[Rappels Java]

I. Introduction

Une classe **(décrit un module)**, gère des données et mets à dispositions des méthodes **(“interface” pour utiliser le module)** pour les manipuler.

**Encapsulation** : en général l’état interne d’un l’objet, et ses méthodes, **ne sont pas accessibles** aux autres classes. Un objet peut être manipulé uniquement par son interface. : Séparer l’implémentation de la spécification

**Spécification** : définir un type de données par interface et ses propriétés (entrée et sortie)

* type abstrait de données (ADT)

**Implémentation** : structure de données + code

**Exemple de spécification (“contrat”) :**

Type abstrait (notation formelle):

**NOM**

Pile <T>

**FONCTIONS**

vide : Pile <T> -> Boolean

nouvelle : -> Pile <T>

empiler : T × Pile <T> -> Pile <T>

dépiler : Pile -> T × Pile

**PRECONDITIONS**

dépiler(s: Pile) <=> (not vide(s))

**AXIOMES**

pour tout e in T, pour tout s in Pile <T>

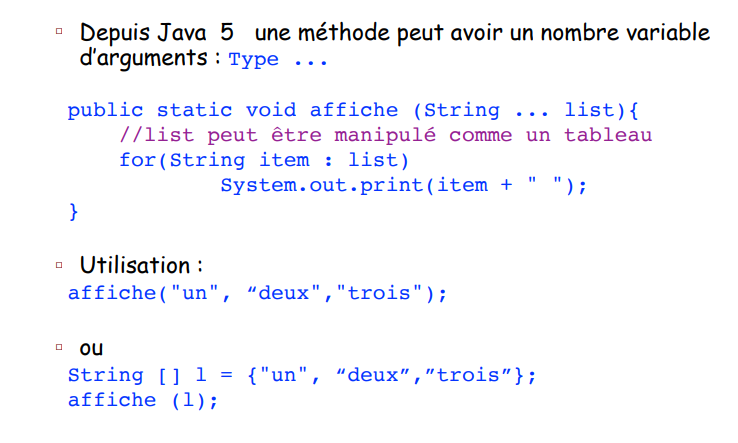
vide(nouvelle())

not vide(empiler(e,s))

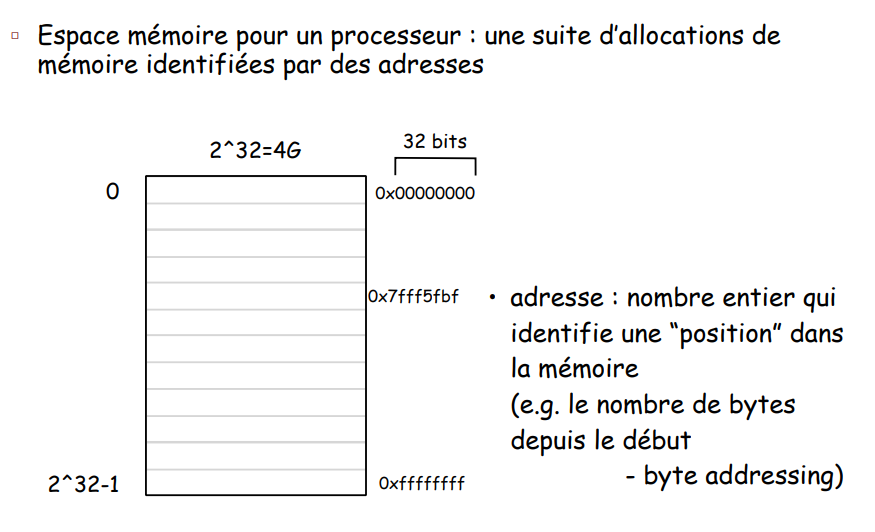
dépiler(empiler(e,s))=(e,s)

