {
  private final String name ;
  private final Address address ;

  public Customer(String name,
        Address address) {
    this.name = name ;
    this.address = address ;
  }

  public boolean validate() {
    return DB.exists(address) &&  !
          name.isEmpty() ;
  }
}
```

```java
public class Firm extends Customer {
  private final String firm ;

  public Firm(String name, Address
        address, String firm) {
    super(name, address) ;
    this.firm = firm ;
  }
  @Override
  public boolean validate() {
    return DB.exists(address) &&  !
          name.isEmpty() &&  !firm.
          isEmpty() ;
  }
}
```

Héritage - code

- Bug ?

```
1  public class Customer {
2    private final String name ;
3    private final Address address ;
4
5    public Customer(String name,
           Address address) {
6      this.name = name ;
7      this.address = address ;
8    }
9
10   public boolean validate() {
11     return DB.exists(address) &&  !
             name.isEmpty() ;
12   }
13 }
```

```
1  public class Firm extends Customer {
2    private final String firm ;
3
4    public Firm(String name, Address
           address, String firm) {
5      super(name, address) ;
6      this.firm = firm ;
7    }
8    @Override
9    public boolean validate() {
10     return DB.exists(address) &&  !
             name.isEmpty() &&  !firm.
             isEmpty() ;
11   }
12 }
```

- Compile pas !

Héritage - code

- Bug ?

```
1  public class Customer {
2    private final String name ;
3    private final Address address ;
4
5    public Customer(String name,
           Address address) {
6      this.name = name ;
7      this.address = address ;
8    }
9
10   public boolean validate() {
11     return DB.exists(address) &&  !
             name.isEmpty() ;
12   }
13 }
```

```
1  public class Firm extends Customer {
2    private final String firm ;
3
4    public Firm(String name, Address
           address, String firm) {
5      super(name, address) ;
6      this.firm = firm ;
7    }
8    @Override
9    public boolean validate() {
10     return super.validate() &&  !firm.
             isEmpty() ;
11   }
12 }
```

- On délègue à la super-classe la validation des champs de Customer.

Héritage

- Bien en 1960, discutable de nos jours.
  - Ré-utiliser un algo : utiliser une méthode `static` dans une classe public...
  - Hériter implique aussi :
    - Le sous-typage.
    - L'héritage de **toutes** les méthodes...
    - Ne peut pas changer l'implémentation de la sous-classe si on ne peut pas toucher celle de la super-classe.

Héritage

- Doit redéfinir **toutes** les méthodes...

```
1  public class EmployeesList extends ArrayList<Employee> {
2    @Override
3    public boolean add(Employee e) {
4      Objects.requireNonNull(e) ; //pas de null dans ma liste
5      return super.add(e) ;
6    }
7    public static void main(String[] args) {
8      List<Employee> l = new ArrayList<Employee>() ;
9      l.add(null) ;
10     EmployeesList el = new EmployeesList() ;
11     el.addAll(l) ;
12     System.out.println(el) ; //[null]
13   }
14 }
```

Héritage

- Solution : déléguer.
- Pas le lien fort entre sous-classe et super-classe.

```
1 public class EmployeesList {
2   private final ArrayList<Employee> al = new ArrayList<Employee>() ;
3   private void add(Employee e) {
4     Objects.requireNonNull(e) ;
5     al.add(e) ; //delegation
6   }
7 }
```

Héritage et constructeurs

- Constructeurs non hérités ($\neq$ méthodes).
- Un constructeur doit faire appel au constructeur de sa super classe en **première** instruction.
- Fait implicitement si constructeur par défaut (sans argument).

```
1 public class Fruit {
2 }
```

```
1 public class Orange extends Fruit {
2   private final String s ;
3   public Orange(int a, String s) {
4     //ok
5     this.s=s ;
6   }
7 }
```

Héritage et constructeurs

- Constructeurs non hérités ($\neq$ méthodes).
- Un constructeur doit faire appel au constructeur de sa super classe en **première** instruction.
- Fait implicitement si constructeur par défaut (sans argument).
  - Attention, rappel, si définition d'un constructeur avec argument, plus de constructeur par défaut !

