Programmation Orientée Objet (POO) - Java

Semestre 5 - Samuel Peltier

Table des Matières

Chapitre 1 : POO et Java

Plusieurs paradigmes de programmation (types d'approche de problème) :

- **Programmation impérative (Ada, C) :** Un programme peut être représenté par une machine d'états. Décomposition d'un problème en sous-problèmes (sous-programmes).
- **Programmation fonctionnelle (LISP, OCaml) :** Un programme est un enchaînement de fonctions mathématiques. Pas d'opérateur d'affectation , pas de boucle.
- **Programmation logique (Prolog) :** Programmation déclarative. Un programme est un ensemble d'axiomes et de règles, à partir desquels on peut lancer des requêtes.
- **Programmation Orientée objet (C++,Java) :** La notion d'objet (brique de base de la programmation orientée objet) regroupe à la fois des données et les opérations qui les manipulent.

Objectif d'un langage OO:

- Faciliter la réutilisation du code (**encapsulation** et **abstraction**)
- Faciliter l'évolution du code (**héritage**)
- Améliorer la conception

Byte code (code objet) java :

- résultat de la compilation du code source
- interprété par tout JVM

Machine virtuelle Java (JVM) :

- programme interprétant le byte code Java
- interprète du byte code quelle que soit sa provenance

Les outils de développement :

- éditeur de texte + javac + java
- Netbeans / Eclipse …

```java
// boucle for similaire au C
for(int i = 0 ; i < LEN_TABLE ; i++)
{
    tab[i] = 0;
}

// boucle foreach
for(int val : tab)
{
    System.out.println(val);
}
```

Dans une boucle (for, while…), possibilité d'utiliser :

- **break** pour en sortir, ou
- **continue** pour skip la fin de la boucle actuelle et passer à la suivante.

Pour convertir une variable d'un type à l'autre, on peut juste faire : **(type voulu)variable**.
Ex :
double d;
int i = (int)d;

Tableaux, boucles, déclarations de variables, types… =C

Convention de nommage à respecter :

- `JeSuisUneClasse`
- `jeSuisUneVariable`
- `jeSuisUneMethode(...)`
- `JE_SUIS_UNE_CONSTANTE`

Chapitre 2 : Classes et Objets

Rappels

- Programmation structurée :
  - Des variables
  - Des méthodes
  - Chaque méthode résout une partie du problème
- Programmation OO

- **Encapsulation** : Le principe d'encapsulation consiste à regrouper dans une même entité des données et des opérations.
- **Abstraction** : Le principe d'abstraction consiste à décrire un objet par des données et des fonctionnalités le décrivant (dans un context donné).

On ne manipule **jamais** directement l'état d'un objet, on passe **toujours** par les méthodes.

Les objets

Un objet est défini par :

- Une identité (nom)
- Un état (attributs)
- Un comportement (fonctions)

Exemple : une voiture

- Identité : myCar, anotherCar, …
- Etat : started, color : blue, speed : 100 km/h
- Comportement :
  - selecteurs : `getColor()`, `getSpeed()`, …
  - mutateurs : `start()`, `stop()`, `accelerate()`, `turn()`, ...

```
public class Car
{
    // Constructeur
    public Car()
    {

    }

}
```

Notion de Classe

Idée intuitive : On peut voire une classe comme un “moule” qui sert à construire des objets similaires. Un objet est **une instance** d’une classe.

Constructeurs

Les constructeurs :

- permettent de **créer des objets** (à partir de données)
- portent le **même nom** que leur **classe**
- ne possèdent pas de retour

Les constructeurs ont deux sortent d’attributs :

- ceux liés à l’**état courant** d’un objet (vitesse)
- ceux qui le caractérisent à sa création (couleur) -> **attributs de propriété**

Usuellement on définit un constructeur :

- où tous les attributs ont une **valeur par défaut**
- où tous les **attributs de propriété** sont indiqués

Constructeurs remarquables :

- Constructeur **par défaut**

```
// Constructeur par défaut
public Car()
{

}
```

- Constructeur **par copie**

```
// Constructeur par copie
public Car(Car c)
{

}
```

Règles sur les constructeurs :

- Si **aucun constructeur** n’est défini dans une classe, alors cette classe possède un **constructeur par défaut** défini de manière implicite.

- **Dès qu'un constructeur est défini explicitement , le constructeur par défaut n'est plus défini implicitement (ce qui n'empêche pas de le définir explicitement).**

- Un constructeur peut faire appel à un autre constructeur :
  `this(paramètres du constructeur appelé);`
  cet appel doit être la première instruction du constructeur

```
// Constructeur avec la couleur en paramètre
public Car(Color c)
{
    this.color = c;
    this.started = false;
    this.speed = 0;
}

// Constructeur appelant un autre constructeur défini
public Car()
{
    this(Color.BLACK);
    speed = 12;
}
```

Création d'un objet :

```
Car myCar; // Déclaration
myCar = new Car(); // Instanciation
Car myCar2 = new Car(); // Déclaration + Instanciation
```

Valeur spéciale **null**

Tout objet (quelle que soit sa classe) peut être affecté à la valeur **null.**

Un objet non instancié vaut **null.**

ATTENTION :

```
Car c = null;
Television tv = null;
```

MAIS on ne peut pas comparer c et tv car les types ne sont pas compatibles

L'objet courant **this**

A quoi ça sert ?

- Désigner l'objet dans lequel on se trouve
- Rendre explicite l'accès aux attributs et méthodes de la classe
- passer en paramètre la référence de l'objet courant

Note : l'objet courant **this** est une référence qui est "cachée".
On ne peut pas affecter une nouvelle valeur à this.

Exemple : Car.java

Visibilité

La visibilité détermine si on a accès à l'élément dans la classe , dans les sous classes , dans le package, ou partout ailleurs.

Les visibilités :

- **private** : seulement dans la classe
+ **public** : partout
 # protected: dans la classe , les sous classes et le package
package private (pas de mot clé): dans la classe et le package

("+, -, #" font partie de la notation UML)

Exemple:Car.java

Pour tester sa classe, il est possible de définir une fonction *main* dans la classe. Ainsi, tous les attributs privés sont accessibles.

Si un attribut est `final` (constante) :

- soit il est défini dans la classe lors de sa déclaration

```
public final Color COLOR = Color.RED;
```

- soit il est défini dans **tous** les constructeurs

Il est donc plus pratique pour un attribut `final` de mettre son accessibilité en `public`.

Les méthodes

Une méthode est définie par:

- un nom
- un type de retour (éventuellement *void*),
- des paramètres d'entrée (éventuellement aucun)

Un paramètre peut être :

- De type primitif
- Une référence d'objet typée par n'importe quelle classe

Appel de méthode :

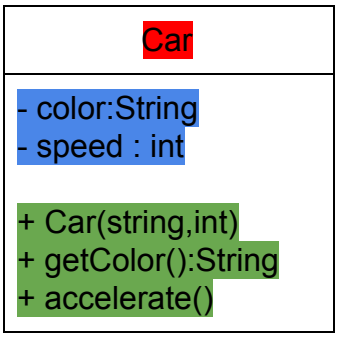
```
anInstance.aMethod(...);
```

En Java, tous les paramètres sont passés **par valeur**. Les fonctions agissent **sur des copies** des paramètres :

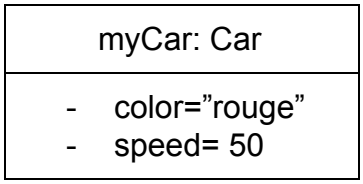
- types simples
- objets (la référence est dupliquée et non pas l'objet)

Modélisation UML

"U.M.L." = Unified Modeling Language :
Langage graphique de modélisation programmes OO.



Nom de la classe
Attributs
Constructeur méthodes

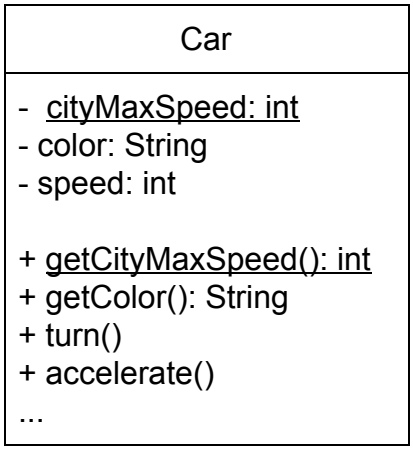


Le mot clé static

Une méthode ou un attribut ***static*** est **relatif à la classe**, et non à une instance particulière.

```
private static int cityMaxSpeed = 50;
public static int getCityMaxSpeed();
```

- Notation UML : souligné



Il n’y a pas de sens d’utiliser le mot clé this dans une méthode static car celle-ci peut être appelée avant l’instanciation d’un objet de la classe.

static -> lié à la classe
non static -> lié à l’objet (instance de la classe)

Accès à un attribut/méthode static :
`ClassName.staticElement`

Exemples :
`Car.getCitySpeed()`
`Car.cityMaxSpeed`

Notes :

- On ne peut pas utiliser l'objet courant this dans une méthode static (ça n'a pas de sens)
- On peut accéder à un attribut/méthode static via une instance de la classe (à ne pas faire, il vaut mieux passer par la classe directement "`className.staticElement`").

Le bloc d'initialisation `static` :

Si l'initialisation d'une variable static ne peut pas se calculer avec une simple affectation, on doit utiliser un bloc `static`.

```
private static Type myStaticInstance;
```

```
static
{
    // Instructions initialisées en même temps que la classe
}
```

Cycle de vie d'un objet

Appel à un constructeur par l'opérateur :

`new ClassName(parameters)`

Si un objet n'est plus référencé, alors le destructeur **`finalize()`** est appelé par la JVM juste avant de libérer de l'espace mémoire.

Le destructeur **`finalize()`** n'est jamais appelé explicitement

La gestion de la mémoire s'effectue par le garbage collector (ramasse miette) qui est lancé régulièrement automatiquement par la JVM.

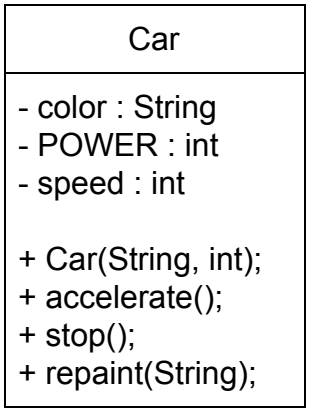
On peut suggérer à la JVM le lancer le GC par la commande `System.gc`

A retenir : En java, la JVM s'occupe de la libération de la mémoire. Lorsqu'un objet n'est plus référencé, on considérera que la mémoire est libérée.