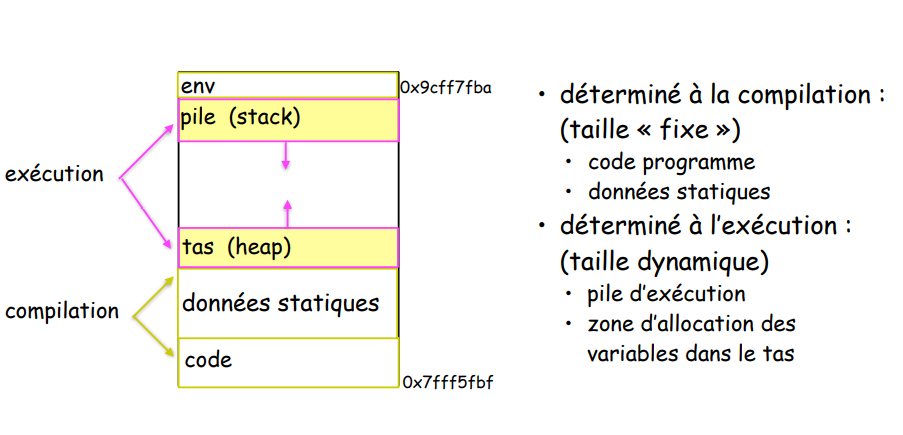
II. Rappels CI et IP ++

* Fonctions de la pile (frame - record d’activation)
* Signatures des méthodes
* This() : pour appeler un constructeur dans un autre ça doit être la 1ère ligne
* Variables/Méthodes d’instances et de classes



III. Objets et mémoires





IV. Préserver l’encapsulation

**But** : éviter de retourner une référence à un champ privé qui soit un objets mutables (=/= immuables : Stringte…)

* Clonage
* Exceptions : parfois on souhaite explicitement donner accès à une partie de l’état interne d’un objet avec droit de modification

V. Règles de visibilité des noms

* Un nom introduit **dans un bloc de classe** est visible partout dans ce bloc et **pénètre dans les blocs imbriqués de tout niveau** et être **redéfini dans un bloc imbriqué dans celui-là** (de tout niveau d’imbrication).
* Un nom introduit **dans un bloc qui n’est pas de classe** est visible dans le bloc, **y compris ses blocs imbriqués**, mais **uniquement à partir de son introduction**. Il n’est pas visible en dehors du bloc, et ne **peut pas être redéfini** plus loin dans le même bloc (**même pas** dans un de ses **blocs imbriqués**)

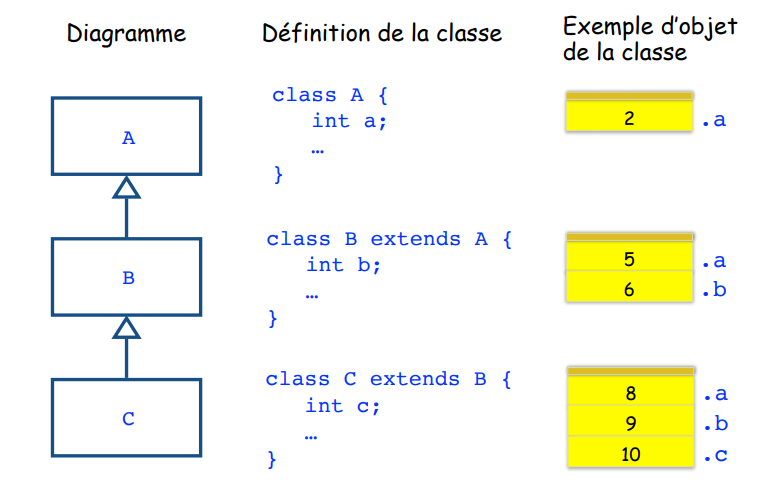
[Héritage]

# Introduction

L’héritage est un mécanisme de **réutilisation de code**. Il permet de réutiliser les champs et méthodes (constructeur non inclus) d’une classe, **tout en y ajoutant**.

| public class A { ... } public class B extends A { ... // champs et méthodes additionnelles/différentes} |
| --- |

* B : **sous-classe** (ou classe enfant)
* A : **super-classe** (ou classe parent)

Une classe peut avoir **plusieurs sous-classes**

Une classe peut avoir **une seule classe parent**

La hiérarchie est en général arborescente : **la classe Object est la racine de cette hiérarchie**

# Constructeurs de sous-classes

Les champs hérités et les champs propres doivent être initialisés : pour invoquer le constructeur d’une classe parent on utilise : super () à la première ligne.