```
1 public class Fruit {
2   public Fruit(int a) {
3   }
4 }
```

```
1 public class Orange extends Fruit {
2   public Orange() {
3     //Compile pas
4     //Implicit super constructor Fruit() is
          undefined.
5     //Must explicitly invoke another
          constructor
6   }
7 }
```

Héritage et constructeurs

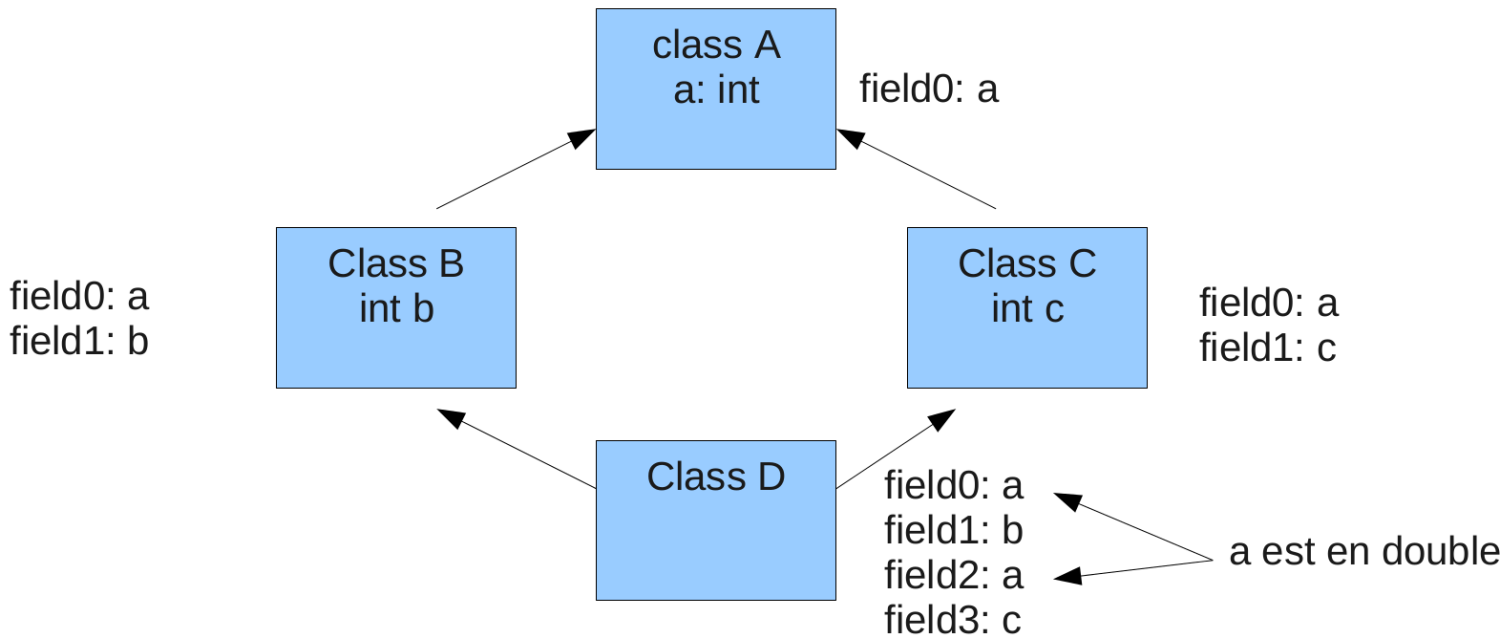
- Constructeurs non hérités ($\neq$ méthodes).
- Un constructeur doit faire appel au constructeur de sa super classe en **première** instruction.
- Fait implicitement si constructeur par défaut (sans argument).
  - Attention, rappel, si définition d'un constructeur avec argument, plus de constructeur par défaut !

