Programmation orienté objet Java

Java est un langage qui se base seulement sur la notion d’objet, développé la 1 ère fois par Sun Microsystems en 1995.

**Garbage Collector :** est un processus de la gestion mémoire qui s’exécute automatiquement pour récupérer la mémoire qui n’est plus référencer.

**La compilation en Java :** un fichier source Java (.java) est traduit par le compilateur java (javac.exe) en un code intermédiaire (n’est pas binaire) appelé bytcode (.class) qui peut être interpréter dans n’importe quel environnement (system d’exploitation) á l’aide d’une machine virtuelle (Java.exe) qui a comme fonction d’exécuter les fichiers.class.

C:\ … > javac Test.java 🡪 pour générer le bytecode associer (fichier .class)

C:\ … > java Test 🡪 Exécution de Test.class (JVM interprète le fichier.class selon l’environnement utiliser)

**JDK (Java Developpement Kit) :** ensemble des programmes nécessaires pour le développement d’application sous java, Il regroupe ainsi les programmes javac.exe (java compilateur), java.exe (JVM), jar.exe (programme pour fabrique ces propres bibliothèques) ….

**JRE (Java Runtime Environment) :** ensemble d’outils nécessaire pour exécuter une application Java y compris la machine virtuelle, classes, et bibliothèque.

**Notion de Package :** l’organisation des classes sous java est gérée par une structure arborescente (répertoire les uns dans les autres), pour utiliser une certaine classe qui se trouve dans un certain package il faut l’importer en utilisant le nom complètement qualifié :

Exemple d’importation de la classe Vector en Java :

**import java.util.Vector ; 🡪 import la classe seulement.**

**import java.util.\* ; 🡪 import toutes les classes qui se trouvent dans le package java/util.**

**Convention de nommage :** pour faciliter la lecture du code en java, il faut respecter certaines règles lors d’écriture d’un programme.

Les packages sont entièrement en minuscule 🡪 javax.swing .

Les mots des classes doivent commencer par une lettre majuscule et les autres en minuscules 🡪 String, StringBuffer …

Les méthodes : comme les classes sauf que la 1ére lettre doit être en minuscule 🡪 append(), toString() …

Les mots des constantes (finale) doivent être en majuscule séparer par <<**\_**>> 🡪 PI, MIN\_VALUE ….

**Structure d’un programme en Java :**

public class NomClasse{

//propriétés et méthodes

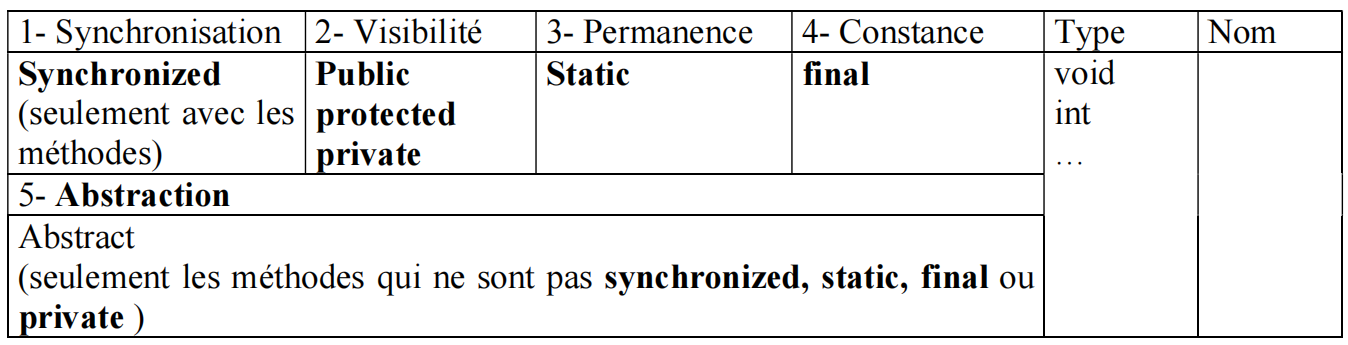
public static void main (String args[]){

//Code de la fonction principale

}

} la fonction principale est facultative.

**Modificateur et visibilité :**



**Mot-clé synchronized :** permet de verrouiller l’accès á une méthode par l’intermédiaire du mécanisme de monitor, elle sera donc << Thread-Safe>> 🡪 exclusion mutuelle :Permet de gérer les accès conçurent á une méthodes par des threads.

