

# Concept avancé de programmation

# Concepts

- Objet & Encapsulation
- Mutable/Immutable
- Typage & Abstraction
- Héritage, interface & traits
- Sous-typage
- Polymorphisme
- Closure/lambda

# Objet & Encapsulation

Un objet définie en dedans et un dehors

- L'idée est de protéger le dedans du dehors
- Seul l'objet peut modifier son état (ses champs)
  - On appel ce principe l'encapsulation

Exemple de code qui casse l'encapsulation

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

```
public class Main {  
    public static void main(String[] args) {  
        Point point = new Point();  
        point.x = 3;    // Mal !  
    }  
}
```

# Objet & Encapsulation

On ne peut changer l'état d'un objet que dans une méthode de celui-ci

Même code en mieux

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

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

En Java, on met tous les champs **private**

# Encapsulation

Principe fondateur de la programmation orienté objet

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

# Encapsulation & Abstraction

Le fait de faire faire les modifications par les méthodes permet de changer l'implantations sans changer *l'interface*

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

Le changement est compatible,  
Main.java n'a pas besoin d'être changé !

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

# Objet et Etat

Un objet doit toujours être dans un état valide

- Une méthode transitionne d'un état valide à un autre état valide
- Le(s) constructeur(s) initialise(nt) l'objet dans un état valide

```
public class Point {  
    private double rho;    // always positive  
    private double theta;  
  
    public Point(double x, double y) {  
        rho = Math.hypot(x, y);    // hypot ~<=> sqrt(x * x + y * y)  
        theta = Math.atan2(x, y);  
    }  
}
```

# Concepts

- Objet & Encapsulation
- Mutable/Immutable
- Typage & Abstraction
- Héritage, interface & traits
- Sous-typage
- Polymorphisme
- Closure/lambda

# Mutabilité/Immutabilité

Qu'affiche le code suivant ?

```
public class AnotherClass {  
    public void anotherMethod() {  
        Point p=new Point(2,3);  
        System.out.println(p);  
        System.out.println(p.getX());  
    }  
}
```

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

# Mutabilité/Immutabilité

Vous êtes sûr ?

```
public class AnotherClass {  
    public void anotherMethod() {  
        Point p=new Point(2,3);  
        System.out.println(p);  
        System.out.println(p.getX());  
    }  
}
```

```
public class Point {  
    private double x;  
    private double y;  
    public Point(double x,double y) {  
        this.x=x;  
        this.y=y;  
    }  
    public int getX() {  
        return x;  
    }  
    public String toString() {  
        x=4;  
        return x+", "+y;  
    }  
}
```

# Mutabilité/Immutabilité

Quel est le problème ?

- Lors de l'utilisation, on considère l'objet comme ne pouvant pas changer de valeur après création  
(on dit que l'objet est **immutable**)
- Alors que le code de `toString()` modifie le champs `x`

Choisir si est un objet est mutable où pas est une décision de design importante

# Problème des objets mutables

L'utilisation d'objets mutables peut casser l'encapsulation

```
public class Cat {
    private final StringBuilder name;
    public Cat(StringBuilder name) {
        this.name=name;
    }
    public StringBuilder getName() {
        return name;
    }
    public static void main(String[] args) {
        StringBuilder name=new StringBuilder("sylvestre");
        Cat cat=new Cat(name);
        name.reverse();
        System.out.println(cat.getName()); // ertsevlys
    }
}
```

# Utiliser un objet non-mutable

Le plus simple est d'utiliser un objet non-mutable

```
public class Cat {
    private final String name;
    public Cat(String name) {
        this.name=name;
    }
    public String getName() {
        return name;
    }
    public static void main(String[] args) {
        String name="sylvestre";
        Cat cat=new Cat(name);
        name.reverse(); // compile pas !
        System.out.println(cat.getName()); // sylvestre
    }
}
```

# Immutabilité & final

Java permet de garantir l'immutabilité d'un champ en le déclarant **final**

```
public class Point {  
    public Point(double x, double y) {  
        this.x=x;  
        this.y=y;  
    }  
    public String toString() {  
        x=4; // erreur  
        return x+", "+y;;  
    }  
    private final double x;  
    private final double y;  
}
```