```
1 public class Fruit {
2   public Fruit(int a) {
3   }
4 }
```

```
1 public class Orange extends Fruit {
2   private final String s ;
3   public Orange(int a, String s) {
4     super(a) ; //en premier !
5     this.s = s ;
6   }
7 }
```

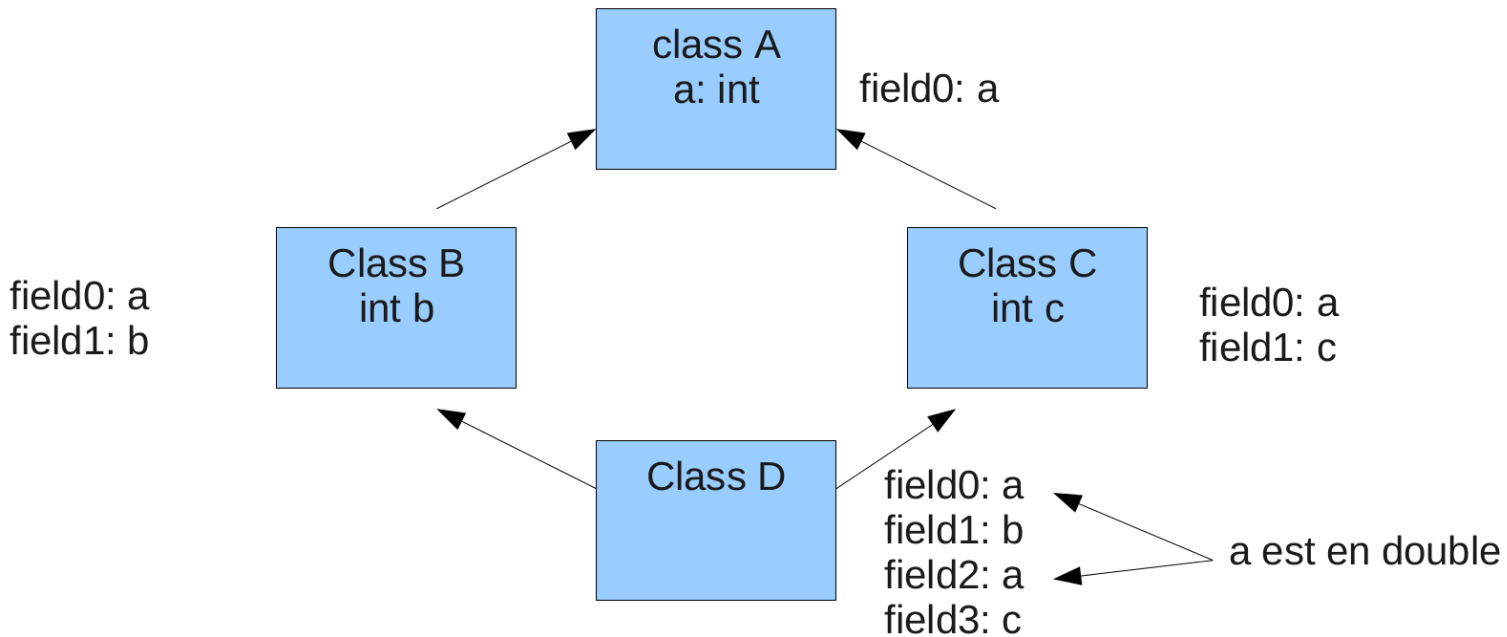
Héritage multiple

- En Java, pas d'héritage multiple...
- Problème du diamant.



Héritage multiple

- En Java, pas d'héritage multiple...
- Problème du diamant.



Avec l'héritage, on devient un objet du type parent **et** on hérite des propriétés du parent.

Interface

- Parfois besoin de voir un objet comme étant deux “trucs”...
- Utiliser des interfaces pour du sous-typage multiple (mot clef `interface`) :
  - Définition d'un type sans son implémentation (méthodes sans code).
  - Interdit d'instancier une interface (car pas de code !).
- Une classe peut fournir l'implémentation de plusieurs interfaces.

  - Pas de problème car pas de code hérité !

Interface

- Parfois besoin de voir un objet comme étant deux “trucs”...
- Utiliser des interfaces pour du sous-typage multiple (mot clef `interface`) :
  - Définition d'un type sans son implémentation (méthodes sans code).
  - Interdit d'instancier une interface (car pas de code !).
- Une classe peut fournir l'implémentation de plusieurs interfaces.
  - Pas de problème car pas de code hérité !
- Amélioration avec Java 8 et les traits (interface avec méthodes mais sans champs), choix à la main en cas de conflit ou méthode par défaut.

Interface

- Pourquoi les interfaces ?

Interface

- Pourquoi les interfaces ?
- Définir "l'interface" entre deux modules du projet.
- Définir une fonctionnalité transversale (ex : `Comparable`, `Cloneable`...).
- Définir un ensemble de fonctionnalités avec plusieurs implémentation possibles (ex : `ArrayList`, `LinkedList`...).

Interface

- Le compilateur vérifie que toutes les méthodes de l'interface sont implémentées par la classe.
- On peut décider de lever une exception si on ne veut pas implémenter cette opération.

```
1  public interface List {
2    public Object get(int index) ;
3    /** optional operation */
4    public void set(int index,Object o) ;
5  }
6
7  public class NullList implements List {
8    public void set(int index,Object o) {
9      throw new UnsupportedOperationException() ;
10   }
11   public Object get(int index) {
12     return null ;
13   }
14 }
```

Interface

- Rien interdit d'ajouter des méthodes dans l'implémentation d'une interface.
  - Difficile de *supprimer* des fonctionnalités

```
1  public interface ReadOnlyHolder<T> {
2    public T get() ;
3  }
4
5  public class Holder<T> implements ReadOnlyHolder<T> {
6    private T t ;
7    public T get() {
8      return t ;
9    }
10   public void set(T t) {
11     this.t=t ; //readOnly  ?   :(
12   }
13 }
```

Interface

- **Méthodes** d'une interface, obligatoirement et implicitement `abstract` et `public`.
- **Champs** d'une interface, obligatoirement et implicitement `final`, `public`, `static`.
- Impossible d'un définir une méthode statique.

Héritage

En résumé,

- **hériter** (avec `extends`) c'est :
  1. Récupérer les champs et les méthodes de la classe mère.
  2. Avoir la possibilité de redéfinir les méthodes de classe mère.
  3. Devenir un **sous-type** de la classe mère
     (Orange est sous-type de Fruit).

- devenir une **interface** (avec `interface`) c'est :
  1. Avoir l'obligation de définir toutes les méthodes de l'interface.
  2. Devenir un **sous-type** de la classe mère.

Héritage

En résumé,

- **hériter** (avec `extends`) c'est :
  1. Récupérer les champs et les méthodes de la classe mère.
  2. Avoir la possibilité de redéfinir les méthodes de classe mère.
  3. Devenir un **sous-type** de la classe mère
     (Orange est sous-type de Fruit).

- devenir une **interface** (avec `interface`) c'est :
  1. Avoir l'obligation de définir toutes les méthodes de l'interface.
  2. Devenir un **sous-type** de la classe mère.

Dans les deux cas, difficile d'utiliser ces méthode pour **supprimer** fonctionnalité existante.

Cours 2 : Polymorphisme et compagnie

Sous-typage

- Substituer un type par un autre.
- A quoi ça sert ?

Sous-typage

- Substituer un type par un autre.
- A quoi ça sert ?
- Utiliser un algorithme écrit pour un certain type et l'utiliser avec un autre.

Sous-typage

- Substituer un type par un autre.
- A quoi ça sert ?
- Utiliser un algorithme écrit pour un certain type et l'utiliser avec un autre.
- Si `Orange` hérite de `Fruit`, alors `Orange` est un sous-type de `Fruit` :
  - Partout où un objet `Fruit` est attendu, on peut lui donner une `Orange`.
  - L'inverse est faux.

Sous-typage

- Existe pour (slides suivantes) :
  - Héritage de classe ou d'interface.
  - Implémentation d'interface.
  - Tableaux d'objets.
  - Types paramétrés.

Sous-typage et héritage