Super () et this (), ne peuvent pas être dans le même constructeur. Si aucun des deux n’est invoqué, un appel super() (sans arguments) est implicite : erreur si la classe parent n’a pas de constructeur sans arguments

# Rédéfinitions

Un champ avec le même nom d’un champ de la super-classe cache ce dernier (accessible avec super.champ)

Il est possible de redéfinir des méthodes de la classe parent : la méthode doit avoir la même signature.

La méthode dans la classe parent est encore accessible avec super.methode().

Le modificateur d’accès peut changer dans une redéfinition uniquement pour élargir l’accès

On peut noter une redéfinition avec @Override

Le modificateur **final** interdit la redéfinition pour une **méthode**.

# Polymorphisme

**Polymorphisme** : une même variable peut référencer des objets de plusieurs types différents pendant sa vie.

Un objet a un type déclaré (déclaration) et un type effectif (référence)

* **getClass** retourne le type effectif d’un objet
* **obj instanceof A** : retourne true si A est un ancêtre du type effectif de obj

Un objet de type effectif C est considéré également de tous les types de plus haut niveau dans la hiérarchie d'héritage

Le polymorphisme permet de traiter des objets uniformément avec les méthodes qu’ils ont en communs. Les fonctions redéfinies sont aussi utilisable.

**Dynamic binding** : C’est le type effectif d’un objet qui détermine la définition de la méthode à utiliser

* le type déclaré détermine quelles méthodes on peut invoquer
* le type effectif détermine quelle définition est à utiliser pour ces méthodes

**Le dynamic binding ne s’applique pas au variables !**

System.out.println(**b**.c); // affiche 4

System.out.println(**ab**.c); // affiche 3, static binding

System.out.println(**((B)ab)**.c); // affiche 4

**Une méthode static peut être redéfinie ou surchargée mais même comportement que les variables**

# Transtypage (casting)

Pour utiliser un objet entièrement il est possible de forcer le changement de son type déclaré

* CompteB cb = new CompteC(“AG675”, 0, 200);
* CompteC cc = (CompteC) cb;

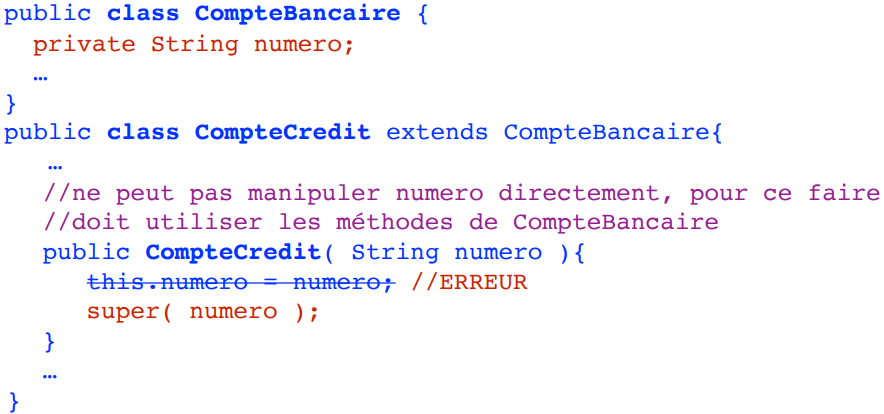
Sur les types référence, le casting est autorisé uniquement si les types déclarés appartiennent à une même hiérarchie d’héritage