On met les champs final **par défaut**

# Immutabilité & final

final est pas suffisant !

```
public class Circle {  
    private final Point center;  
    ...  
    public Circle(Point center, int radius) {  
        this.center = center;  
        ...  
    }  
    public void translate(int dx, int dy) {  
        center.translate(dx, dy);  
    }  
}
```

Il faut que les objets référencés ne soit pas eux-même des objets mutables

# Immutabilité & création

Si un objet est non mutable, toutes modifications entraînent la création d'un nouvel objet

```
public class Point { //immutable
    private final int x;
    private final int y;
    public Point(int x,int y) {
        this.x=x;
        this.y=y;
    }
    public Point translate(int dx,int dy) {
        return new Point(x+dx,y+dy);
    }
}
```

```
public class Point { // mutable
    private int x;
    private int y;
    public Point(int x,int y) {
        this.x=x;
        this.y=y;
    }
    public void translate(int dx,int dy) {
        x+=dx; y+=dy;
    }
}
```

# Immutabilité & création

Il n'existe pas de façon de dire au compilateur qu'il faut absolument récupérer la valeur de retour

```
public class Point { //immutable
    private final int x;
    private final int y;
    public Point(int x,int y) {
        this.x=x;
        this.y=y;
    }
    public Point translate(int dx,int dy) {
        return new Point(x+dx,y+dy);
    }
}
```

```
public void anotherMethod {
    Point p=new Point(2,3);
    p.translate(1,1); // oups
    System.out.println(p); // 2,3
}
```

# Objets mutables/non mutables

Un objet est mutable s'il est possible de modifier son état après sa création

sinon il est non mutable :

- Champs **final**
- A des méthodes de consultation (getter), pas de méthode de modification (setter)
- Certain objet peuvent être les deux, levant une exception ou non si l'on essaye de les modifier (cf les collections en Java)

# Alors mutable ou pas ?

## En pratique

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

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

=> Copie défensive

# La copie défensive

Faire une copie lors de la création n'est pas suffisant !!!

```
public class Cat {
    private final StringBuilder name;
    public Cat(StringBuilder name) {
        this.name = new StringBuilder(name); // defensive copy
    }
    public StringBuilder getName() {
        return name;
    }
    public static void main(String[] args) {
        StringBuilder name = new StringBuilder("sylvestre");
        Cat cat = new Cat(name);
        cat.getName().reverse();
        System.out.println(cat.getName()); // ertsevlys
    }
}
```

# La copie défensive (suite)

La copie défensive doit être fait aussi lors de l'envoi de paramètre

```
public class Cat {
    private final StringBuilder name;
    public Cat(StringBuilder name) {
        this.name = new StringBuilder(name); // defensive copy
    }
    public StringBuilder getName() {
        return new StringBuilder(name);
    }
    public static void main(String[] args) {
        StringBuilder name = new StringBuilder("sylvestre");
        Cat cat = new Cat(name);
        name.reverse();
        cat.getName().reverse();
        System.out.println(cat.getName()); // sylvestre
    }
}
```

# Tableau toujours mutable

En Java, les tableaux sont toujours mutables !

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

# Protection si un objet non mutable utilise un objet mutable

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

# Concepts

- Objet & Encapsulation
- Mutable/Immutable
- Typage & Abstraction
- Héritage, interface & traits
- Sous-typage
- Polymorphisme
- Closure/lambda

# Typage et Objet

- Les langages objet peuvent être typés à la compilation et/ou à l'exécution
- 3 sortes de langages
  - Typé à la compilation pas à l'exécution
    - C++ (sans RTTI), OCaml
  - Typé à la compilation et à l'exécution
    - Java, C#
  - Typé à l'exécution
    - PHP, Javascript, Python, Ruby

# Type et Objet

```
URI uri = new URI("http://www.playboy.com");
```

Type pour le compilateur

Classe pour la VM

Le type d'un objet correspond à son *interface* c-a-d l'ensemble des méthodes que l'on peut appeler sur l'objet

La classe d'un objet correspond à l'ensemble des propriétés + méthodes utilisée pour créer un objet

En Java, la classe est aussi un objet !