```
1 class A {
2 }
3 class B extends A {
4 }
5 public static void main(String[] args) {
6   A a = new B() ;
7 }
```

- B hérite de A.
- B est un sous-type de A.
- B récupère tous les membres de A.

Sous-typage et interface

```
1  interface I {
2    void m() ;
3  }
4  class A implements I {
5    public void m() {
6    }
7  }
8  public static void mani(String[] args) {
9    I i = new A() ;
10 }
```

- A est un sous-type de I.
- A possède un code pour toutes les méthodes de I.

Sous-typage et tableaux

- Tableaux java sont des sous-types d'Object (et Serializable et Cloneable).
- U[] est un sous type de T[] si U est un sous-type de T, et si U et T ne sont pas primitifs.

```
1 double[] t = new int[2] ;  //ko
2 Number[] tab = new Integer[2] ; //ok
```

Sous-typage et tableaux

- Problème à l'exécution...

```
1 Number[] tab = new Integer[2] ; //ok
2
3 tab[0] = 2 ;
4 tab[1] = 2.2 ;  //java.lang.ArrayStoreException
```

Sous-typage et types paramétrés

- Pas de check à l'exécution pour les types paramétrés.
- `List<String>` n'est pas un sous-type d'une `List<Object>`.
- Voir cours sur les wildcards pour gérer ce problème.

```
1  ArrayList<String> al1 = new ArrayList<>() ;
2  ArrayList<Object> al2 = al1 ; //compile pas
3
4  //si compilait :
5  al2.add(new Integer(3)) ; //compile ! !  :(
6  String s = al1.get(0) ; //Aie, ClassCastException...
```

Cours 2 : Polymorphisme et compagnie

Polymorphisme

- Idée du **polymorphisme** : considérer les fonctionnalités suivant le type réel d'un objet et non le type de la variable où il est référencé.
- **Sous-typage** : stocker un objet comme une variable d'un super-type.

Polymorphisme

- Intérêt ?

Polymorphisme

- Intérêt ?
- Avoir certaines parties d'un algorithme spécialisé en fonction du type réel de l'objet.
- Pas besoin de dispatcheur dans l'algorithme
- PAS DE INSTANCEOF !!

Polymorphisme

```java
public class Fruit {
  public void name() {
    System.out.println("Je suis un
        fruit") ;
  }
}
```

```java
public class Orange extends Fruit {
  @Override
  public void name() {
    System.out.println("Je suis une
        orange") ;
  }
}
```

```java
public class Pamplemousse extends
    Fruit {
  @Override
  public void name() {
    System.out.println("Je suis un
        pamplemousse") ;
  }
}
```

```java
public class Polym {
  public static void main(String[]
      args) {
    Fruit[] fruits = new Fruit[]{new
        Orange(), new Pamplemousse()
        } ;

    for (Fruit f  : fruits) {
      f.name() ;
    }
  }
}
```

- Affiche ?

Polymorphisme

```java
public class Fruit {
  public void name() {
    System.out.println("Je suis un
        fruit") ;
  }
}
```

```java
public class Orange extends Fruit {
  @Override
  public void name() {
    System.out.println("Je suis une
        orange") ;
  }
}
```

```java
public class Pamplemousse extends
    Fruit {
  @Override
  public void name() {
    System.out.println("Je suis un
        pamplemousse") ;
  }
}
```

```java
public class Polym {
  public static void main(String[]
      args) {
    Fruit[] fruits = new Fruit[]{new
        Orange(), new Pamplemousse()
        } ;