* **Upcasting** : toujours sûr
* **Downcasting** : sûr uniquement s’il ne dépasse pas le type effectif => Toujours tester avant un downcasting

# Contrôle d’accès à la super-classe

Comme toute autre classe, la sous-classe :

* **n’a pas accès aux membres private** de la super-classe
* a accès aux membres avec visibilité package uniquement si elle se trouve dans le même
* **a accès aux membres protected et public** de la super-classe



# Ordre d’initialisation

(voir BBB pour explication)

# 

# 

# 

# 

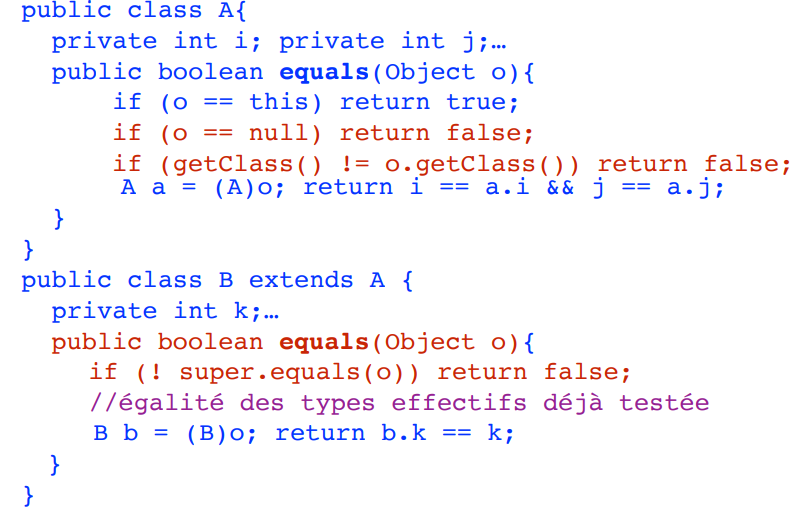
# 

# La classe Object

## La redéfinition de equals()

On teste avec instanceof + ses champs spécifiques

* Quand c’est dans une sous-classe B, la classe A devrait tester égalité des types effectifs



## La redéfinition de hashCode()

(voir BBB pour explication)

## La redéfinition de toString()

(voir BBB pour explication)

## La redéfinition de clone()

(voir BBB pour explication)

[Interfaces]

# Classes abstraites

Classe abstraite : classe dont certaines méthodes n’ont pas définition (seulement la signature est fournie)

* Impossible de créer des objets
* Sert comme base pour l’héritage
* Une méthode sans définition doit avoir le modificateur abstract
* Une classe qui contient des méthodes abstract doit avoir elle même le modificateur abstract

Une interface est une **collection de signatures de méthodes**. Elle peut contenir également des champs constants et des classes internes, mais **pas des variables d’instance**. Elle peut être abstraite.

Une sous classe peut **implémenter plusieurs interfaces**. Permet d’écrire du code générique.

Les méthodes d’une interface sont automatiquement publiques. En revanche les classes qui implémentent l’interface doivent utiliser le modificateur public pour les méthodes redéfinies de l’interface.

Les méthodes d’une interface peuvent être static (Java 8). Dans ce cas elles doivent avoir une définition.

Depuis Java 9 il est possible d’introduire des **méthodes private d’interface**, elles sont **visibles uniquement dans l’interface**, donc peuvent être **utilisées uniquement par les autres méthodes de l’interface** qui ont une définition. Elles **doivent être définies** (avoir un corps), statiques ou pas.

Depuis Java 8 une interface peut fournir une définition par défaut des méthodes. Les méthodes avec une définition doivent porter le modificateur default (sauf private ou static. Les classes qui implémentent cette interface héritent alors les définitions des méthodes par défaut, et **peuvent** ne pas les redéfinir.