# Types en Java

- Il y a deux sortes de type en Java, les types réifiés et les non-réifiés (n'existe pas à l'exécution)
- Types réifiés:
  - les classes (class), les énumérations (enum), les classes abstraites (abstract class), les annotations (@interface), les interfaces et les wildcards non bornés (List<?>)
- Types non-réifiés
  - Les types paramétrés (List<String>), les variables de type (E, T), les wildcards bornés (List<? extends Foo>), les *raw types* (List)

# Opérations dynamiques

Opérations qui utilise la classe de l'objet

- création: `new Foo`
- création d'un tableau: `new Foo[3]`
- `objet.getClass()`
- Test dynamique:
  - `object instanceof Foo`
  - `cast` (vers autre chose qu'un super-type)
- stockage dans un tableau: `array[i] = x`
- `catch(MonException e)`

Ces opérations **ne marchent pas** (ou ne compile pas) sur les types **non réifiés** !

# Concepts

- Objet & Encapsulation
- Mutable/Immutable
- Typage & Abstraction
- Héritage, interface & traits
- Sous-typage
- Polymorphisme
- Closure/lambda

# Héritage, interface & traits

- L'idée première de l'héritage est de réutiliser le code déjà existant en l'étendant au besoin
  - Bonne idée en 1960, moins bonne en 2000
    - Si on veut ré-utiliser un algorithme, pourquoi ne pas le mettre dans une fonction, une methode static dans une classe public tout simplement
    - L'héritage implique plus que la récupération du code
      - Le sous-typage
      - On récupère aussi toutes les membres
        - Toutes les méthodes et pas uniquement celles que l'on veut
        - Cela empêche de changer l'implantation de la sous-classe car on ne peut toucher à l'implantation de la super-classe.

# Il faut redéfinir **toutes** les méthodes

Code idiot que l'on voit trop souvent

```
public class StudentList extends ArrayList<Student> {  
    public void add(Student student) {  
        Objects.requireNonNull(student);  
        // assure que la liste ne contiendra jamais de null  
        super.add(student);  
    }  
}
```

Le code est idiot, car on peut écrire

```
public static void main(String[] args) {  
    List<Student> list = new ArrayList<>();  
    list.add(null);  
    StudentList studentList = new StudentList();  
  
    studentList.addAll(list); // zut on a oublié  
                             // de redefinir addAll  
    studentList.listIterator().add(list); // et aussi l'iterateur  
}
```