**Mot-clé private :** inaccessible depuis l’extérieur de la classe.

**Default package** **:** accessible seulement par les classes du même packages (les classes dérivées qui appartient á un autre package ne peuvent pas accéder au donner déclarer default package).

**Mot-clé protected :** accessible par les classes filles et les classe qui appartient au même package de la classe.

**Mot-clé public :** accessible à tout le monde.

**Mot-clé static :** permet de définir une méthode ou une propriété commune entre toutes les instances de la classe, dépend que de la classe elle-même, on peut y accéder sans instanciation par l’instruction suivante : className.staticMeth(). className.staticVar .

Les méthodes et les propriétés statique, sont appelées méthodes et propriétés de la classe, les méthodes statiques ne peuvent pas accéder qu’aux attributs et méthodes statique, par contre les méthodes simples peuvent accéder au méthodes et attributs statique ou bien non statique.

**Mot-clé final :** utiliser pour déclarer les constantes en Java, il doit toujours accompagner par public static puis final, car les constantes doivent être public (déjà on ne peut pas les modifier, prq les protéger donc ?), static pour dire commune entre toutes les instances de la classe (une constante doit être global pour tous les objets) sinon il va être dupliquer sur tout objet de la classe (gaspillage de mémoire).

Les variables finales doivent être initialiser immédiatement 🡪 public static int PI = 3.14 ;

Les méthodes finales veulent dire, ne peuvent pas être redéfinie dans les classes filles.

Le mot clé final pour une classe sert á interdire l’héritage de cette classe.

Un objet déclarer finale est un objet qu’on ne peut pas changer sa référence ça veut dire on ne peut pas lui affecter une nouvelle instance, mais on peut modifier son contenu.

**Mot-clé abstract :** permet de définir une méthode abstraite, Càd méthodes qui n’a pas de corps, une classe qui contient une méthode abstraite doit être aussi abstraite (on ne peut pas crée une instance de cette classe), et c’est aux classes fille de redéfinir les méthodes abstraites de la classe mère.

abstract Employe{

abstract public float calculeSalaire(){}

}

Une méthode abstraite ne peut pas être :

* Synchronized (car elle n’est pas encore définie)
* Static (Les membres static existe dès la création de la classe),
* Final (va empêcher la redéfinition de la méthode, alors qu’elle est faite que pour ça),
* Private (car les méthodes déclarer private ne peuvent pas être redéfinie, si non si on essaye de redéfinir une méthode privée, on va juste crée une méthode qui porte le même nom).

**Méthode main :** est une méthode statique qui ne peut pas accéder aux autres membres de la classe, pour résoudre ce problème, il faut crée une instance á l’intérieur de main, et de mettre l’appel aux autres membres á l’intérieur de constructeur de la classe.

public class ClassName{

var1 ;void methode1() ;

public ClassName(){

methode1() ;

}

public static void main(String[] args){

new ClassName() ;

}

}

**Objets en Java :** les objets en Java sont dynamiques et issues d’une classe Object qui est la mère de toutes les classes on Java, pour crée un objet d’une classe A en java 🡪 **A objet1 = new A()** ; , objet1 est un pointeur de type A, pointe sur l’instance qui était créé par l’opérateur new, pas besoin de libérer l’espace avec delete, car la libération se fait automatiquement si cette instance n’est plus référencié.

A objet2 = objet1 ; ici c’est une copie de référence, les 2 pointeurs 1, et 2 pointent sur la même instance, si on modifie l’un, l’autre sera modifier aussi, et pour provoquer la suppression d’une instance, il suffit d’affecter null au pointeur qu’elle la pointe.

Lorsqu’on compare 2 objets de la même classe á l’aide de l’opérateur ==, il s’agit alors d’une comparaison de référence (Vrai si les 2 pointeurs pointe sur la même instance), pour comparer le contenus, il faut redéfinir la méthode equals() de la classe Object.

**Opérateur instanceof :** permet de déterminer si un objet appartient á une certaine classe 🡪 (objet1 instanceof A) : utiliser souvent dans les conditions.

**Méthodes de la classe Object :**

1- equals() :compare le contenus des objets 🡪 besoin de la redéfinition.