Réalisation d'un programme OO :

1. **On conçoit (en UML)**
2. **On implante.**

Exemple : on souhaite réaliser un jeu de course de voitures en ligne droite. Chacune ayant une couleur et une puissance fixées à sa construction. Une voiture peut accélérer, s'arrêter et pourra être repeinte.




```
public class Car {
	private String color;
	private final int POWER;
	private int speed;

	public Car(String col, int pow) {
		this.color = col;
		this.POWER = pow;
		this.speed = 0;
	}
	...
}
```

Classes de base en Java

Chaque type primitif a son analogue objet, muni de méthodes utiles :

- `boolean` → `Boolean`
- `byte`, `short`, `int`, `long` → `Byte`, `Short`, `Integer`, `LongInteger`
  - Constantes de classes
    `MIN_VALUE ET MAX_VALUE`
  - Conversion à partir de String
    ```
    String s = "123";
    Integer i = Integer.parseInt(s);
    ```
- `char` → `Character`
  `isDigit()`, `isJavaLetter()`, `isLowerCase()`, `toUpperCase()`, `...`
- `float`, `double` → `Float`, `Double`
  Valeurs spécifiques:
  `NaN ATTENTION : NaN != NaN`
  `NEGATIVE_INFINITY, POSITIVE_INFINITY`
  `Le zéro est négatif, etc.`

Exemple: API JAVA (oracle)

La classe String

- c'est une classe **non mutable**, qui n'a pas d'analogue primitif
- Concaténation : `+`
  ```
  System.out.println("blabla"+12+''+(2>3));
  ```
- méthode `length()` pour récupérer la longueur
- pour tester si deux chaînes sont égales `s1.equals(s2)`
- NB.Java effectue des optimisations mémoire ,et dans certains cas , on a `s1.equals(s2)` est équivalent à `s1 == s2.`

s1 == s2 est une comparaison de référence, s1.equals(s2) compare les chaînes de caractères.

A chaque concaténation, un "new" est fait implicitement. La zone de stockage se libère et une nouvelle zone est affecté à la chaîne concaténée.

Comparaison de variables

Type primitifs :

- `a == b` retourne vrai si a = b

Pour les objets c'est pareil mais **Attention** : a et b sont des références.

Si on souhaite comparer les attributs des objets, on peut :

- Définir une méthode ad hoc

```
public boolean myEqualMethod(MyClass x)
```

- Redéfinir la méthode (on verra ça plus tard tkt bro)

```
@Override //on verra ça plus tard
public boolean equals(Object o)
```

Les tableaux d'objets

Même chose que pour les tableaux de types primitifs :

```
Type[] tab = new Type[size];
```

ATTENTION : Le tableau est instancié, mais chaque case du tableau contient une référence qui n'a pas été instanciée.

<table>
<tr><th>tab[0]</th><th>tab[1]</th><th>tab[2]</th><th>tab[3]</th></tr>
<tr><td>↓</td><td>↓</td><td>↓</td><td>↓</td></tr>
<tr><td colspan="2">→</td><td colspan="2">←</td></tr>
<tr><td colspan="4">null</td></tr>
</table>

Affichage

Affichage à l'écran :

`System.out.print()` → affichage sans retour à la ligne

`System.out.println()` → affichage avec retour à la ligne

Différentes sorties possibles :

`out` → sortie standard

`err` → en cas d'erreur

Affichage d'objets :

```
System.out.print(myObject);
affichage : package.Classe@adresse
```

Redéfinir l'affichage :

```
@Override
public String toString(){
…
}
```

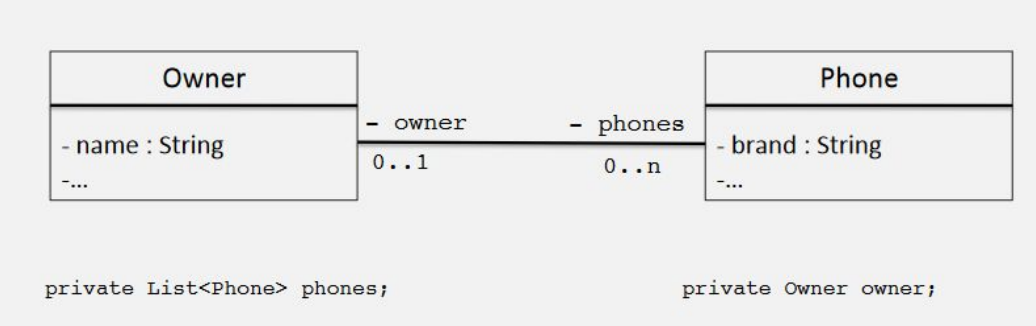
Chapitre 3 : Associations de classes

- Associations unidirectionnelles
- Associations bidirectionnelles
- Agrégations
- Compositions

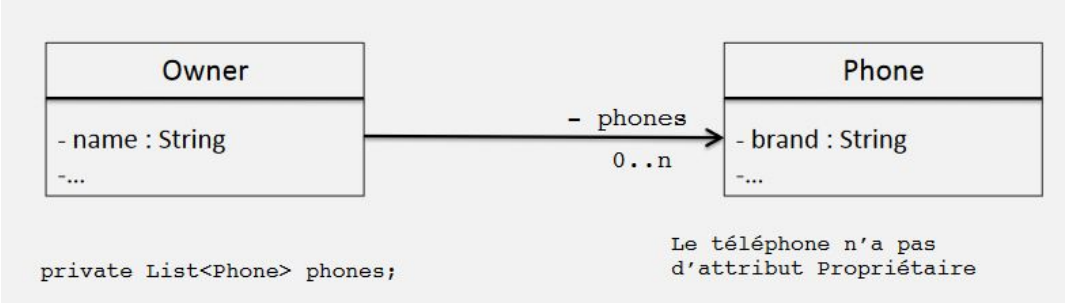
Associations de Classes

- Si l'attribut d'une classe A est un objet de type B (et/ou inversement), alors les deux classes A et B sont liées. Il faut alors définir l'association qui lie ces deux classes.

- **Association simple** :



- **Association unidirectionnelle :**



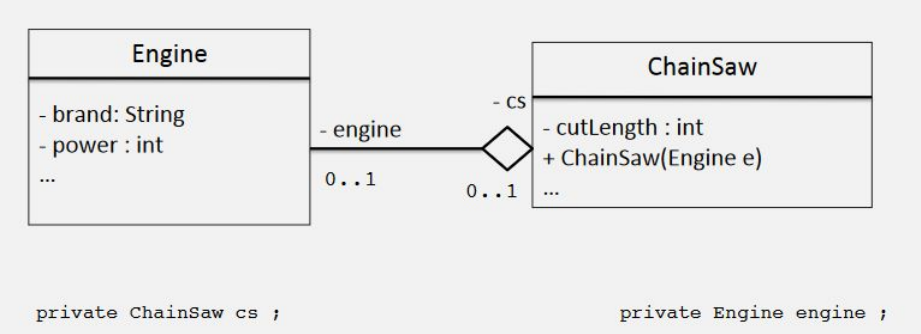
Question à se poser: de quel type est l'association entre les deux objets ?

Deux cas de figure:

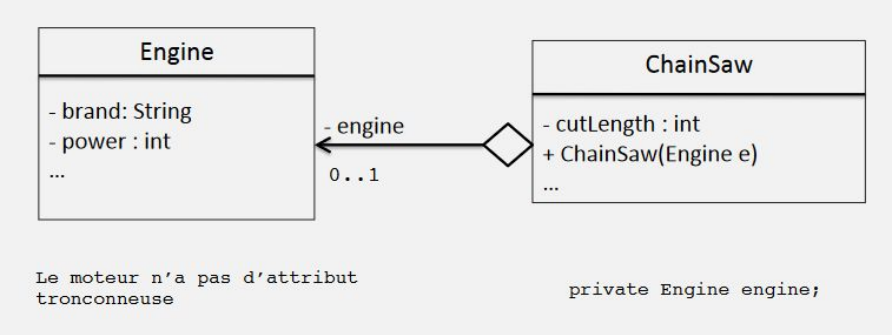
- **Agrégation**
  - Les durées de vie des deux objets sont indépendantes

- **Composition**
  - La durée de vie de l'attribut est incluse dans celle de l'objet.
  - Conséquence : l'attribut ne peut pas être partagé.

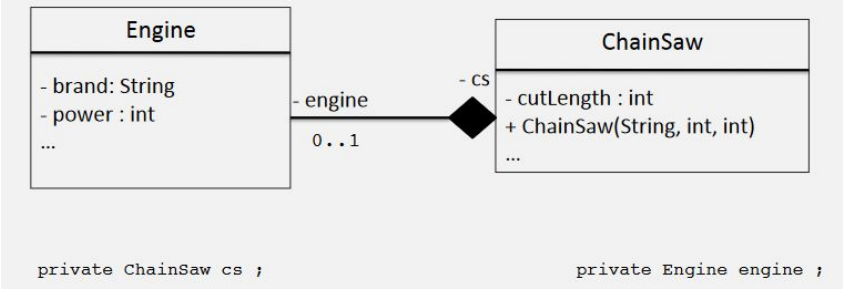
Un exemple d'agrégation :



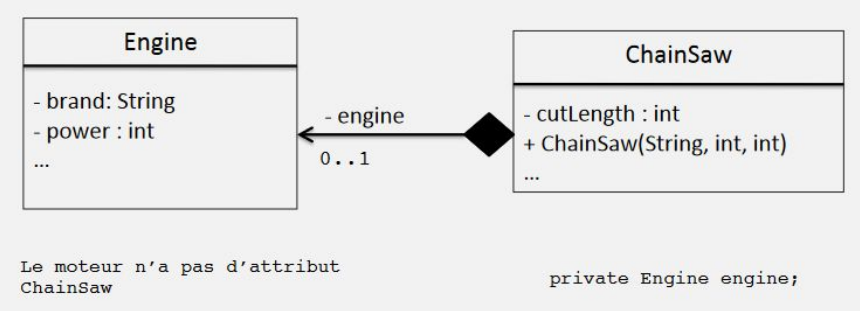
Un exemple d'agrégation unidirectionnelle :



Un exemple de composition :



Un exemple de composition unidirectionnelle :



Question : comment se traduisent l'agrégation et la composition en terme d'implémentation ?

- **Agrégation :**
  - L'attribut est passé en paramètre d'une méthode (constructeur ou autre) de l'objet composé

- **Composition :**
  - L'attribut est créé dans une méthode (constructeur ou autre) de l'objet composé
  - Aucun sélecteur sur l'objet composite.

Exemples : CarAggregation.java & CarComposition.java

Association vs Agrégation ?