# Préférez la délégation

Problème résolu !

```
public class StudentList {
    private final ArrayList<Student> list;

    public StudentList() {
        list = new ArrayList<>();
    }

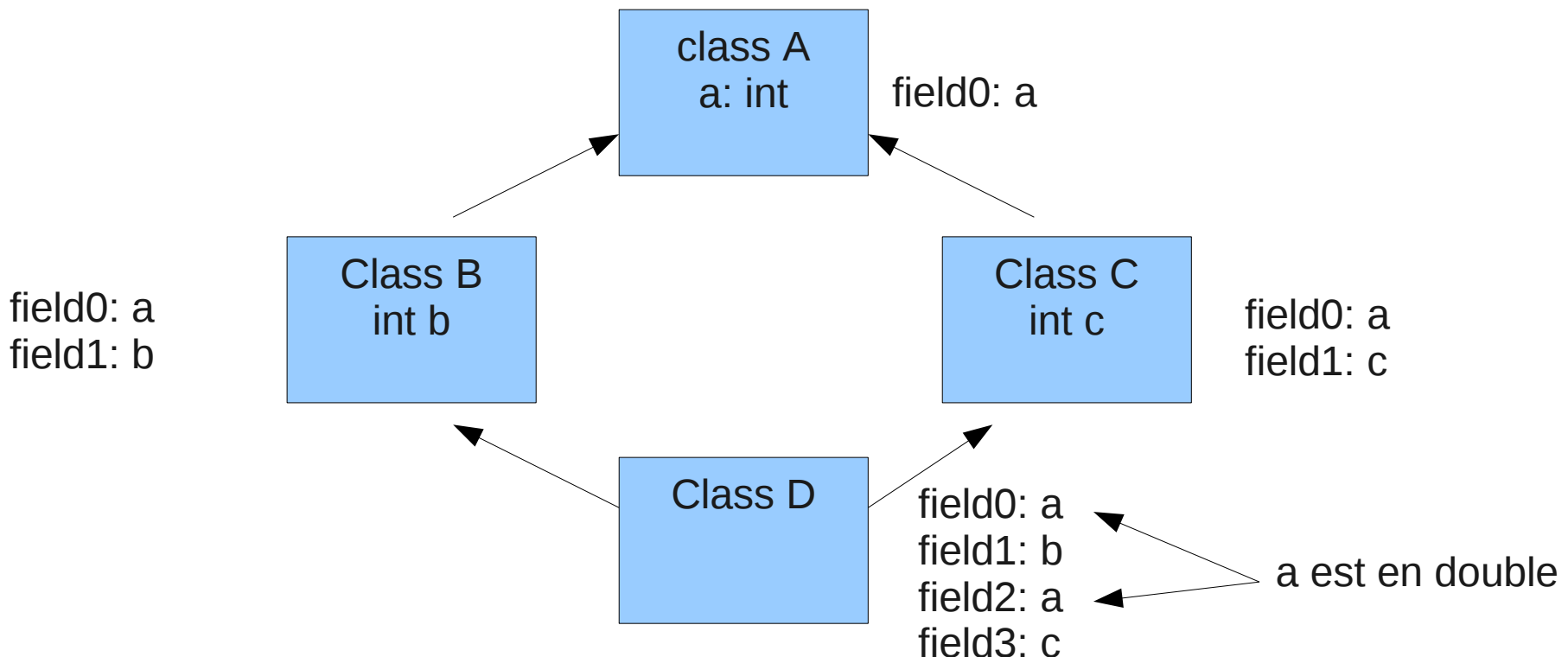
    public void add(Student student) {
        Objects.requireNonNull(student);
        list.add(student);           // delegation
    }
}
```

La délégation n'a pas le couplage fort qu'à l'héritage entre la super-classe et la sous-classe

# Héritage multiple

C++ permet l'héritage multiple,  
Java ou C# non !

Le problème du diamand :



# Interface

- Et si on veut voir le même objet et comme un truc et comme un autre
  - Solution le sous-typage multiple
- Une interface définit un type sans définir son implémentation
- Une classe peut fournir l'implémentation de plusieurs interfaces

# Traits

- On peut faire de l'héritage multiple sur le code des méthodes si il n'y a pas de champs !
- Un trait est une interface qui peut avoir des méthode contenant du code mais ne contient pas de champs
- Si la classe implante deux traits qui on la même méthode, on doit indiquer laquelle choisir à la main
- Il n'y a pas de trait en Java :(

# Concepts

- Objet & Encapsulation
- Mutable/Immutable
- Typage & Abstraction
- Héritage, interface & traits
- Sous-typage
- Polymorphisme
- Closure/lambda

# Sous-typage

Le sous-typage correspond au fait de pouvoir substituer un type par un autre

Cela permet la réutilisation d'algorithmes écrits pour un type et utilisés avec un autre

# Principe de liskov

Barbara Liskov(88) :

*If for each object  $o1$  of type  $S$  there is an object  $o2$  of type  $T$  such that for all programs  $P$  defined in terms of  $T$ , the behavior of  $P$  is unchanged when  $o1$  is substituted for  $o2$  then  $S$  is a subtype of  $T$ .*

Le sous-typage est défini de façon générale sans notion de classe

# Sous-typage en Java

Le sous-typage existe pour les différents types de Java :

- Pour l'héritage de classes ou d'interface
- Pour une classe implémentant l'interface
- Pour les tableaux d'objets
- Pour les types paramétrés