Les champs d’une interface sont automatiquement : public static final implicite

Une classe qui implémente une interface doit en redéfinir toutes les méthodes abstraites. Sinon la classe doit être déclarée abstraite

# Exemples d’interface

Cloneable: est une interface vide un objet qui l'implémente peut redéfinir clone()

Comparable: permet de comparer les éléments

Runnable: permet de définir des “threads"

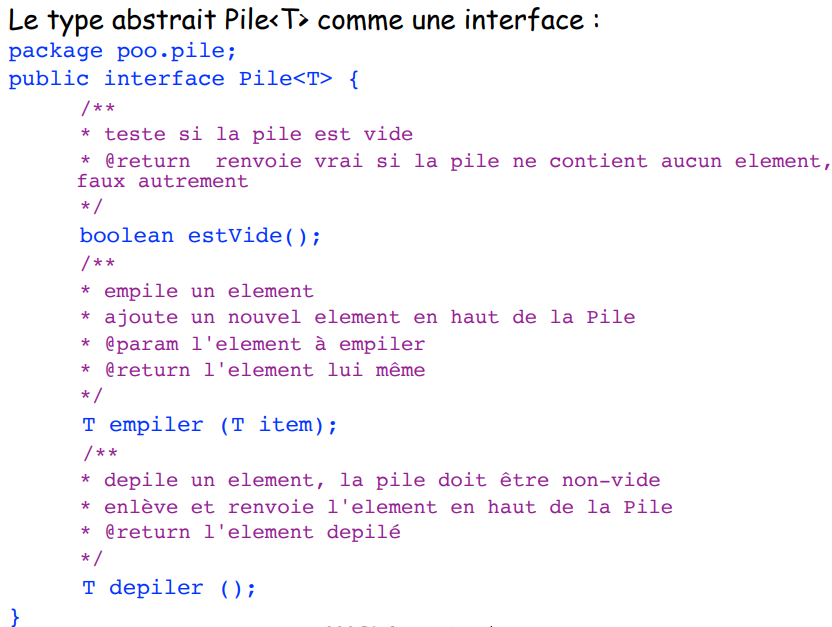
Serializable: un objet d’une classe qui implémente peut être "sérialisé" = converti en une suite d'octets pour être sauvegardé

# Héritage multiple

Des conflits de définitions peuvent être générés par l’héritage multiple. Il faudra enlever l’ambiguïté avec la notation : **NomClasse.nomConstante**.

Si A et I (interface) possèdent une méthode avec la même signature B **hérite la méthode de la classe mère A**, une éventuelle définition de la même signature dans I est ignorée.

Si deux méthodes ont la même signature en I1 et I2 et au moins une des deux a une définition, **la classe B doit redéfinir la méthode**. Dans la classe B on peut faire référence à l’une ou l’autre des deux fonctions conflictuelles, si concrète (non abstract).



Une pile peut être implémentée de plusieurs façons, tout en respectant la même interface

[Classes internes et expressions lambda]

# Classes internes

**Définition** : Classes définies dans d’autres classes

**Classes membres (CM)** : membres d'une autre classe (statiques ou non)

**Classes locales** : classes définies dans un bloc de code

* **Classes anonymes** : classes locales sans nom

## Membre non statique d'une classe englobante

Utilité :

* Définir des classes qui ont du sens uniquement en relation à la classe englobante (CE)

Ex. un Curseur qui parcourt les pages a du sens seulement si attaché à un TextDocument

* Quand la classe membre est private : limiter la visibilité d’une classe à une seule autre classe

Ex. Un Curseur est utilisable uniquement par un TextDocument

Un objet de la CM est créé uniquement à partir d’un objet de la CE : **B b = a.new B ()** (a dans la CE A)