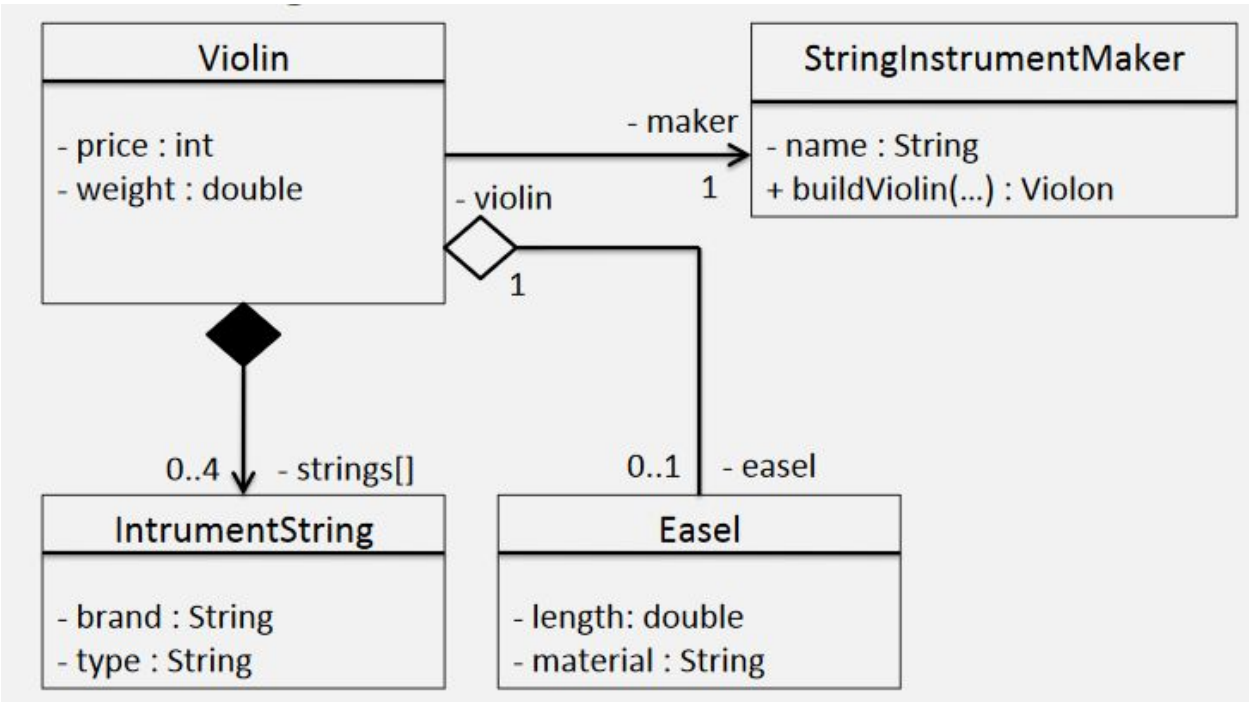
- D'un point de vue implémentation, il n'y a aucune différence entre une **association** et une **agrégation**.
- Deux écoles :
  1. L'agrégation ajoute un aspect sémantique (un objet est une partie de l'autre)
  2. Une association = on n'a pas encore décidé entre agrégation et composition

Diagramme de classe UML :

- Lors de la conception d'un logiciel orienté objet, de nombreuses classes peuvent apparaître.

- Il faut identifier les classes, les liens (association, agrégation, composition) et les cardinalités (1, 0 .. 1, 1..*, etc.) afin d'établir un **diagramme de classes UML**.

Que signifie cette modélisation ?



Chapitre 4 : Héritage et ses implications

- Principes de l'Héritage
- Redéfinition et Surcharge
- Contrôle de l’Héritage
- Polymorphisme
- Classes abstraites
- Interfaces

Jusqu’à présent, nous avons vu comment construire des classes et définir des associations

Mais un des principaux intérêts de la POO est de pouvoir étendre aisément

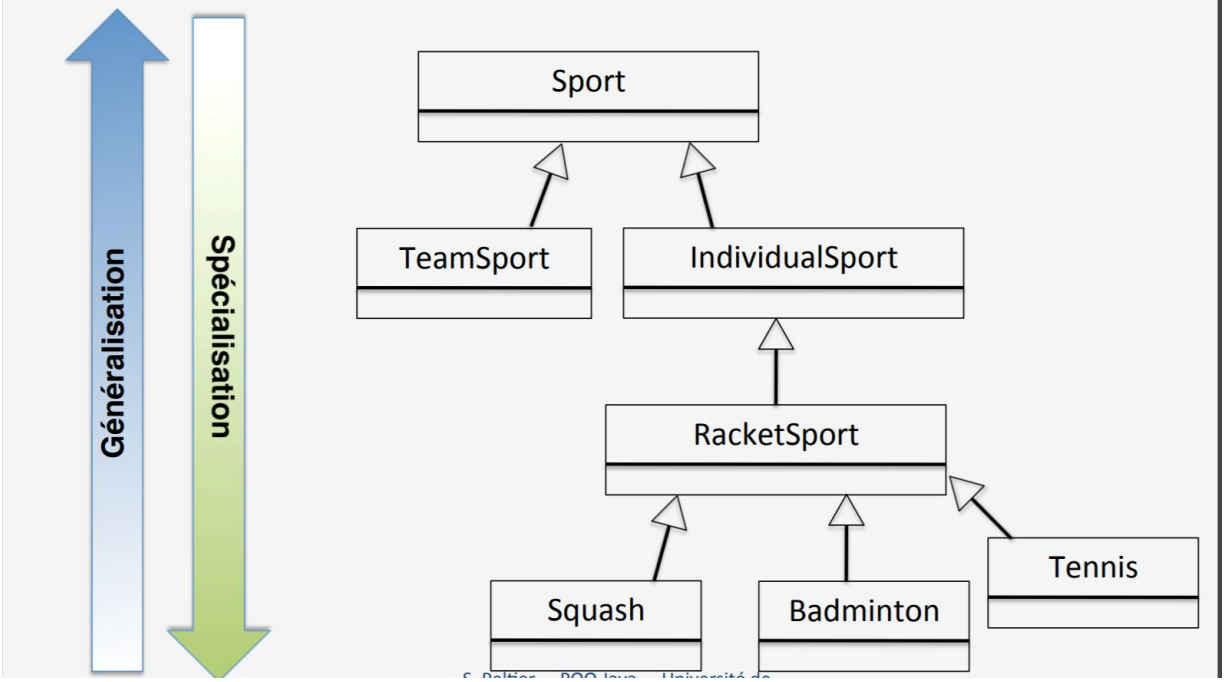
Au moment de la conception :

- s’appuyer sur des classes existantes et testées
- utilisation du polymorphisme

Au moment de la maintenance :

- corriger des erreurs dans une classe mère corrige du même coup les mêmes erreurs dans toutes les classes dérivées

Un exemple d'héritage



Héritage et ses implications

Le mécanisme de l'héritage permet de définir une classe (fille) à partir d'une autre classe (mère)

La classe fille:

- contient tous les attributs / méthodes de la classe mère
- peut être dotée d'attributs / méthodes supplémentaires
- peut redéfinir des méthodes de la classe mère
- peut à son tour être dérivée

Spécialisation des objets :

- ajout d'attributs / méthodes : **extension**
- ajout de méthodes de même nom : **surcharge**
- modification de méthodes existantes : **redéfinition**

Héritage :
Un objet de type `ClasseFille` est aussi du type `ClasseMère`. Autrement dit, tout ce que sait faire un objet de type `ClasseMère`, un objet de type `ClasseFille` sait le faire également.

Tester le **type** d'un objet avec `instanceof` :

```
if(myInstance instanceof Aclass)
{
    ...
}
```

Vocabulaire :

- Lien direct : *classe mère*, *classe fille*
- Lien hiérarchique : *sous-classe*, *sur-classe*
- La classe fille *hérite* de la classe mère

La syntaxe Java :

```
public class Boat extends Vehicule
{
    ...
}
```

Héritage et constructeurs
Un constructeur de la classe fille fait **toujours** appel à un constructeur de la classe mère.

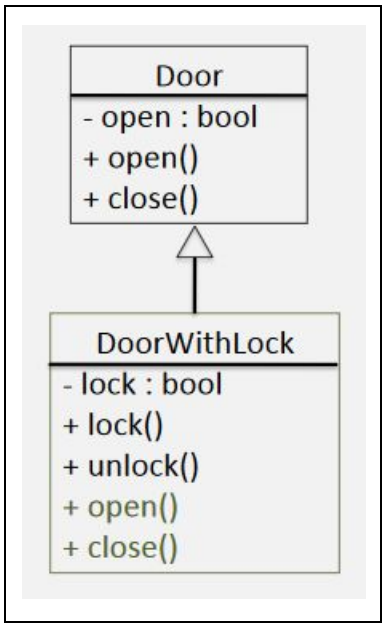
Soit de manière **explicite** : `super(...);`
→ (cet appel explicite doit être effectué en première instruction).

Soit de manière **implicite** : si l'instruction `super(...)` n'est pas explicitée dans le constructeur, alors java insère l'instruction `super();`

La visibilité protected
Un élément **protected** (# en UML) est accessible depuis les sous-classes et les classes du package.

Note : Pour les attributs, il est fortement recommandé d'utiliser une visibilité private et d'accéder aux attributs avec une méthode (éventuellement protected).

La redéfinition de méthode
Une porte s'ouvre et se ferme "directement", alors qu'une une porte avec verrou ne s'ouvre que si le verrou n'est pas enclenché.
Problème, en utilisant les méthodes open et close de la classe héritée on peut ouvrir et fermer une porte verrouillé. La solution est de les redéfinir dans notre nouvelle classe.



Exemple : redéfinition de la méthode open

```
public class AutoLockDoor extends Door
{

...

    @Override
    public void open()
    {
        if(!this.lock)
        {
            super.open();
        }
    }

}
```

Utiliser une méthode de la classe mère c'est super

- Appel d'une méthode de la classe mère : `super.method(...)`
- **Limitation** : en Java, on ne peut remonter que d'un niveau de définition. ~~`super.super.method();`~~

Héritage à plusieurs niveaux

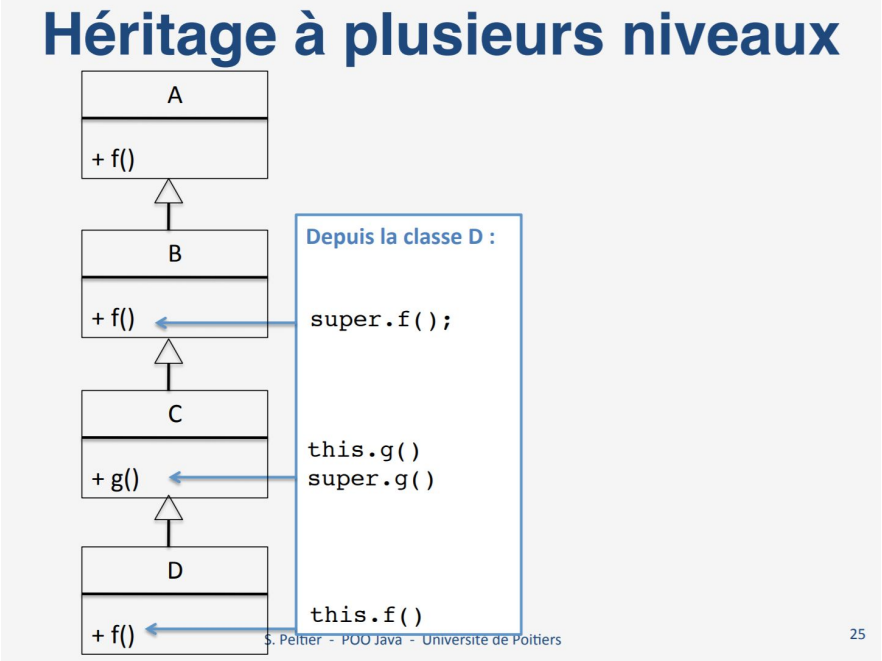


Schéma explicatif de l'héritage à plusieurs niveaux

La surcharge de méthode

Principe de la surcharge :

Possibilité de définir des méthodes possédant le même nom mais dont les paramètres d'entrée (et éventuellement les valeurs de retour) diffèrent.

Deux méthodes surchargées peuvent avoir des type de retour différents à condition qu'elles aient des paramètres différents.