Le sous-typage ne marche qu'avec des objets

# Sous-typage et héritage

- Si un classe (resp. interface) hérite d'une autre, la sous-classe (resp.interface) est un sous-type de la classe (resp.interface) de base

```
class A {  
}  
class B extends A {  
}  
...  
public static void main(String[] args) {  
    A a = new B();  
}
```

- Comme B hérite de A, B récupère l'ensemble des membres de A

# Sous-typage et interface

- Si une classe implante une interface, alors la classe est un sous-type de l'interface

```
interface I {  
    void m();  
}  
class A implements I {  
    public void m() {  
        System.out.println("hello");  
    }  
}  
...  
public static void main(String[] args) {  
    I i = new A();  
}
```

- Comme A implante I, A possède un code pour toutes les méthodes de I

# Sous-typage et tableau

- En Java, les tableaux possèdent un sous-typage généralisés :
  - Un tableau est un sous-type de Object, Serializable et Clonable
  - Un tableau de U[] est un sous-type de T[] si U est un sous-type de T et T,U ne sont pas primitifs

```
public static void main(String[] args) {  
    Object[] o = args; // ok  
    double[] array = new int[3]; // illégal  
}
```

# Tableau & ArrayStoreException

- Comme le sous-typage sur les tableaux existe, cela pose un problème :

```
public static void main(String[] args) {  
    Object[] array = args;  
    array[0] = new Object(); // ArrayStoreException à l'exécution  
}
```

- Il est possible de considérer un tableau de String comme un tableau d'objet mais il n'est possible d'ajouter un Object à ce tableau

# Sous-typage et Type paramétré

- Les types paramétrés ne sont pas réifiés
  - => pas de check à l'exécution possible
  - donc `List<String>` n'est pas un sous-type de `List<Object>`
- Solutions théorique pour résoudre le problème:
  - Indiquer la covariance à la déclaration (Scala, C#)
  - Indiquer la covariance lors de l'utilisation (Java)

```
public void printAll(List<? extends Object> list) { ... }  
...  
List<String> list = ...  
printAll(list);
```

# Concepts

- Objet & Encapsulation
- Mutable/Immutable
- Typage & Abstraction
- Héritage, interface & traits
- Sous-typage
- Polymorphisme
- Closure/lambda

# Polymorphisme

- Le polymorphisme consiste à considérer les fonctionnalités suivant la **classe** d'un objet et non suivant le **type** de la variable dans laquelle il est stocké
- Le sous-typage permet de stocker un objet comme une variable d'un super-type, le polymorphisme fait en sorte que les méthodes soient appelées en fonction de la **classe** de l'objet

# A quoi ça sert ?

Le polymorphisme fait en sorte que certaines parties de l'algorithme soit spécialisée en fonction du type réel de l'objet

```
public class Polymorphic {
    private static void print(Object[] array) {
        for(Object o: array)
            System.out.println(o.toString());
            // appel dynamiquement Integer.toString(), String.toString(),
            // Double.toString(), Boolean.toString()
    }
    public static void main(String[] args) {
        Object[] array = new Object[] { 2, arg[0], 3.4, false };
        print(array);
    }
}
```

En Java, seul l'appel de méthode est polymorphe

# Appel virtuel & compilation

Le mécanisme d'appel polymorphe (appel virtuel) est décomposé en deux phases :

- 1) Lors de la **compilation**, le compilateur choisit la méthode la **plus spécifique** en fonction du **type déclaré** des arguments
- 2) Lors de **l'exécution**, la VM choisie la méthode en fonction de la **classe** du receveur (l'objet sur lequel on applique '.')