2- toString() : utiliser dans System.out.println(objet1) [équivalente á dire System.out.println(objet1.toString) ] pour afficher le contenus de l’objet1 🡪 besoin de la redéfinition.

3- getClass().getName() : renvoie le nom de la classe de l’objet appelant.

4- getClass().getSuperClass().getName() : permet de renvoyer le nom de la classe mère d’un objet appelant.

5- clone() : permet de crée une copie de l’objet appelant.

6- finalize() : appeler par le Garbage collector pour éliminer un objet de la mémoire.

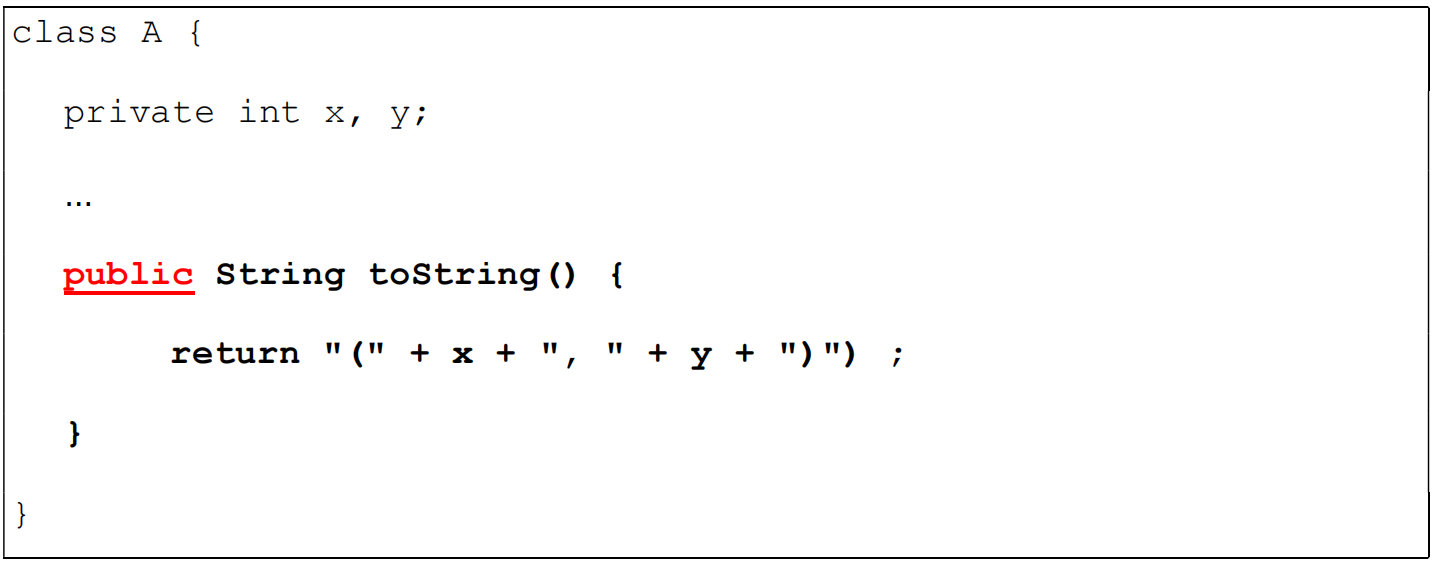
7- hashCode() : retourne un hash-code pour l’objet appelant.

8- notify() : appeler seulement dans un moniteur( méthode synchronized) pour réveiller un processus qui s’était fait bloquer.

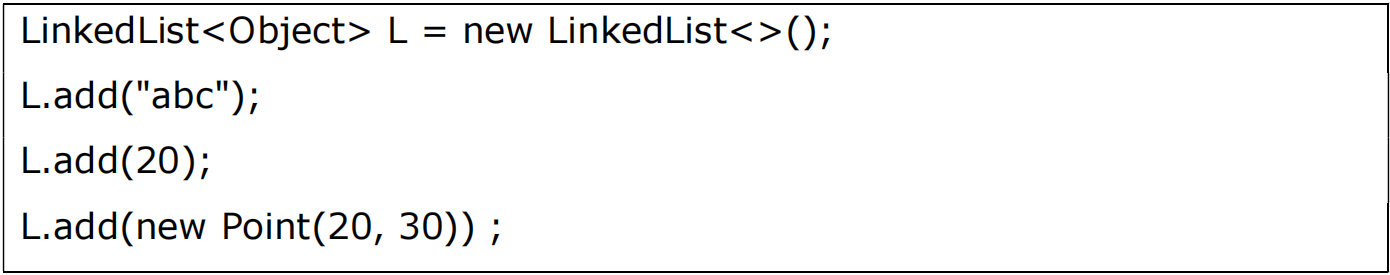
9- notifyAll() : réveille tout les processus bloqués dans une méthode synchronized.