Par exemple, on ne peut pas avoir une même classe les deux méthodes suivantes :

```
public boolean method(int x)
{
    ...
}

public int method(int y)
{
    ...
}
```

Surcharge vs Redéfinition

Redéfinition : spécialisation d'une méthode existante

Surcharge : ajout d'une méthode

Contrôle de l'héritage

Le mot clé final ne s'applique pas qu'aux données (constantes)

Il permet aussi :

- d'empêcher la redéfinition dans la classe fille

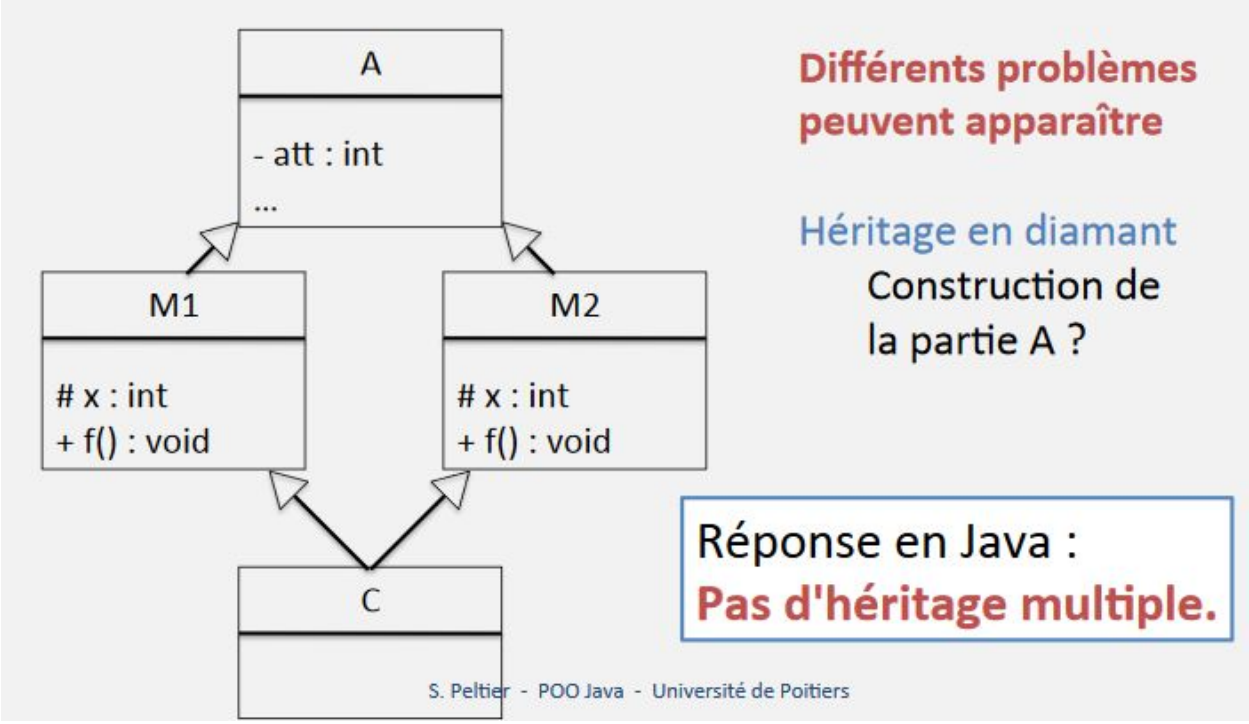
```
public final void aMethod()
{
    ...
}
```

- d'empêcher la création de sous-classes

```
public final class MyClass()
{
    ...
}
```

Héritage Multiple

Principe : une classe hérite de plus d'une classe



Polymorphisme

Le surclassement (upcasting**) :**

Utiliser un objet de type sous-classe comme un objet de type surclasse.

```
SuperClass obj = new SubClass(...);
```

A la compilation :

- Lorsqu'un objet est surclassé, il est vu par le compilateur comme un objet du type défini lors de la déclaration.
- Ses fonctionnalités sont donc restreintes à celles proposées par la surclasse

A l'exécution :

- Lorsqu'une méthode d'un objet surclassé est appelée, c'est la méthode de la classe effective de l'objet qui est invoquée.
- La méthode à exécuter est déterminée à l'exécution et non à la compilation.

On parle de liaison **tardive**, ou **lien dynamique**

Caractéristiques essentielle d'un langage orienté objet avec l'abstraction, l'encapsulation et l'héritage.

Avantages:

- Pas besoin de différencier différents cas en fonction de la classe des objets (avec **instanceof**)
- Maintenance du code plus aisée.
- Code facilement extensible

Un exemple:

Gestion d'un ensemble de formes géométrique.

On souhaite afficher le périmètre de chaque formes géométrique sans se soucier de chaque forme.

Le downcasting:

- **Intérêt** : forcer un objet à libérer les fonctionnalités cachés par le surclassement
- Conversion de type explicite (cast) déjà vu pour les types primitifs.

```
SuperClass obj = new SubClass(...);
((SubClass)obj).methodOfSubClass();
```

Pour que le <<cast>> fonctionne , il faut qu'à l'exécution le type effectif de obj soit compatible avec le type SubClass. Sinon, une exception ClassCastException est levée.

```
if(obj instanceof SubClass)
{
    ((SubClass)obj).methodOfSubClass();
}
```

La Classe Object

La classe Object:

- classe de plus haut niveau dans la hiérarchie,
- Seule classe qui ne possède pas de sur-classe

➔ **Toute instance est donc de type Object.**

Une classe qui n'a pas de clause extends hérite implicitement de la classe Object.

Quelques méthodes de la classe Object :

```
@Override
public boolean equals(Object o)
{
    if (o instanceof MyClass)
    {
        MyClass obj = (MyClass)o;
            return …
    }
    return false;
}
```

```
protected void finalize();

public String toString();
```

Les classes abstraites

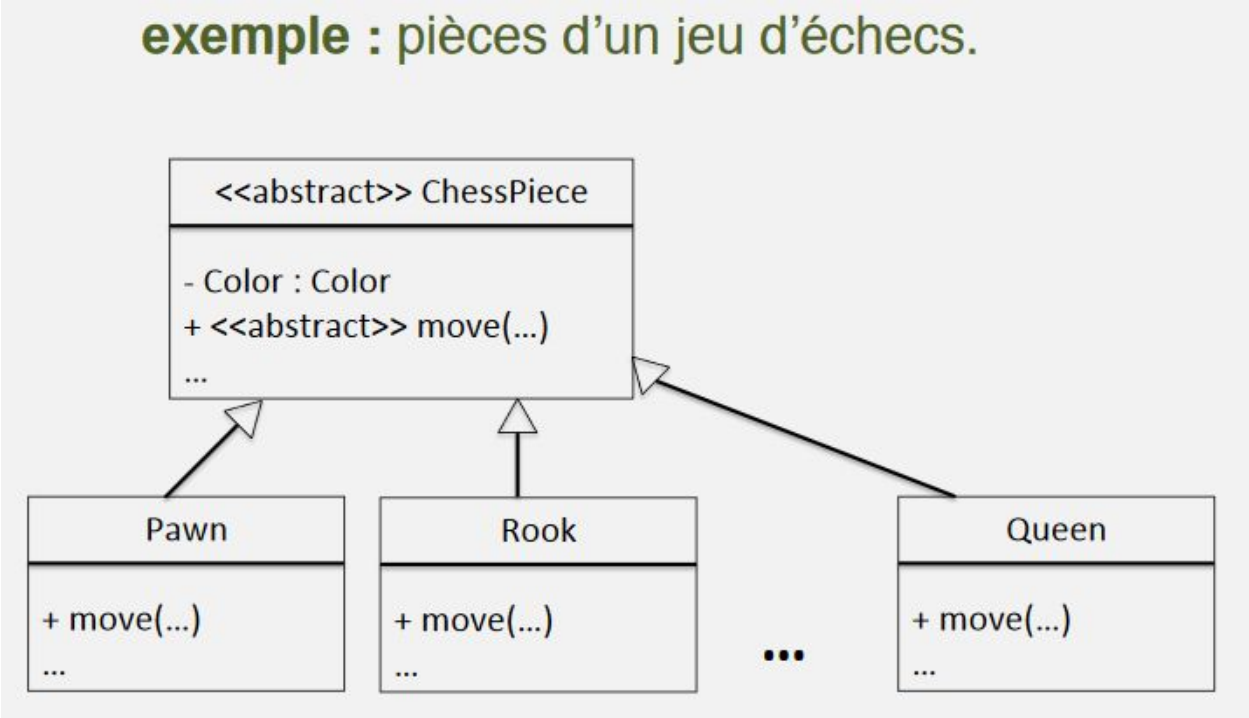
Intérêt :

- concevoir des applications avec un haut niveau d'abstraction
- il n'y a pas toujours un sens à donner une implantation par défaut d'une opération qui doit être redéfinie dans les sous classes (par exemple , le périmètre d'une forme géométrique)
- Force toutes les sous-classes concrètes à donner une implantation des méthodes abstraites

On peut écrire et compiler du code utilisant des méthodes abstraites.

Notation UML : *italique* ou << abstract>>

<<abstract>> ClasseAbstraite
+ <<abstract>> methodeAbstraite(...) ...



- Une classe abstraite **peut** avoir des méthodes abstraites (éventuellement aucune)
- Une classe contenant au moins une méthode abstraite est abstraite
- On ne peut pas **instancier** une classe abstraite.
- **Pour être un classe concrète**, une classe héritant d'une classe abstraite **doit implémenter toutes les méthodes abstraites** de sa classe mère.

Mot clé abstract :

- pour les classes

```
public abstract class ChessPiece
{
    ...
}
```

- pour les méthodes

```
public abstract void move(...);
```

exemple : ensemble de formes géométriques

-instanciation :

-direct :

```
ChessPiece obj = new ChessPiece(); ILLÉGALE
```

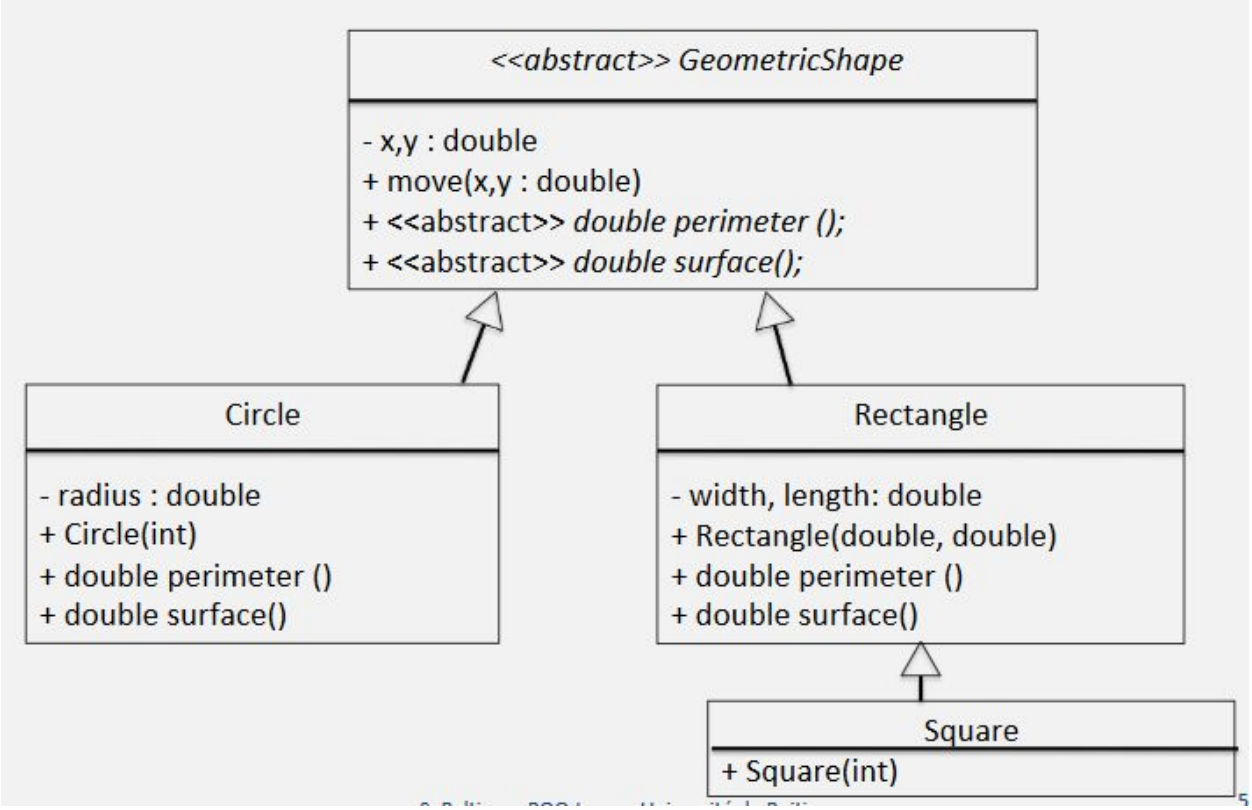
-Via une sous classe concrète :

```
ChessPiece obj = new Rook();
```