# Condition d'appel virtuel

Il n'y a pas d'appel virtuel si la méthode est :

- **statique** (pas de receveur)
- **private** (pas de redéfinition possible, car pas visible)
- **final** ou la classe est **final** (pas le droit de redéfinir)
- Si l'appel se fait par **super**

dans les autres cas, l'appel est virtuel

# Même un appel avec this est polymorphe

```
public class FixedSizeList {
    boolean isEmpty() {
        return size() == 0;
        // est équivalent à: return this.size()==0;
    }
    int size() {
        return 10;
    }
}

public class EmptyList extends FixedSizeList {
    int size() {
        return 0;
    }
}

public class void main(String[] args) {
    FixedSizeList list = new EmptyList();
    System.out.println(list.isEmpty()); // true
}
```

# Site d'appel et implantations

- On distingue la méthode, les implantations de cette méthode et le site d'appel à la méthode

méthode

```
public class A {  
    void call() {  
        // implantation 1  
    }  
}
```

Implantations

```
public class B extends A  
{  
    @Override void call() {  
        // implantation 2  
    }  
}
```

```
private static void callAll(A[] array) {  
    for(A a:array)  
        a.call();  
}
```

Site d'appel

# Condition de la redéfinition

Il y a redéfinition de méthode s'il est possible pour un site d'appel donné d'appeler la méthode redéfinie en lieu et place de la méthode choisie à la compilation

Le fait qu'une méthode redéfinisse une autre dépend :

- Du nom de la méthode
- Des modificateurs de visibilité des méthodes
- De la signature des méthodes
- Des exceptions levées (throws) par la méthode

# Visibilité et redéfinition

Il faut que la méthode redéfinie ait une visibilité au moins aussi grande (private < protected < public)

```
private static void callAll(A[] array)
{
    for(A a:array)
        a.call();
}
```

compilation

```
public class A {
    protected void call() {
        // implantation 1
    }
}
```

exécution

```
public class B extends A {
    @Override public void call() {
        // implantation 2
    }
}
```

# Covariance du type de retour

Le type de retour de la méthode redéfinie peut-être un sous-type de la méthode à redéfinir

```
private static void callAll(A[] array)
{
    Object o;
    for(A a:array)
        o=a.call();
}
```

```
public class A {
    Object call() {
        // implantation 1
    }
}
```

```
public class B extends A {
    @Override String call() {
        // implantation 2
    }
}
```

Ne marche qu'à partir de la 1.5

# Contravariance des paramètres

- Les types des paramètres peuvent être des super-types du type de la méthode à redéfinir

```
private static void callAll(A[] array)
{
    String s = ...
    for(A a:array)
        a.call(s);
}
```

```
public class A {
    void call(String s) {
        // implantation 1
    }
}
```

```
public class B extends A {
    void call(Object o) {
        // implantation 2
    }
}
```

Pas implanté en Java, dommage !

# Covariance des exceptions

Les exceptions levés peuvent être des sous-types de celles déclarées

```
private static void callAll(A[] array)
{
    for(A a:array) {
        try {
            a.call();
        } catch(Exception e) {
            ...
        }
    }
}
```

```
public class A {
    void call() throws Exception {
        // implantation 1
    }
}
```

```
public class B extends A {
    @Override void call()
        throws IOException {
        // implantation 2
    }
}
```

Les exceptions non *checked* ne compte pas

# Covariance des exceptions (2)

La méthode redéfinie peut ne pas levée d'exception

```
private static void callAll(A[] array)
{
    for(A a:array) {
        try {
            a.call();
        } catch(Exception e) {
            ...
        }
    }
}
```

```
public class A {
    void call() throws Exception {
        // implantation 1
    }
}
```

```
public class B extends A {
    @Override void call() {
        // implantation 2
    }
}
```

L'inverse ne marche pas !!!

# Appel de méthode

L'algorithme d'appel de méthode s'effectue en deux temps

A. on recherche les méthodes applicables (celles que l'on peut appeler)

B. parmi les méthodes applicables, on recherche s'il existe une méthode plus spécifique (dont les paramètres seraient sous-types des paramètres des autres méthodes)

Cet algorithme est effectué par le compilateur

# A. méthodes applicables

Ordre dans la recherche des méthodes applicables :

- 1) Recherche des méthodes à nombre fixe d'argument en fonction du sous-typage & conversions primitifs
- 2) Recherche des méthodes à nombre fixe d'argument en permettant l'auto-[un]boxing
- 3) Recherche des méthodes en prenant en compte les varargs