    for (Fruit f  : fruits) {
      f.name() ;
    }
  }
}
```

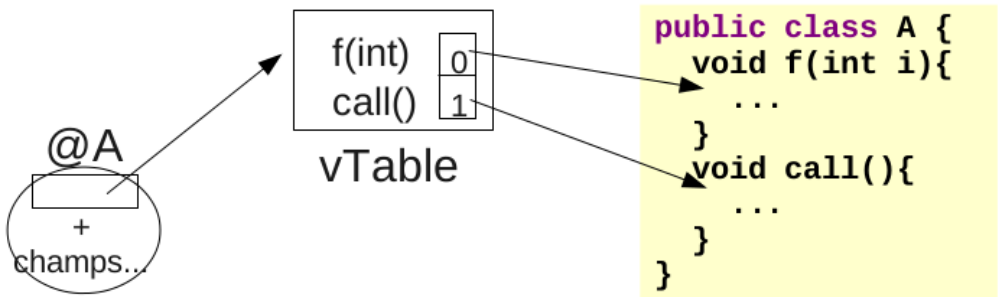
- Affiche ?
  - Je suis une orange
  - Je suis un pamplemousse

Redéfinition de méthodes

- Il y a **redéfinition** de méthode si le site d'appel (ici dans la boucle) peut appeler la méthode redéfinie à la place de celle choisie à la compilation.
- Rédéfinition d'une méthode selon :
  - Son nom.
  - Sa visibilité (au moins aussi grande que l'originale).
  - Sa signature (depuis 1.5, type de retour peut être un sous-type).
  - Ses exceptions (peut être un sous-type ou pas d'exception).
  - Son paramétrage (soit toutes les 2, soit aucune).
- Vérifié par l'annotation `@Override` à la compilation.
  - Compile pas si non respecté.
  - Pas indispensable au polymorphisme.

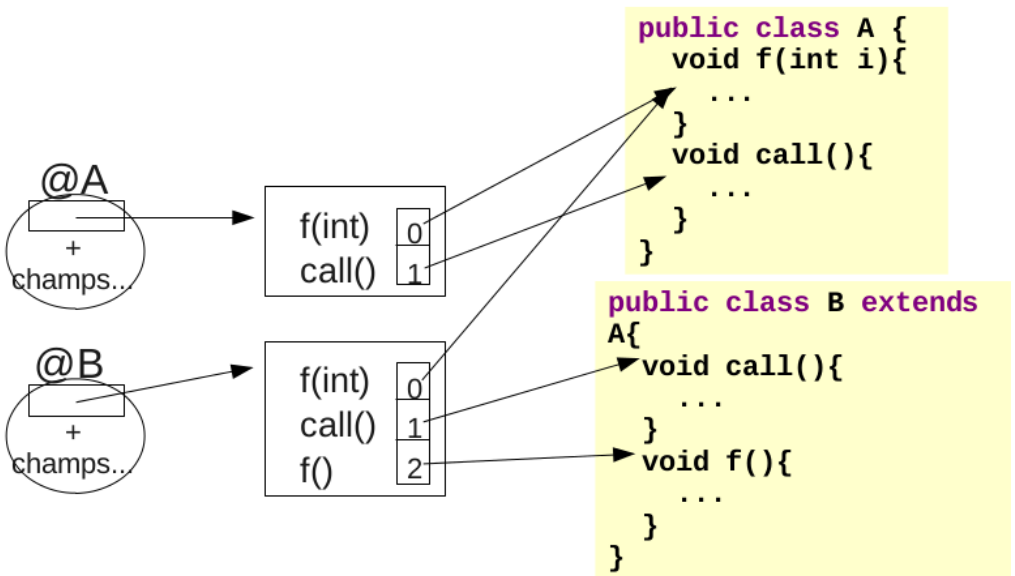
Redéfinition - Comment ça marche ?

- Chaque objet possède une reference vers la **vTable** de sa classe.
- Pour chaque méthode : un index dans la table, en commençant par les super-classes.



Polymorphisme - Comment ça marche ?

- Deux méthodes redéfinies ont le **même index**.
- L'appel polymorphe est : `object.vtable[index](args)`.



Rédéfinition vs Surcharge

- Quelle différence ?

Rédéfinition vs Surcharge

- Quelle différence ?
- **Surcharge** : **des** méthodes de **même noms** mais de **profils différents** dans une même classe.
  - Choisi à la **compilation** selon les arguments.
- **Redéfinition** : **deux** méthodes de **même noms** et de **même profil** dans 2 classes dont l'une hérite de l'autre.
  - Choisi à l'exécution selon le type réel.

Rédéfinition vs Surcharge

```
1 public class A {
2   public void m(CharSequence a) {...} // surcharge
3   public void m(List<Character> a) {...} // surcharge
4 }
5 public class B extends A {
6   public void m(Object a) {...} // surcharge
7   public void m(CharSequence a) {...} // redéfinition
8 }
```

Exemple

- On désire dessiner un tableau avec des rectangles et des ellipses sur une surface graphique.

```
1 public class Ellipse {
2   private final int x,y,w,h ;
3 }
```

```
1 public class Rectangle {
2   private final int x,y,w,h ;
3 }
```

```
1 public class DrawingArea {
2   public void drawRect(x,y,w,h){...}
3   public void drawEllipse(x,y,w,h) {...}
4 }
```

Exemple - 1ère tentative

```
1  public static void drawAll(DrawingArea area, Object[] array) {
2    for(Object o : array) {
3      if (o instanceof Rectangle) {
4        Rectangle r = (Rectangle)o ;
5        area.drawRect(r.x, r.y, r.width, r.height) ;
6      } else
7      if (o instanceof Ellipse) {
8        Ellipse e = (Ellipse)o ;
9        area.drawEllipse(e.x, e.y, e.width, e.height) ;
10     } else {
11       throw new AssertionError() ;
12     }
13   }
14 }
```

Exemple - 1ère tentative