10- wait() : utiliser pour bloquer le processus appelant, qui restera bloqué jusqu’à recevoir un signal notify() envoyé par un autre processus.

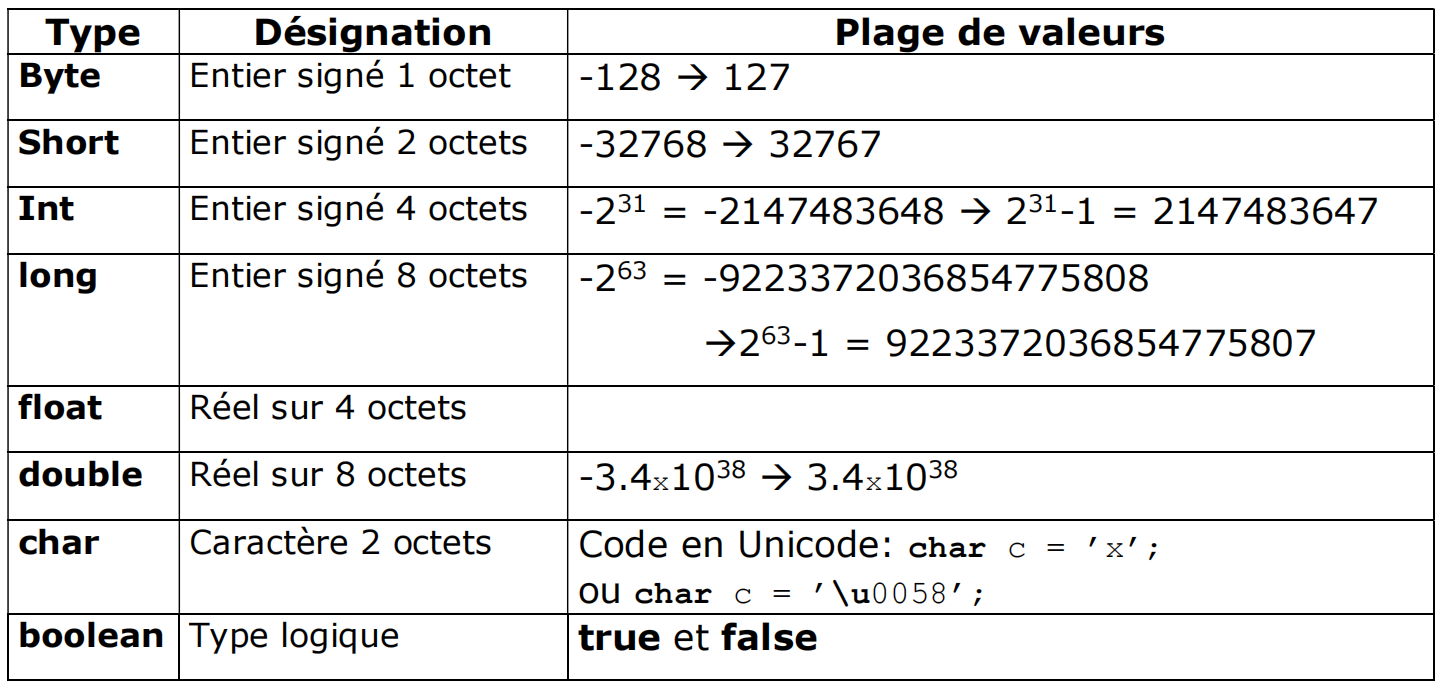
Example de la redéfinition de toString() :