Dès qu'une des recherches trouve une ou plusieurs méthodes la recherche s'arrête

# Exemple de méthodes applicables

Le compilateur cherche les méthodes applicables

```
public class Example {  
    public void add(Object value) {  
    }  
    public void add(CharSequence value) {  
    }  
}
```

```
public static void main(String[] args) {  
    Example example=new Example();  
    for(String arg:args)  
        example.add(arg);  
}
```

ici, **add(Object)** et **add(CharSequence)** sont applicables

## B. méthode la plus spécifique

Recherche parmi les méthodes applicables, la méthode la plus spécifique

```
public class Example {  
    public void add(Object value) {  
    }  
    public void add(CharSequence value) {  
    }  
}  
  
    public static void main(String[] args) {  
        Example example=new Example();  
        for(String arg:args)  
            example.add(arg); // appel add(CharSequence)  
    }
```

La méthode la plus spécifique est la méthode dont tous les paramètres sont sous-type des paramètres des autres méthodes

# Méthode la plus spécifique (2)

Si aucune méthode n'est plus spécifique que les autres, il y a alors ambiguïté

```
public class Example {  
    public void add(Object v1, String v2) { ... }  
    public void add(String v1, Object v2) { ... }  
}
```

```
public static void main(String[] args) {  
    Example example=new Example();  
    for(String arg:args)  
        example.add(arg,arg);  
    // reference to add is ambiguous, both method  
    // add(Object,String) and method add(String,Object) match  
}
```

Les deux méthodes **add()** sont applicables, aucune n'est plus précise

# Et lors de l'exécution ?

Le polymorphisme est implémenté de la même façon quelque soit le langage Objet typé (C++, Java, C#)

```
public class A {  
    void f(int i){  
        ...  
    }  
    void call(){  
        ...  
    }  
}
```

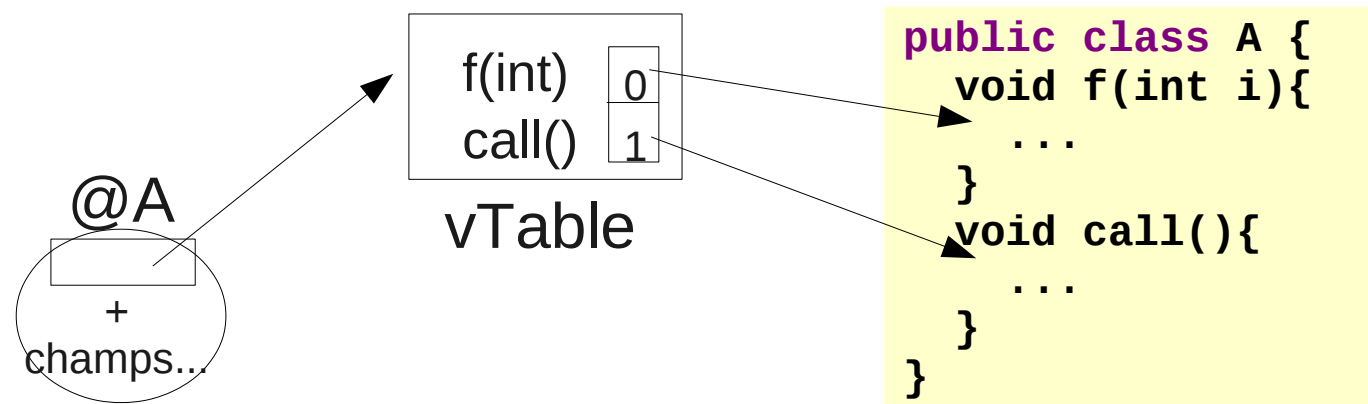
```
public class AnotherClass {  
    void callSite(){  
        A a=new B();  
        a.call();  
        a.f(7);  
    }  
}
```

```
public class B extends  
A{  
    void call(){  
        ...  
    }  
    void f(){  
        ...  
    }  
}
```