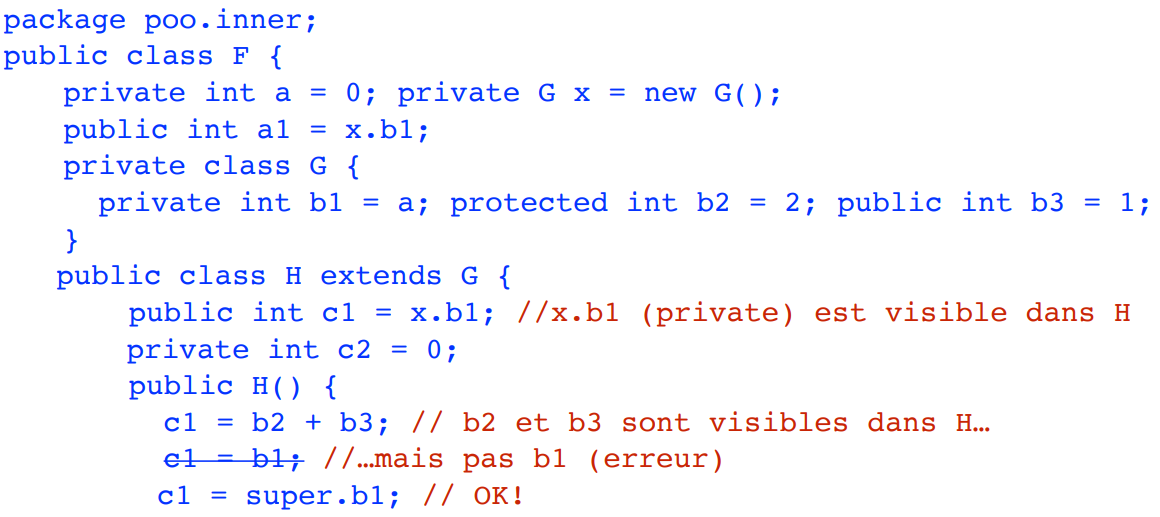
=> L’objet de la CE est donc accessible depuis l’objet de la classe interne

Dans la classe interne la référence implicite à l’objet de CE est **NomClasseEnglobante.this** mais elle peut être omise en général (sauf si occultée)

La CM **a accès aux autres membres de la CE** indépendamment de leur modificateur d’accès

La **CE a accès à ses classes membre, ainsi qu’à leur champs et méthodes**, indépendamment de leur modificateurs d’accès (utile en dehors de la CE)

Une classe fille d’une CM (bien que interne à la même classe englobante) **maintient les règles habituelles de visibilité des champs hérités**

****

Si une CM est visible en dehors de sa CE, on la note :  
**NomClasseEnglobante.NomClasseMembre;**

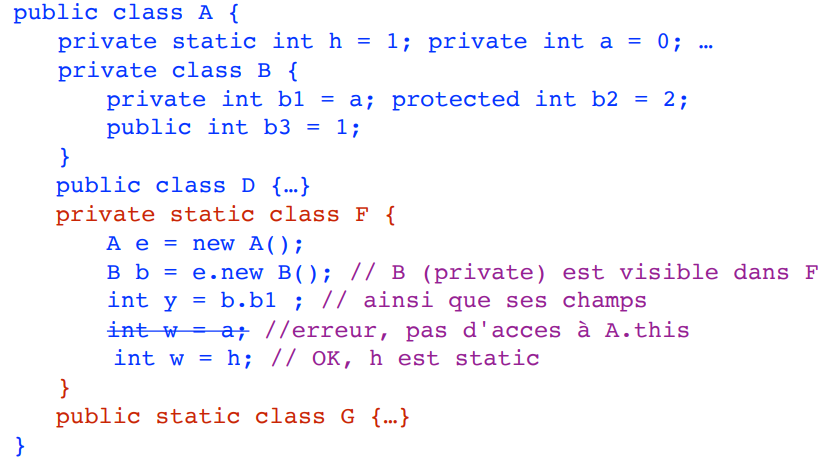
On peut créer ses objets à partir de n’importe quel objet de la CE ainsi :   
**F a = new F();   
F.H x = a.new H();**