-Via une classe anonyme :

```
ChessPiece obj = new ChessPiece()
{
    @Override
    public void move(...)
    {
       ...
    }
}
```

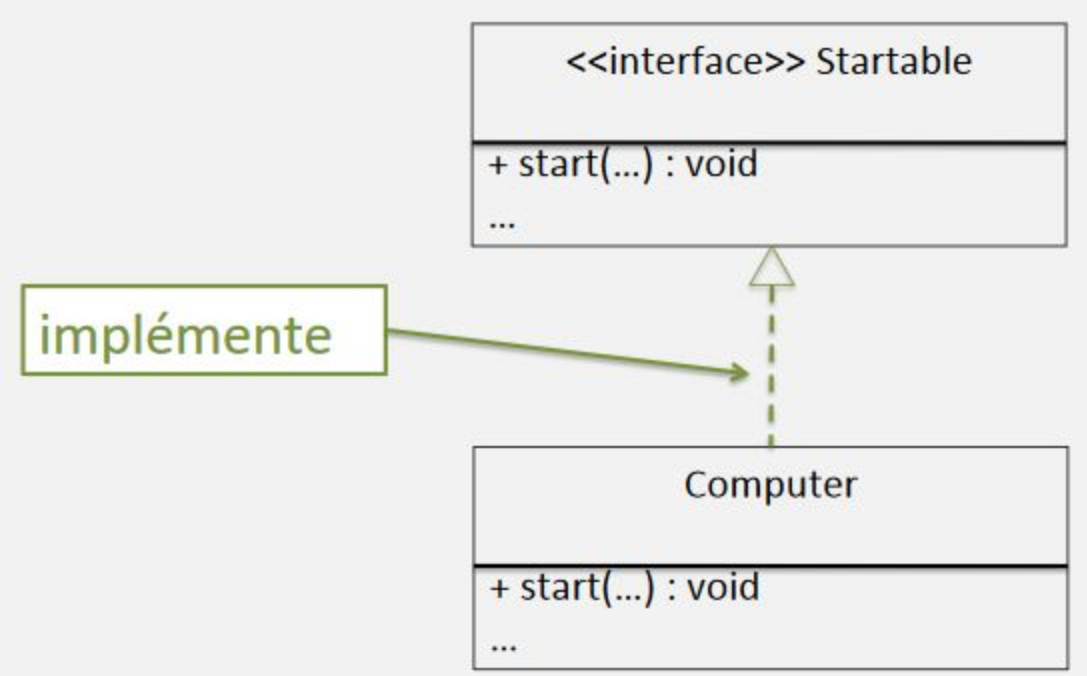
Implantation des méthodes abstraites



A force d'abstraction, on arrive finalement à la notion d'interface :

- *Attributs :* constantes publiques statiques
- *Constructeurs :* aucun
- *Méthodes :*
  - `public static`
  - `public abstract`

- `public default (implantation par défaut)`



Notation UML

Les interfaces

Pour créer une interface : mot clé **interface**

```
public interface Recyclable
{
    ...
}
```

Exemple : interfaces `Startable Recyclable`

Une classe **implémente** une interface

```
public class Car implements Startable
{
}
```

Une classe peut implémenter plusieurs interfaces

```
public class Car implements Startable, Runnable, Recyclable
{

}
```

Instanciation :

- Directe :

```
Startable obj = new Startable(); ILLÉGAL
```

- Via une classe concrète qui l'implémente :

```
Startable obj = new Computer();
```

- Via une classe anonyme :

```
Startable obj new Startable()
{
      // Implantation de toutes les méthodes
      // (facultatif pour celles qui ont une implantation par
défaut)
}
```

Pour créer une interface : mot clé **interface**.

```
public interface Recyclable
{
...
}
```

Exemple : interfaces Startable Recyclable

Sous-interface

Une interface **peut hériter d'une ou plusieurs interfaces**, on parle alors de sous-interfaces :

```
public interface MyInt extends Int1, Int2
{
...
}
```

Implantation par défaut (Java 8) :

```
default void toto()
{
      //default implantation
}
```

Et les problèmes d'héritage en diamant alors ?

→ Pas de problème lié à la construction de l'objet ! (voir slide p76)

```
IntA obj = new C();
obj.f(); // ?
```

Lorsqu'une classe "hérite" d'une méthode depuis plusieurs endroits (classe et/ou interfaces) :

1. La définition dans une classe est prioritaire
2. S'il n'y en a pas, la définition "default" d'une sous-interface est prioritaire.
3. S'il y a ambiguïté, la méthode doit être redéfinie dans la classe.

Accès explicites aux implémentations par défaut

```
public class C implements IntB1, IntB2
{
...
    public void myMethod()
    {
       IntB1.super.f();
       IntB2.super.f();
    }
}
```

Interfaces fonctionnelles

(Java 8) Interface qui possède une et une seule méthode sans implémentation par défaut.
Objectif: simplification d'écriture avec l'utilisation de lambda expressions

```
@FunctionalInterface // ← Annotation
public interface Operation
{
    ...
    public int eval(int x, int y);
}
```

Lambda expression
(Paramètre d'entrée) -> {implémentation};
exemples (cf java.util.fonction)

Sans Lambda :

```
IntPredicate even = new IntPredicate()
{
    @Override
    public boolean test(int value)
    {
       return value % 2 == 0;
    }
};
```

Avec Lambda :

```
IntPredicate even = (x) -> x % 2 == 0;
```

Interfaces, classes, classes abstraites : **Bilan**

- Une classe (abstraite ou concrète) :
  - Peut hériter d'une seule classe
  - Peut implémenter plusieurs interfaces
  - Peut être dérivée en classes abstraite ou concrète
- Une classe concrète

    - Ne possède aucune méthode abstraite
- Une classe abstraite
    - peut posséder des méthodes abstraites (éventuellement aucune)

- Une interface
    - Peut posséder des attributs public final static
    - Peut hériter d'une ou plusieurs interfaces
    - Peut posséder des méthodes public static (java 8)
    - Peut posséder des méthodes public abstract
    - Peut fournir une implantation par défaut de ses méthodes (Java 8)
    - fonctionnelle : possède une et une seule méthode abstraite

Les types énumérés

Première solution : créer une classe contenant uniquement des attributs static :

```
public class Jour
{
    public static final String LUNDI = "lundi";
    ...
    public static final String DIMANCHE = "dimanche";
}
```

→ Inconvénients :

- Pas de contrôle du type
- Pas d'itération possible

Bonne solution : créer un type enum

```
public enum Jour
{
    LUNDI, MARDI, ..., DIMANCHE
}
```

Utilisation :

```
public class MaClasse
{
    private Jour leJour = Jour.LUNDI;
}
```

Itération :

```
for(Jour j : Jour.values())
{
     ...
}
```

Ajout de méthodes :

```
public enum Jour
{
    LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE;

    public Jour lendemain()
    {
        Jour[] semaine = Jour.values();
        return (semaine[(this.ordinal() + 1) % semaine.length]);
    }
}


public enum Devise
{
       Livre(0.872), YEN(105.949), DOLLAR(1.378), ROUBLE(42.092);

       private double taux;

       Devise(double d)
     {
              this.taux = d;
       }

       public double tauxCourant()
       {
              return this.taux;
       }

       public void changeTaux(double t)
       {
              this.taux = t;
       }

       public double convertInEuros(double val)
       {
              return (val/this.taux);
       }
}
```

Les Generics

Il arrive que l'on ait une classe dont un attribut (ou une méthode) puisse avoir différents types (qui n'ont pas nécessairement de lien hiérarchique).
Première approche: utilisation d'Object

```
public class MyClass
{
    Object elt;

    public MyClass(Object o)
    {
       this.elt = o;
    }

    public Object getElt()
    {
       return this.elt;
    }
}
```

Inconvénients :

- Cast obligatoire

```
MyClass obj1 =  new MyClass(2);
MyClass obj1 =  new MyClass("abc");
Integer obj1 =  (Integer)obj1.getElt(2);
```

- Pas de typage fort à la compilation

```
Integer y = (Integer)obj2.getElt(); // erreur à l'exécution
```

Seconde approche : classe générique

```
public class MyGenClass<E>
{
    private E elt;

    public MyGenClass(E e)
```

```
    {
       this.elt = e;
    }

    public E getElt()
    {
       return this.elt;
    }
}
<a mettre les avantage de generic p 95~ du diapo>
```

Attention : Carre hérite de Rectangle, **mais** List<Carre> n'hérite pas de List<Rectangle>.

```
Rectangle r = new Carre(4); // pas de pb
List<Rectangle> lr = new ArrayList<Carre>(); // error : incompatible
```

Solution : utilisation de **wildcard**, noté "**?**"

```
public static void printList(List<GeometricShape> l)
{
    l.forEach(x -> x.print())
}

List<Rectangle> lr = new ArrayList<>();
printList(lr) // erreur à la compilation


List<? extends FormeGeometrique> l;
```

Se lit : "Une liste d'objets qui héritent de `FormeGeometrique`".

Un paramètre de type générique défini à l'aide d'une wildcard de type `<? extends ...>` ne peut pas être modifié.

```
public static void modif(List<? extends FormeGeometrique> l)
{
    l.add(new Carre(3)); // error à la compilation
}
```

On peut aussi utiliser les wildcards pour restreindre la sous-classe d'objets admis dans une collection :

```java
public static void modif(List<? super Carre> l)
{
    l.add(new Carre(3)); // OK
}
```

`List<? super Carre>` se lit “Une liste d’objets dont Carre hérite (directement ou non)”.
On pourra donc ajouter des carrés, mais pas des Cercles.