# Implantation du polymorphisme

Chaque objet possède en plus de ces champs un pointeur sur une table de pointeurs de fonction

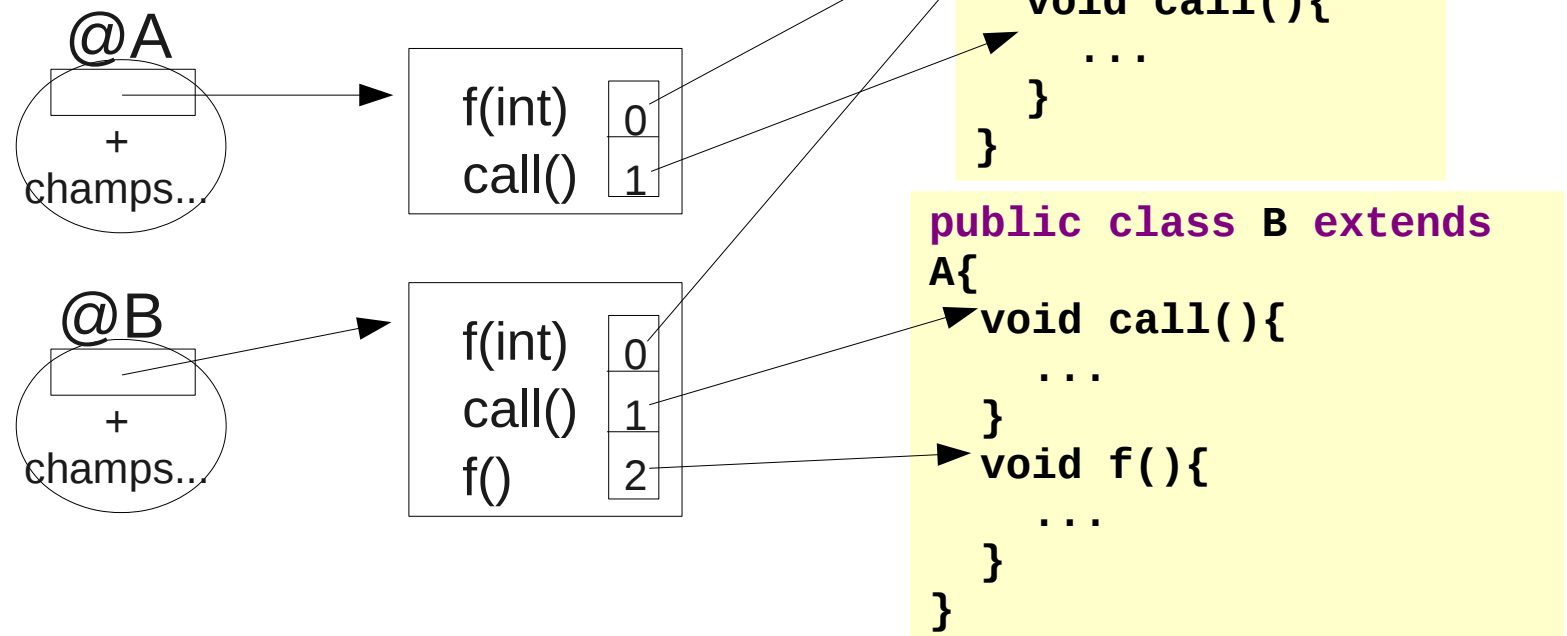
Le compilateur attribue à chaque méthode un index dans la table en commençant par numéroter les méthodes des classes de base



# vtable

Deux méthodes redéfinies ont **le même index**

L'appel polymorphe est alors :  
object.vtable[index](argument)



# vtable et interface

Le mécanisme de vtable ne marche pas bien avec l'héritage multiple car la numérotation n'est plus unique

Les implantations d'interface la classe possède une vtable par interface implantée

Sans optimisation, l'appel à travers une interface est plus lent que l'appel à travers une classe même abstraite

# Concepts

- Objet & Encapsulation
- Mutable/Immutable
- Typage & Abstraction
- Héritage, interface & traits
- Sous-typage
- Polymorphisme
- Closure/lambda