```
1  public static void drawAll(DrawingArea area, Object[] array) {
2    for(Object o : array) {
3      if (o instanceof Rectangle) {
4        Rectangle r = (Rectangle)o ;
5        area.drawRect(r.x, r.y, r.width, r.height) ;
6      } else
7      if (o instanceof Ellipse) {
8        Ellipse e = (Ellipse)o ;
9        area.drawEllipse(e.x, e.y, e.width, e.height) ;
10     } else {
11       throw new AssertionError() ;
12     }
13   }
14 }
```

- Si on rajoute une forme, il faut penser à aller dans cette static...
- Ajout de forme ne peut être fait que par le développeur de drawAll()

Exemple - 2ème tentative - héritage

```java
public class Rectangle {
  private final int x,y,w,h ;
  public void draw(DrawingArea area){
    area.drawRect(x,y,w,h) ;
  }
}
```

```java
public class Ellipse extends Rectangle{
  private final int x,y,w,h ;
  @Override
  public void draw(DrawingArea area){
    area.drawEllipse(x,y,w,h) ;
  }
}
```

```java
public static void drawAll(DrawingArea area, Rectangle[] array) { //astuce sous-
    typage
  for(Rectangle r : array) {
    r.draw(area) ; //astuce polymorphisme
  }
}
```

Exemple - 2ème tentative - héritage

```java
public class Rectangle {
  private final int x,y,w,h ;
  public void draw(DrawingArea area){
    area.drawRect(x,y,w,h) ;
  }
}
```

```java
public class Ellipse extends Rectangle{
  private final int x,y,w,h ;
  @Override
  public void draw(DrawingArea area){
    area.drawEllipse(x,y,w,h) ;
  }
}
```

```java
public static void drawAll(DrawingArea area, Rectangle[] array) { //astuce sous-
    typage
  for(Rectangle r : array) {
    r.draw(area) ; //astuce polymorphisme
  }
}
```

- Mais pourquoi une ellipse est un rectangle ?

Exemple - 3ème tentative - interface

- Besoin d'un "contrat" que nos forment doivent respecter (draw).
- Interface permet sous-typage et polymorphisme, mais sans l'héritage des champs et méthodes ("héritage simplifié").

Exemple - 3ème tentative - interface

```java
public interface Shape {
  public void draw(DrawingArea area) ;
}
```

```java
public static void drawAll(DrawingArea
      area, Shape[] array) { //sous-
    typage
  for(Shape s : array) {
    s.draw(area) ; //polymorphisme
  }
}
```

```java
public class Rectangle implements Shape{
  private final int x,y,w,h ;
  @Override
  public void draw(DrawingArea area){
    area.drawRect(x,y,w,h) ;
  }
}
```

```java
public class Ellipse implements Shape{
  private final int x,y,w,h ;
  @Override
  public void draw(DrawingArea area){
    area.drawEllipse(x,y,w,h) ;
  }
}
```

Cours 2 : Polymorphisme et compagnie

Test d'égalité

- Que testent les opérateurs == et != ?

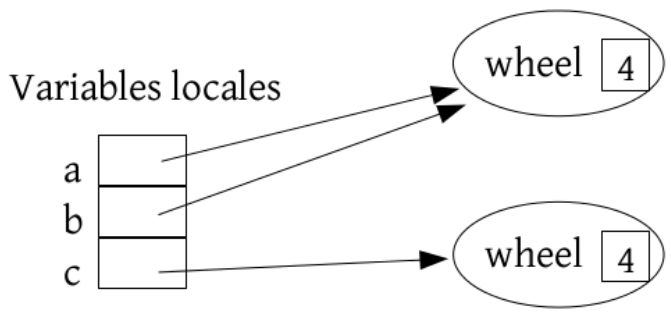
Test d'égalité

- Que testent les opérateurs == et != ?
- Ca dépend du type !
  - Types primitifs : leurs valeurs.
  - Types objets : la valeur des **références** !

Test d'égalité

- Que testent les opérateurs == et != ?
- Ca dépend du type !
  - Types primitifs : leurs valeurs.
  - Types objets : la valeur des **références** !

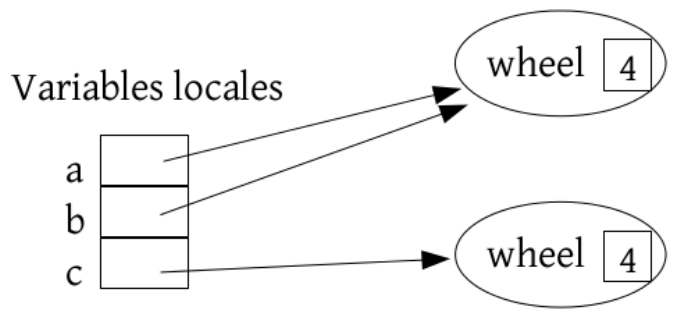
```
1 Truck a = new Truck(4) ;
2 Truck b = a ;
3 Truck c = new Truck(4) ;
4
5 syso(a==b, b==c, a==c) ;
```



Test d'égalité

- Que testent les opérateurs == et != ?
- Ca dépend du type !
  - Types primitifs : leurs valeurs.
  - Types objets : la valeur des **références** !