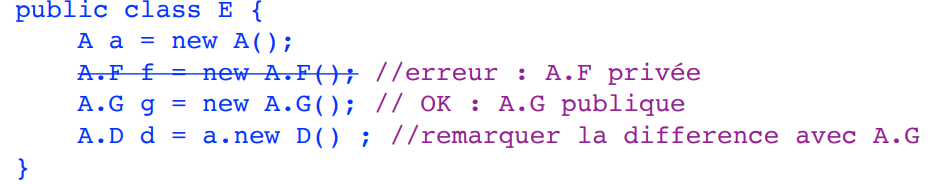
## Membre statique d'une autre classe :

* Class ou interface
* A accès uniquement aux champs static de la CE
* Même règle d’accès que non static
* Pas d’accès CE.this mais CE.CM : On peut l’éviter avec un import

Utilité :

* Définir des classes/interfaces dont les objets sont à utiliser en relation avec les objets d’une CE, sans nécessiter une référence à un OE
* Regrouper les classes dans une hiérarchie logique
* Private : définir une classe utilisée dans une seule classe et ne nécessite pas de réf. à un OE





## Façons d’étendre une classe membre

| (1) class Externe{  class Interne {}  class InterneEtendue extends Interne{}}  (2) class Externe{  class Interne {} }  class ExterneEtendue extends Externe{  // Interne est aussi une classe interne de  // ExterneEtendue  // elle est héritée comme les autres membres  // elle peut donc être étendue ici  class InterneEtendue extends Interne{...}  Interne r = new InterneEtendue();  } |
| --- |

(3) Interne peut être étendue en dehors de sa CE

| (Non) static class Autre extends Externe.Interne {  Autre((Externe r,) int i){ (r.)super(i);}  } |
| --- |

## Classe locale

Classes définies à l'intérieur d'un bloc de code

* une CI locale n'est pas membre de la CE
* elle est visible uniquement dans le bloc de code

Usage : créer des objets d'une classe/ extensions de classes qui n'ont de sens que localement

Une classe locale a accès :

* aux membres de la classe englobante
* aux variables locales visibles dans le bloc de code englobant (à condition que ces variables locales soit “effectively final” : variable jamais modifiée après sa création)

## Classe anonyme

Une classe anonyme est une classe locale sans nom : on la déclare en créant un nouvel objet suivi d’accolade

# 

# 

# 

# Expressions lambda

Une expression lambda est un bloc de code, précédé par la liste de tous les paramètres utilisés dans ce code

| (int k) -> {  for (int i = k; i >= 0; i--) System.out.println(i);  } |
| --- |

une expression lambda peut être utilisée à chaque fois qu’on s’attend un objet de type I, où I est une interface fonctionnelle dont la méthode abstraite est compatible avec l’expression lambda

Parfois la fonction qu’on veut passer en paramètre existe déjà. On passe donc une référence à la méthode

* **nomClasse :: nomMethode**
* **o :: nomMethode** (objet o)
* Référence de constructeur : **NomClasse::new**

**Exemples**

* Math::pow <=> x -> Math.pow(x)
* System.out::println <=> s -> System.out.println (s)
* String::concat <=> (x,y) -> x.concat (y)

Comme une classe locale, elle a accès :

* à tous les champs et méthodes de la classes englobante, indépendamment de leur modificateur d’accès
* à toute variable locale visible dans le bloc de code englobant, qui soit “effectively final”

Différence avec les classes locales / anonymes : this dans une expression lambda fait référence à l’objet de la classe englobante et non pas à l’objet de la classe anonyme implicitement crée

[Exceptions]

Tout programme doit prévoir la présence d’erreurs

* comportements imprévus du programme
* entrées non valides
* erreurs de communication avec les périphériques / le réseau

Une méthode peut terminer

* dans un état normal : l’exécution de la méthode a terminé correctement
* dans un état d’erreur : une exception a été soulevée pendant l’exécution de la méthode, la méthode termine avant sa fin