Les collections (Java collections Framework)

Une collection est un objet qui permet de manipuler un groupe d’objets.
Il existe différents type de collections, qui permettent :

- de manipuler des groupes d’objets ordonnées ou non
- d’autoriser les doublons ou non
- d’avoir l’objet null ou non

Implémentations (list, Set, Map):

- Dépend de l’utilisation
- Exemple : arraylist et Linkedlist
    - ArrayList : un tableau redimensionnable:
        + accès direct à un élément
        - ajout / suppression d’éléments

    - LinkedList : structure doublement chainée : <j’ai pas eu le temps p104>

Déclaration et instanciation :

```java
Set<Car> mySet = new HashSet<>();

List<Car> myList = new ArrayList<>(10);
List<Car> myList2 = new ArrayList<>();

Map<Car, Integer> myMap = new HashMap<>();
```

http://download.oracle.com/javase/7/docs/api/

Parcours de collections

- **boucle for each:**

```
List<GeometricShape> l = new ArrayList<>();
for(GeometricShape gs : l)
{
    ...
}
```

- **Iterator :**

```
List<GeometricShape> l = new ArrayList<>();
...
Iterator<GeometricShape> it = l.iterator();
```

  - `it.next()` : retourne le prochain élément de la collection . Le premier appel de la méthode next() retourne le premier élément.
  - `it.hasNext()` :retourne true si l'itérateur n'a pas parcouru toute la collection
  - `it.remove()` : supprime l'élément pointé par l'itérateur

```
List<GeometricShape> l = new ArrayList<>();
...
Iterator<GeometricShape> it = l.iterator();
while(it.hasNext())
{
    GeometricShape gs = it.next();
    if(gs instanceof Circle)
    {
       it.remove();
    }
}
```

Iterator -> seul moyen sûr de modifier une collection en itérant dessus.

L'utilisation d'un Iterator est le seul moyen sûr permettant de supprimer un élément d'une collection lors d'un parcours.

- **Interfaces fonctionnelles :**

```
List<GeometricShape> l = new ArrayList<>();

l.forEach(x -> x.print());
l.removeIf(x -> x instanceof Circle);
```

Streams

Permet de définir tout un **pipeline d'opérations** à effectuer sur des données provenant d'une source (collection tableau…)

Ne modifie pas les données initiales

Usage unique

➔ **Source :**

```
List<GeometricShape> l = new ArrayList<>();
```

➔ **Création :**

```
Stream<GeometricShape> s = l.stream();
```

➔ **Opérations Intermédiaires :**

```
filter, map, distinct,...
```

➔ **Opérations finales :**

```
reduce, forEach, collect, count,...
```

Un exemple :

```
List<GeometricShape> l = new ArrayList<>();
Stream<GeometricShape> s = l.stream();

double result =
            s.filter(x -> (x instanceof Circle))
            .map(x -> x.perimeter())
            .reduce(0.0, (x, y) -> x + y);
```

calcule la somme des périmètres des cercles

Chapitre 5 : Java avancé

- Duplication d'objets
- Refactoring
- Gestion des dépendances
- Création et utilisation de bibliothèques
- Javadoc
- Exceptions
- Entrées/Sorties

Refactoring

- **Déplacer, renommer** une classe
- **Renommer** un attribut, une méthode
- **Créer** un attribut, une variable à partir d'une valeur
- **Générer** une méthode à partir de code
- **Modifier** la signature d'une méthode
- **Faire remonter** un attribut dans la classe mère
- **Faire descendre** un attribut dans une classe fille
- **Extraire** l'interface

Gestion des dépendances

Gestion des packages

- Un package rassemble :
  - des classes et/ou des interfaces
  - des sous-packages

exemples de packages :

- java.lang : classes de bases String, Integer …
- java.util : classes utilitaires Collections, Timer

En java

- Une classe (ou une interface) = un fichier
- un package = un répertoire

Importer des classes et des packages

- Dans un programme Java, toutes les classes définies dans le même package ou dans le package java.lang sont incluses implicitement.
- Pour utiliser toute autre classe, il faut l'importer explicitement avec la directive **import**

Importation d'une classe ou d'un package

```
import monpackage.UneClasse;
```

```
import monpackage.*;
```

Note : l'instruction import monpackage.* n'importe pas les classes des sous-packages

Indiquer le package d'une classe :

```
package packagedelaclasse;
```

Note : cette instruction est la première du fichier décrivant la classe ou l'interface

Création de bibliothèques

L'archiver Jar (java Archive):

- l'outil jar permet de construire des bibliothèques (des fichiers .jar) à partir de bytecode (et d'autres fichiers, par ex. des images).

Création d'une archive à partir des fichiers du répertoire courant :

```
jar cvf monarchive.jar
```

Compiler en utilisant un jar :

```
javac -classpath monjar.jar monprojet/*.java
```

Lancer un programme qui utilise un jar :

```
java -classpath monjar.jar: monprojet/Main
```

Note: le classpath indique l'ensemble des répertoires (séparés par :) qui sont explorés pour compiler ou exécuter un programme java.

Le classpath

Deux manières de définir le classpath :

- il peut être défini pour chaque appel d'un outil de JDK (javac, java, jdb, …) avec l'option -classpath
  cf. exemple diapo précédente

- il peut être défini par la variable d'environnement CLASSPATH
  exemple bash
  `export CLASSPATH=path1:path2...`

Exceptions :

Qu'est ce que c'est ?

- c'est une signal qui intervient au cours de l'exécution d'un programme et qui indique qu'une instruction ne s'est pas déroulée normalement.

A quoi ça sert ?

- gérer les erreurs qui peuvent intervenir dans le déroulement d'un programme

Ne pas gérer les erreurs peut avoir de grave conséquence

Quelques exceptions disponibles :

- `CloneNotSupportedException`
  La classe d'un objet qui appelle la méthode clone() à n'implémente pas l'interface Cloneable
  <suite exemple page 181 et 182>

Créer une nouvelle classe exception :

```
public class MonException extends Exception
{
    ...
}
```

exemple : Point Exception.java

Lancer une exception avec throw :

```
if(...)
{
    throw new MonException();
}
```

Que faire d'une exception?

deux choses :

1) la propager
2) la traiter

Propager une exception :

<schéma 186>

- on indique que la méthode peut lancer des exceptions :

```
public void methodA (...) throws MonException
{
    ...
}
```

Note : une méthode peut lancer différents types d'exceptions .On indique alors les exceptions séparées par une virgule.

Récupérer une exception:

Si une instruction peut lancer une exception de type MonException alors on peut protéger son appel par un bloc try catch :

```
public void methodB(...)
{
    ...
    try
    {
      methodA(...);
    }
    catch(MonException e)
    {
      // ici, on a récupéré l'exception
    }
}
```

Le mot clé **finally** (optionnel):

- le bloc **finally** est exécuté à la fin, du bloc try.
- cela permet de :
    - factoriser du code qui serait sinon dupliqué dans différents blocs catch

- effectuer des instructions après le bloc try même si une exception a été lancée et non catchée

le mot clé finally:

```
public void methodB(...)
{
    ...
    try
    {
      // instructions
    }
    catch(...)
    {
      // instructions
    }
    catch(...)
    {
      // instructions
    }
    ...
    finally
    {
      // instructions
    }
}
```

Dupliquer un objet

Tous les attributs sont dupliqués
Copie de surface **Shallow copy** :

Si des attributs sont des référence , alors seule la référence est dupliquée.

Copie en profondeur **Deep copy** :

Si des attributs sont des références (mutables), alors les objets référencés sont dupliqués en deepcopy également.

***Exemple** : duplication de MyClasse obj*

Constructeurs par copie

```
public MyClass(MyClass obj)
```

Deep copy : il faut faire appel aux constructeurs par copie des attributs (référence mutables)

Implémenter l'interface cloneable
redéfinir la méthode clone (de object)

```
MyClass copy = obj.clone();
```

Si MyClass n'implément pas Cloneable :
`CloneNotSupportedException`

Deep copy : il faut aussi cloner les attributs

Serialization (cf la suite)

- permet de faire une deep copy

Les Entrées/Sorties

Les entrées/sorties sont basées sur la notion de **flux** (**stream**).

Un flux est un ensemble de données (e.g. flux de caractères, de bytes, d'objets…) dirigé depuis ou vers le programme

Manipulation de flux :

- création
- lecture
- écriture
- fermeture

Le package **java.io** permet de manipuler les flux.

Deux types de flux :

- flux entrants (**InputStream**) ont lus depuis une entrée (e.g. clavier, fichier, ...)
- flux sortants (**OutputStream**) sont écrits vers une sortie (écran, fichier)

Il existe des flux déjà ouverts

- System.in (InputStream) : flux (byte) d'entrée standard
- System.out (PrintStream) : flux (char) de sortie standard
- System.err (PrintStream) : flux (char) d'erreur

Exemple :

```
int c;
c = System.in.read();
while(c != -1)
{
    System.out.println((char)c);
    c = System.in.read();
}
```

Les E/S de types primitifs et de Strings

La classe **Scanner** permet de parser des flux de types primitifs (int, char, boolean) mais aussi des flux chaînes de caractères (String).

```
// exemple à partir de l'entrée standard
Scanner sc1 = new Scanner(System.in);

// exemple à partir d'une chaîne
Scanner sc2 = new Scanner("coucou ca va 123 ?");

// exemple à partir d'un fichier
Scanner sc3 = new Scanner(new File("toto_in.txt"));
```

Quelques méthodes de la classe Scanner (http://download.oracle.com/javase/7/docs/api/) :

- `hasNext()` : retourne vrai s'il reste un lexème dans le flux
- `hasNextInt()` : retourne vrai si le prochain lexème peut être interprété comme un int
- `next()` : retourne le prochain lexème (par défaut , délimité par un caractère d'espacement (character.isWhitespace())
- `nextInt()` : retourne l'entier correspondant au lexème suivant
- `useDelemiter(String pattern)` : permet de changer le délimiteur des lexèmes par une expression régulière

Les E/S d'objets

Création de flux d'objets :

- **ObjectInputStream** : flux d'entrée
- **ObjectOutputStream** : flux de sortie

Sérialization : enregistrement d'un objet dans un flux
Désérialization : récupération d'un objet d'un flux

> Pour pouvoir être sérialisé, un objet doit implémenter l'interface (vide) Serializable. Sinon, une exception sera levée.

La **Sérialization** :

- Création d'un flux de sortie d'objets :

```
ObjectOuputStream oos = new
ObjectOutputStream(newFileOutputStream("toto.ser"));
```

- **Sérialization** des objets :

```
oos.writeObject(maVoiture);
oos.writeObject(monCercle);
```

Note : On peut réaliser des objets de différents types dans un même flux. Mais lors de la désérialization, il faudra procéder dans le même ordre.

La **Désérialization** :

- Création d'un flux d'objets :

```
objectInputStream ois = new objectInputStream(new
FileInputStream("toto.ser"));
```

- Désérialization des objets (dans le même ordre) :

```
Voiture v = (Voiture)ois.readObject();
Cercle c = (Cercle)ois.readObject();
```

Note : La méthode readObject() retourne un Object

La sérialization d'un objet entraîne la sérialisation de tous ses attributs à l'exception des **transient**. Ils doivent donc être également sérializables.

> transient : empêche la sérialisation d'un attribut

> Sérialization : deep copy

- *(cf. in-memory serialization)*

Chapitre 6 : Documenter avec UML

- *Diagrammes d'états*
- *Diagrammes de séquence*

Pourquoi documenter ?