```
1 Truck a = new Truck(4) ;
2 Truck b = a ;
3 Truck c = new Truck(4) ;
4
5 syso(a==b, b==c, a==c) ;
```



- Vrai, Faux, Faux.

La méthode equals()

- Permet de définir une comparaison **sémantique** des objets
- La plupart des classes de l'API redéfinissent `equals`.

```
1    String s1 = new String("MIAGE") ;
2    String s2 = new String("MIAGE") ;
3    System.out.println(s1==s2) ; //false
4    System.out.println(s1.equals(s2)) ; //true
```

- L'API collections Java utilisent equals() en interne
- **ATTENTION !** La méthode `equals` d'`Object` teste les références !

```
1 public boolean a.equals(b) { return a == b ; }
```

Pourquoi redéfinir equals

- Pour permettre la comparaison **sémantique** d'objets distincts en mémoire

```
1  public class Car {
2      private final String brand ;
3      private final String model ;
4      private final int id ;
5
6      public Car(String brand, String model, int id) {
7          this.brand = brand ;
8          this.model = model ;
9          this.id = id ;
10     }
11
12     public static void main(String[] args) {
13
14         Car a = new Car("Peugeot", "306", 1234) ;
15         Car b = new Car("Peugeot", "306", 1234) ;
16
17         a == b ; // false  :(
18         a.equals(b) ; // false  :(((
19
20     }
21 }
```

Equals dans les collections

```java
public class Car {
    private final int id ;

    public Car(int id) {
        this.id = id ;
    }

    public static void main(String[] args) {
        ArrayList<Car> carDB = new ArrayList<>() ;
        carDB.add(new Car(1234)) ;

        System.out.println(carDB.contains(new Car(1234))) ; //false
    }
}
```

Comment redéfinir equals

```
1  public class Car {
2    private final int id ;
3
4    @Override
5    public boolean equals(Object o) { //Object sinon .. ?
6      if( !(o instanceof Car)) return false ;
7      Car c = (Car)o ; //cast safe
8      return id == c.id  ;
9    }
10 }
```

Comment redéfinir equals

```
1  public class Car {
2    private final int id ;
3
4    @Override
5    public boolean equals(Object o) { //Object sinon .. ?
6      if( !(o instanceof Car)) return false ;
7      Car c = (Car)o ; //cast safe
8      return id == c.id  ;
9    }
10 }
```

- Comme définie sur `Object`, la méthode redéfinie doit prendre un `Object` en argument !
- Doit renvoyer `false` si l'objet en argument n'est pas de la même classe que l'objet courant.
  - (Un des) seul cas où on peut utiliser `instanceof`.

instanceof

- Format : `objectReference instanceof type`, renvoie un boolean.

```
1 String s = "Miage" ;
2 if(s instanceof String) { ...}
3 Object o = new String("egaiM") ;
4 if(o instanceof String) {...}
5 if(o instanceof Integer) {...}
```

- Vrai/Faux ?

instanceof

- Format : `objectReference instanceof type`, renvoie un boolean.

```
1 String s = "Miage" ;
2 if(s instanceof String) { ...}
3 Object o = new String("egaiM") ;
4 if(o instanceof String) {...}
5 if(o instanceof Integer) {...}
```

- Vrai/Faux ?
  - vrai, vrai, faux.

instanceof

- Format : `objectReference instanceof type`, renvoie un boolean.

```
1 String s = "Miage" ;
2 if(s instanceof String) { ...}
3 Object o = new String("egaiM") ;
4 if(o instanceof String) {...}
5 if(o instanceof Integer) {...}
```

- Vrai/Faux ?
  - vrai, vrai, faux.
- Une sous-classe est un type d'une super-classe, donc :

```
1 String s = "Miage" ;
2 if(s instanceof Object) { ...} //vrai
```

instanceof

- Format : `objectReference instanceof type`, renvoie un boolean.

```
1 String s = "Miage" ;
2 if(s instanceof String) { ...}
3 Object o = new String("egaiM") ;
4 if(o instanceof String) {...}
5 if(o instanceof Integer) {...}
```

- Vrai/Faux ?
  - vrai, vrai, faux.
- Une sous-classe est un type d'une super-classe, donc :

```
1 String s = "Miage" ;
2 if(s instanceof Object) { ...} //vrai
```

```
1 String s = "Miage" ;
2 if(s instanceof Integer) { ...} //compile pas
```

instanceof

- Format : `objectReference instanceof type`, renvoie un boolean.

```
1 String s = "Miage" ;
2 if(s instanceof String) { ...}
3 Object o = new String("egaiM") ;
4 if(o instanceof String) {...}
5 if(o instanceof Integer) {...}
```

- Vrai/Faux ?
  - vrai, vrai, faux.
- Une sous-classe est un type d'une super-classe, donc :

```
1 String s = "Miage" ;
2 if(s instanceof Object) { ...} //vrai
```

```
1 String s = "Miage" ;
2 if(s instanceof Integer) { ...} //compile pas
```

```
1 String s = null ;
2 if(s instanceof String) { ...} //faux
```

instanceof et getClass

- `getClass()` permet d'obtenir la classe d'un objet à l'exécution.

```
1 Object o = new Integer(3) ;
2 o.getClass() == Integer.class ; //true
```

instanceof et getClass

- `getClass()` permet d'obtenir la classe d'un objet à l'exécution.