Le contrôle revient à la méthode appelante

* Dans le cas de terminaison normale l’appelant reçoit la valeur de retour de la méthode
* Dans le cas de terminaison en état d’erreur, l’appelant reçoit un objet de type Exception, qui peut contenir des informations sur la situation d’erreur qui l’a produite

L’appelant a deux choix :

* gérer (“capturer”) l’exception et continuer son exécution
* ne rien faire : dans ce cas soulève à son tour l’exception, termine en situation d’erreur et rend le contrôle à son appelant, …

| try { <code qui peut soulever une exception> } catch (<Type exception> e) {  <code qui gère l'exception>  //exécuté uniquement si le bloc try soulève une exception de <Type exception> } //suite de la méthode exécutée : // - directement après le bloc try //- après le block catch, si une exception a été capturée |
| --- |

Remarque : si un exception d’un type incompatible avec est soulevée pendant l’exécution du bloc try, la méthode termine en état d’erreur avant de compléter le bloc try et soulève l’exception non-capturée

Afin de debugger le programme le bloc catch peut afficher l’état de la pile d’exécution au moment de la capture de l’exception

| catch (ArrayIndexOutOfBoundsException e) {  e.printStackTrace(); System.out.println ("pas calculé");  } |
| --- |

Exception “checked” et “unchecked”

**Checked :** Classes qui dérivent de Exception, mais pas de RuntimeException : erreurs qui dépendent de conditions externes (I/O, fichiers,réseaux,…)

**Unchecked :** Classes qui dérivent de RuntimeException

* Erreur du programmeur. En général évitables avec des tests (pointeur null, index sup à la dernière position d’un tab,…)
* Les exceptions de ce type sont souvent capturées uniquement pour garantir une sortie “gentille” du programme

# Contraintes sur les exceptions checked

Toute méthode qui peut soulever une exception checked doit le déclarer :

| public void fullRead(String filename) **throws FileNotFoundException** {  InputStream in = new FileInputStream(filename); int b; while ((b = in.read()) != -1){…}  } |
| --- |

Si la méthode capture l’exception elle ne peut plus la soulever donc pas de déclaration throws

# Lancer une exception

Une exception peut être lancée automatiquement par une opération “primitive” (accès à un tableau, ….). Ou par l’utilisation d’une méthode qui en soulève une. Mais on peut également lancer une exception (checked ou unchecked) explicitement quand on détecte une situation d’erreur

On peut capturer plusieurs exceptions dans le même bloc

# La clause finally

Contient du code qui sera exécuté immédiatement après le bloc try-catch, quelle que soit la façon d’en sortir.

Façons possibles de sortir d’un bloc try-catch :

* par l’exécution correcte du bloc try
* par l’exécution d’un des blocs catch
* par une exception soulevée par le bloc try ou par le bloc catch et pas capturée
* par un return exécuté dans le bloc try ou dans le bloc catch

[Interfaces graphiques]

| **Class interne static** | **Obj, int,...** | **méthode** |
| --- | --- | --- |
| Créer static | **oui** | **Oui accès par CI.methode** |
| Créer non static | **oui** | **Oui accès par l’objet CE.CI b = new CE.CI()** |
| Accès static | **oui** | **Oui, accès par CE.methode** |
| Accès non static | **Non sauf si l’objet est créé dans la classe interne (et utilisé dans une méthode non static)**  Dans CI  A aOb = new A();  aOb.membre; | **non** |
| Static affecter à... | **oui** | **/** |
| Non static affecter à membre CI | **Non sauf dans une variable non static**  Dans CI  A aOb = new A();  int b = aOb.membre; | **/** |
| **Class interne non static** | | |
| Créer static | **Non** | **non** |
| Créer non static | **oui** | **oui** |
| Accès static CE | **Oui CE.membre** | **Oui CE.methode** |
| Accès non static CE | **Oui CE.this.membre** | **Oui CE.this.methode** |
| Static affecter à... | **oui** | **/** |
| Non static affecter à | **oui** | **/** |