- *Travail en équipe*
- *Support pour les tests*
- *Maintenance*

Diagramme d'états

Description **dynamique** d'une classe à l'aide d'un automate fini (déterministe):
Représentation des différents cycles de vie possibles d'un objet.

Note : tous les diagrammes d'états ne sont pas pertinents.

Représentation graphique :

<schéma page 125>

État : situation où l'objet

- satisfait certaines conditions
- peut attendre un **événement**
- peut exécuter une **activité**

Représentation graphique :
<représentation page 126>

Transition : relation orientée entre deux états.

- peut être automatique ou déclenchée par un **évènement**.
- peut être conditionnée par une **garde**.
- peut également spécifier un **effet** (instruction de base , appel de méthode..)

Représentation graphique et notation :
<schema page 127>

Différentes types d'événements :

- Réception d'un **signal** : en général envoyé par un autre objet (ex : clic souris)
- **Appel** d'une opération : ce sont des méthodes de l'objet
- Écoulement du **temps** : mot clé `after(durée)`
- **Changement** d'une condition : mot clé `when(condition)`

Transition interne : définie dans un état.
Événements spécifiques :

- **entry** / effet : à l'arrivée dans l'état
- **exit** / effet : à la sortie de l'état
- **do** / effet : pendant l'état

Transitions propre : état départ = état arrivée

Note : une transition déclenche à chaque fois les effets entry et exit

Le pseudo état **point de choix** : une transition entrante et au moins deux transitions sortantes

<schéma page 130>

Exemple :distributeur de café:

<Schéma page 131>

Un état peut éventuellement être décomposé en sous-états,on parle alors d'**états composite**

<schéma page 132>

Utilisation d'un état historique.

<schéma page 133>

On peut indiquer qu'un état est un état composite, sans le détailler :

<schéma p 134>

Exemple : distributeur de café

Attention : dans notre exemple, le diagramme d'états ne spécifie pas entièrement la classe DistibuteurCafé.

Par exemple, que fait la méthode `insérerPièce` si l'instance en état *en préparation* ? (c.f. chapitre sur les tests)

zDiagramme de séquences

Un diagramme de séquence décrit **une fonctionnalité**, selon un point de vue temporel.

Il permet de représenter des **interactions entre des classes**.

Ligne de vie d'un objet : le temps est représenté par l'axe vertical

<schéma p 137>

Échange de message entre objets

<schéma p 138>

Les fragments combinés permettent de décrire entre autres les structures de contrôle usuelles.

<schéma p 139 >

Les acteurs:

Rôle joué par quelque chose ou quelqu'un d'extérieur au système et qui interagit avec (client, gestionnaire, imprimante…).

<schéma p140>

<schéma p141>

Chapitre 5 : Les Tests

- Pourquoi tester ?

- Tests structurels et fonctionnels

- Critères de tests

- Junit

Quelques bugs tristement célèbres

- Bug de l'an 2000
- Sonde Mariner 1 (1962)
- Ariane 5 vol 501 (1996)

Assurer la qualité logicielle

Logiciels critiques (transports, énergie, santé, militaire…) :

- Bugs inacceptables (vies, coût...)
- Norme, certifications

Autres logiciels :

- Niveau de qualité fixé par le client

Plus un bug est détecté tôt, moins il coûte cher.

"Testing is the process of executing a program with the intent of finding errors"- Sun Tzu, The Art of War

Vérification et Validation (V&V)

Vérification :

- Le logiciel fonctionne t-il correctement ?
    - “Are we building the product right?”

Validation :

- Le logiciel répond-il correctement ?
    - “Are we building the right product?”

Méthodes de V&V :

- tests statiques (revues de code)
- **tests dynamiques (exécution de code)**
- model checking, preuves formelles...

Test dynamique

- “Program testing can be used to prove the presence of bugs, but never their absence”

schéma

Différents types de tests :

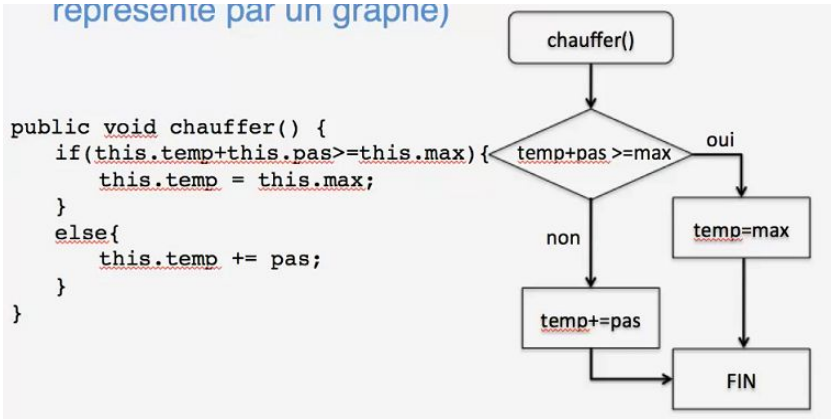
- **unitaires** (chaque méthode de chaque module)
- **intégration** (composition de méthodes ou de modules)
- **système** (conforme aux spécifications, performances, sur site)
- **utilisateurs** (conforme aux besoins)
- **non régression** (les évolutions du code n’ajoutent pas de bugs)

Critères de sélection

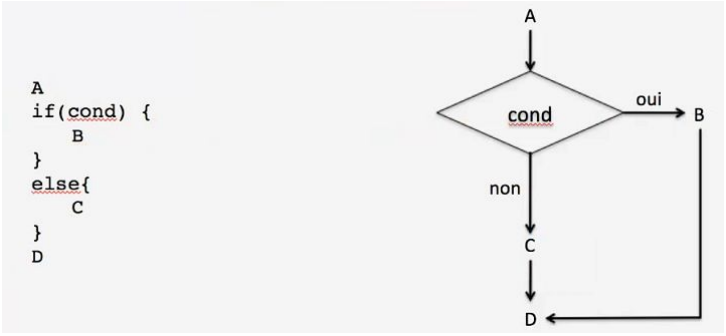
- **structurel** (tests boîte blanche)
  - tests conçus à partir de la structure du code
  - différents critères de couverture du code
- **fonctionnel** (test boîte noire)
  - on utilise uniquement l'exécutable
  - tests conçus à partir de la spécification

Ces deux critères de tests sont complémentaires.

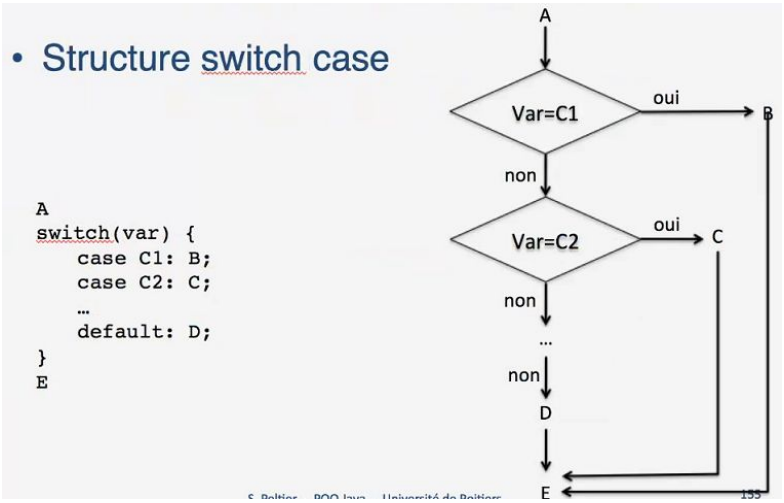
Tests boîte blanche



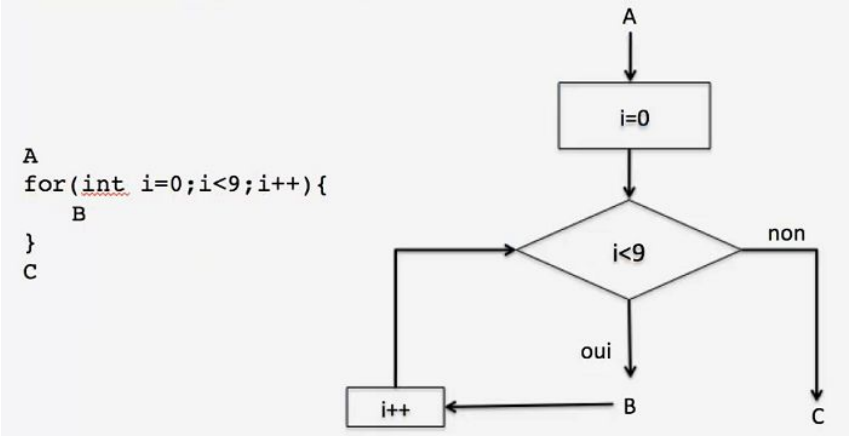
Graphe de flot de contrôle (programme représenté par un graphe)



Structure if then else

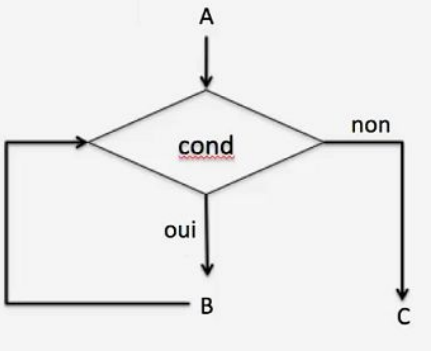


Structure switch case



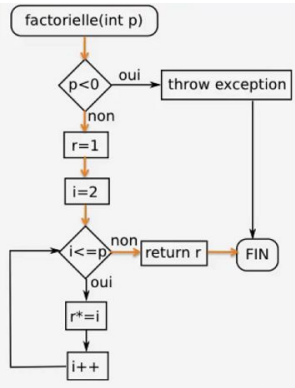
Structure boucle for

```
A
while(cond){
    B
}
C
```



Structure boucle while

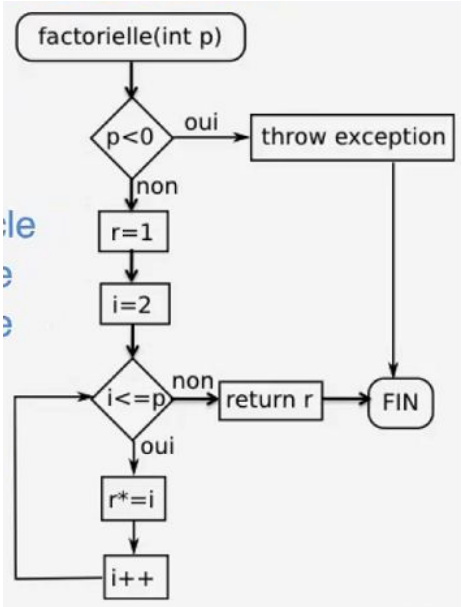
Activation d'un chemin



Prédicat de chemin
But : caractériser les tests qui caractérisent le chemin
Principe : condition → prédicat, instruction → substitution

```
!p < 0 && !i <= p [r = 1, i = 2]
soit p >= 0 && p < 2
```

Critères de sélection



Tous les chemins :

- levée de l'exception
- ne passe pas dans la boucles
- passe une fois dans la boucle
- passe deux fois dans la boucle
- …