```
1 Object o = new Integer(3) ;
2 o.getClass() == Integer.class ; //true
```

- Différence entre ces if ?
- Indice : `Number` est une classe abstraite...

```
1 Object o = new Integer(3) ;
2 if(o.getClass() == Number.class) {...
3 if(o instanceof Number) { ...
```

instanceof et getClass

- `getClass()` permet d'obtenir la classe d'un objet à l'exécution.

```
1 Object o = new Integer(3) ;
2 o.getClass() == Integer.class ; //true
```

- Différence entre ces if ?
- Indice : `Number` est une classe abstraite...

```
1 Object o = new Integer(3) ;
2 if(o.getClass() == Number.class) {...
3 if(o instanceof Number) { ...
```

- `instanceof` teste si la classe de l'instance est la classe demandée ou une sous-classe.
- `.getClass() ==` est une égalité entre classes.
- `.getclass() == Toto` toujours faux si Toto est une interface ou une classe abstraite !

Comment redéfinir equals – astuces

```java
public class Car {
  private final String brand ;
  private final int nbDoors ;

  @Override
  public boolean equals(Object o) {
    if(this==o) return true ; //raccourci
    if( !(o instanceof Car)) return false ;
    Car c = (Car)o ;
    return nbDoors==c.nbDoors && brand.equals(c.brand) ;
    //&& paresseux, primitifs d'abord
  }
}
```

Hashcode

- `o.hashCode()` renvoie un entier qui "résume" l'objet `o`.
- Utilisé par l'API des collections basés sur les tables de hachage (HashMap, HashSet...).
- Doit bien couvrir l'ensemble des entiers possibles, sinon perd l'intérêt de la table de hachage.

Hashcode et equals

- **Si** `o1.equals(o2) == true`, **alors** on doit avoir `o1.hashCode() == o2.hashCode()`.
- L'implication inverse n'est pas requise
- **Attention !** La méthode `hashCode` d'`Object` retourne un hash code différent pour deux objets distincts en mémoire

Donc ...

Hashcode et equals

- **Si** `o1.equals(o2) == true`, **alors** on doit avoir `o1.hashCode() == o2.hashCode()`.
- L'implication inverse n'est pas requise
- **Attention !** La méthode `hashCode` d'`Object` retourne un hash code différent pour deux objets distincts en mémoire

Donc ...

- Si `equals` est redéfini, `hashCode` doit l'être aussi !
- Exemple correct :
  - Book.equals() comparaison mot à mot
  - Book.hashCode() première lettre du titre.

Equals sans Hashcode

```
1  public class Phone {
2    private final int areaCode ;
3    private final int number ;
4
5    public Phone(int ac, int n) {
6      areaCode = ac ;
7      number=n ;
8    }
9
10   @Override public boolean equals(Object o) {
11     if(o==this)return true ;
12     if( !(o instanceof Phone)) return false ;
13     Phone pn = (Phone)o ;
14     return areaCode==pn.areaCode && number==pn.number ;
15   }
16
17   public static void main(String[] args) {
18     Map<Phone, String> m = new HashMap<Phone, String>() ;
19     m.put(new Phone(0033, 123456), "Cornaz") ;
20
21     System.out.println(m.get(new Phone(0033, 123456))) ;
22   }
23 }
```

Equals sans Hashcode

- Affiche quoi ?

Equals sans Hashcode

- Affiche quoi ?
- `null`...
- Deux instances de Phone sont utilisées (ajout puis recherche) : `hashCode` d'objet est différent : pas dans le même sac.

Hashcode

- Le but est d'éviter au maximum les collisions et de bien répartir (pour les objets utilisés dans le programme).
- Doit être rapide a calculer ! (possibilité de le stocker dans le cas d'un objet non mutable !)
- On peut utiliser un XOR entre des valeurs de hash déjà bien calculées.
- Éventuellement un `Integer.rotateLeft` pour décaler des bits de façon circulaire.
- Utiliser `Objects.hash`

Hashcode

```
1  public class Car {
2    private final String brand ;
3    private final int nbDoors ;
4
5    /* très mauvasis hashCode
6    @Override
7    public int hashCode() {
8      return 42 ;
9    }
10   */
11
12   /* mauvais hashcode
13   @Override
14   public int hashCode() {
15     return nbDoors+brand.hashCode() ;
16   }
17   */
18
19   @Override
20   public int hashCode() {
21     return nbDoors^Integer.rotateLeft(brand.hashCode(), 16) ;
22   }
23 }
```

Switch de strings

- En java, `switch` sur les `String` possible.
- Le compilateur calcule le hashCode pour chaque chaîne.
- Puis equals car possible d'avoir un même hashCode pour 2 chaines différentes.

Switch de strings

```
1  String s = ...
2  switch(s){
3    case "Chimon" :
4      ...
5    case "Roland" :
6      ...
7  }
```

```
1        switch(s.hashCode()) {
2        case "Chimon".hashCode() :
3          if(s.equals("Chimon")) {
4
5          }
6        case "Roland".hashCode() :
7          if(s.equals("Roland")) {
8
9          }
10       }
```