Problème : nombre infini de chemins (et donc de tests)

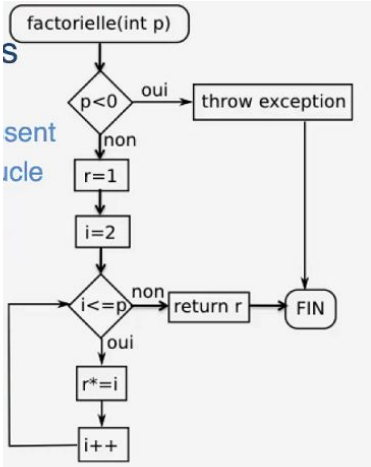
Critère inutilisable en général

Tous les chemins simples :

But : se limiter aux chemins qui passent 1 fois au plus dans chaque boucle

Trois chemins simples :

- levée de l'exception
- ne passe pas dans la boucle
- passe 1 fois dans la boucle

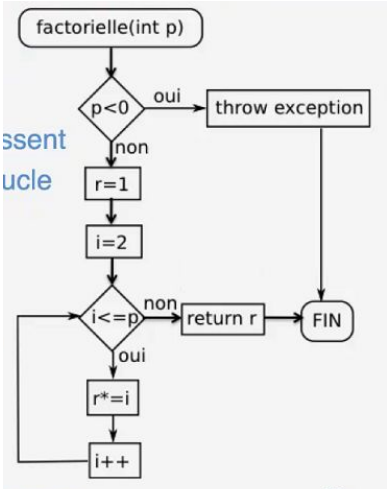


Tous les k-chemins :

But : se limiter aux chemins qui passent k fois au plus dans chaque boucle

cinq 3-chemins :

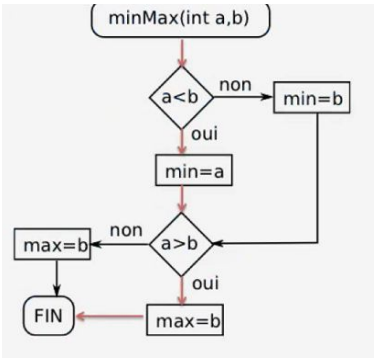
- levée de l'exception
- ne passe pas dans la boucle
- passe 1 fois dans la boucle
- passe 2 fois dans la boucle
- passe 3 fois dans la boucle



Chemins infaisables

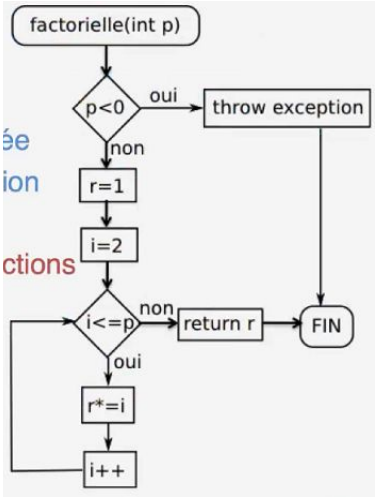
Prédicat de chemin
a < b && a > b

Aucune entrée de test n'active ce chemin !



Toutes les instructions

Définition : une instruction est activée lorsque le chemin contient l'instruction



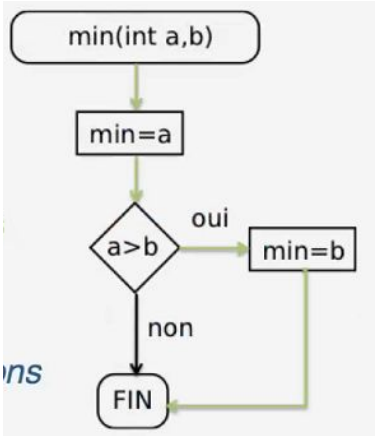
2 chemins activent toutes les instructions

- levée de l'exception
- passe 1 fois dans la boucle

Toutes les décisions

Définition : couverture des arcs

Chemin qui active toutes les instructions, **mais pas toutes les décisions !**



Critère plus fort que *toutes les instructions*.

Tests boîte noire

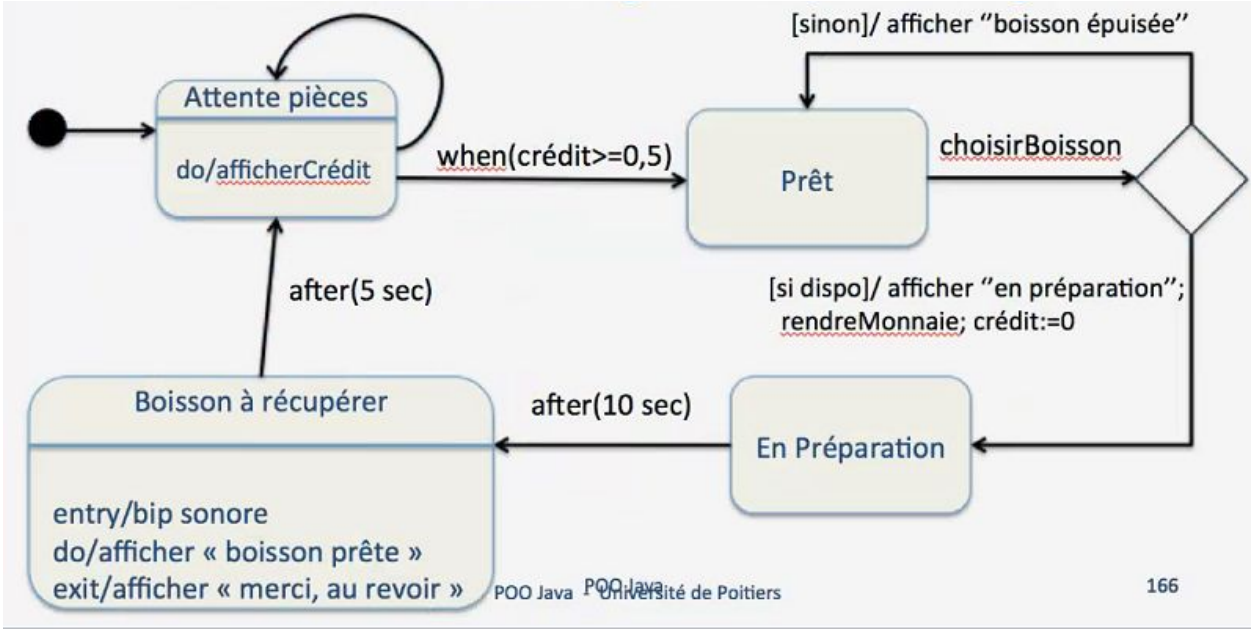
But : couvrir la spécification
- tests fonctionnels

Problème : une spécification n'est pas un objet formel
- pas de critère précis

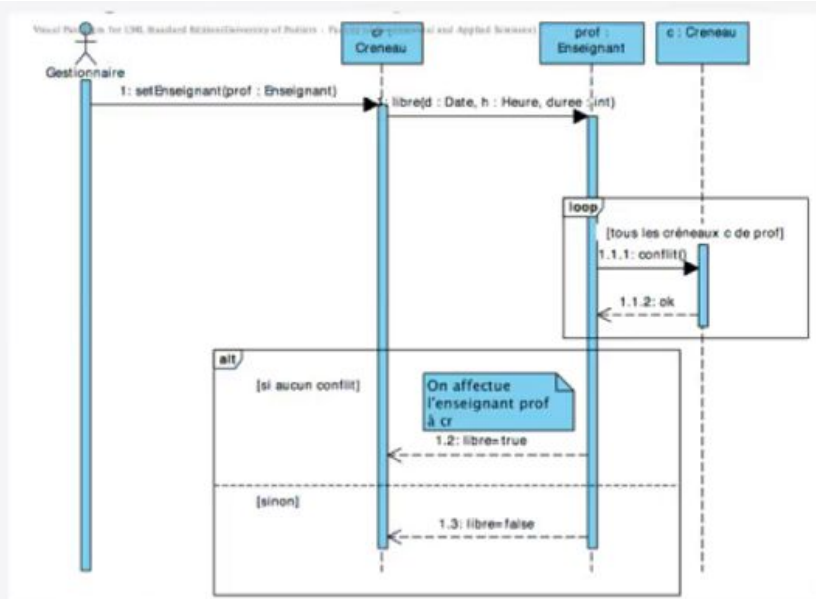
Solution : UML propose un cadre clair
- utilisation des diagrammes

Couverture d'un diagramme d'états :

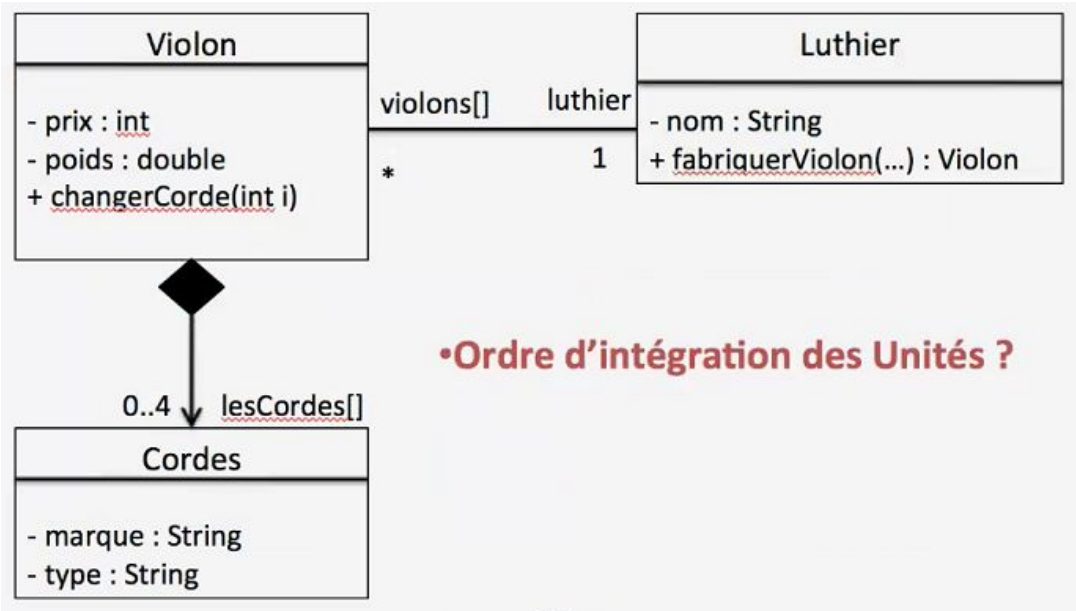
- Couverture de graphes (états, transitions, chemins...)
- Couverture des sous diagrammes puis du diagramme



Couverture d’un diagramme de séquences



Couverture d’un diagramme de classes



Tests unitaires

Chaque classe est testée unitairement dans une unité de test (avec bouchons si besoin)

- Critères structurels
- Critères fonctionnels

Les bouchons de test : remplacent les unités appelées par les résultats attendus

Tests d'intégration

Les unités (testées unitairement) sont ensuite assemblées et testées ensemble.

Approche Bottom-up :

- Tests des unités les plus basses (sans bouchon)
- Tests des unités plus hautes utilisant les plus basses

- **Avantage : pas de bouchon**
- **Inconvénient : localisation des erreurs** (bug non détecté lors du test unitaire)

Junit

Automatisation

- des tests
- du dépouillement