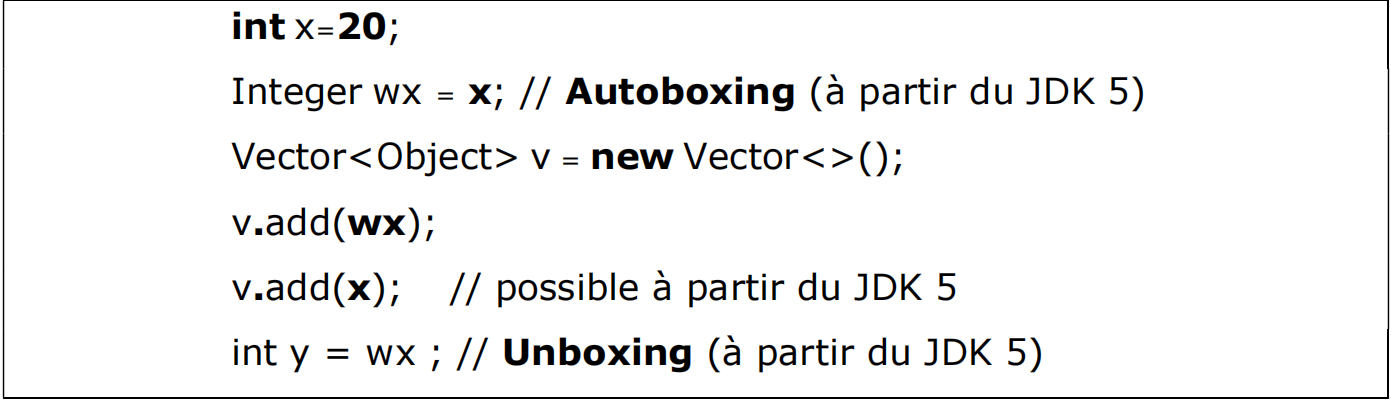
Si on a un Vector de type Object on peut lui fait affecter tous types d’objets, car toutes les classes descendent de la classes Object, ce qui fait que c’est une affectation mère <- fille qui est valable on java aussi.



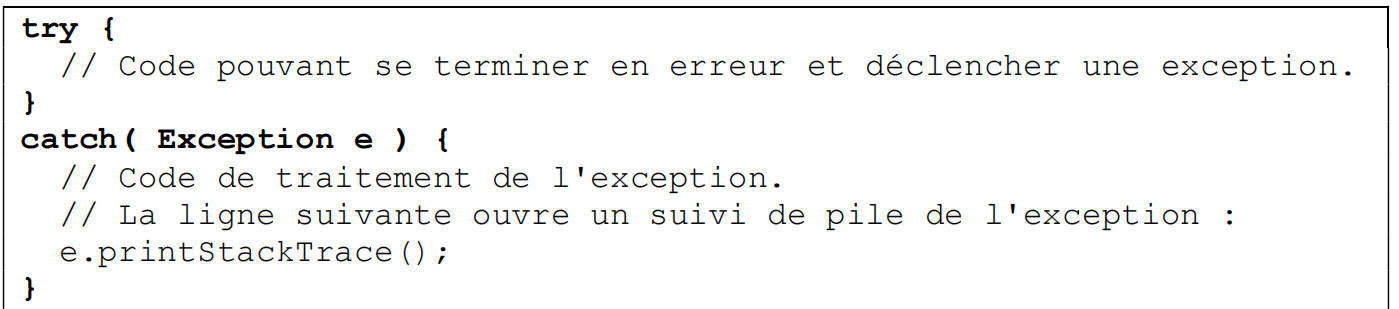
**Types primitifs**: Java offre 8 types de primitifs, les primitifs sont gérés en java statiquement.



**Les types wrappers :** ce sont des objets crée pour remplacer les types primitifs, tout simplement se sont utiliser pour permettre la conversion des types primitifs vers la classe Object ce qui n’était pas valable jusqu’au JDK 5.



**Gestion des exceptions en Java :**



Toute exception est une instance de la classe Exception qui se trouve dans le package java.lang.

**Héritage en Java :** une classe B herite d’une classe A se fait á l’aide du code suivant :

**Class B extends A {**

**}**

Si A dispos seulement un constructeur avec paramètres, ce qui fait que le compilateur ne génère pas un constructeur par défaut pour A, par conséquent l'écriture précédente n’est pas correcte, car le constructeur par défaut de B généré par le compilateur fait appel au constructeur par défaut de A qui n’existe pas, d’où l’erreur.

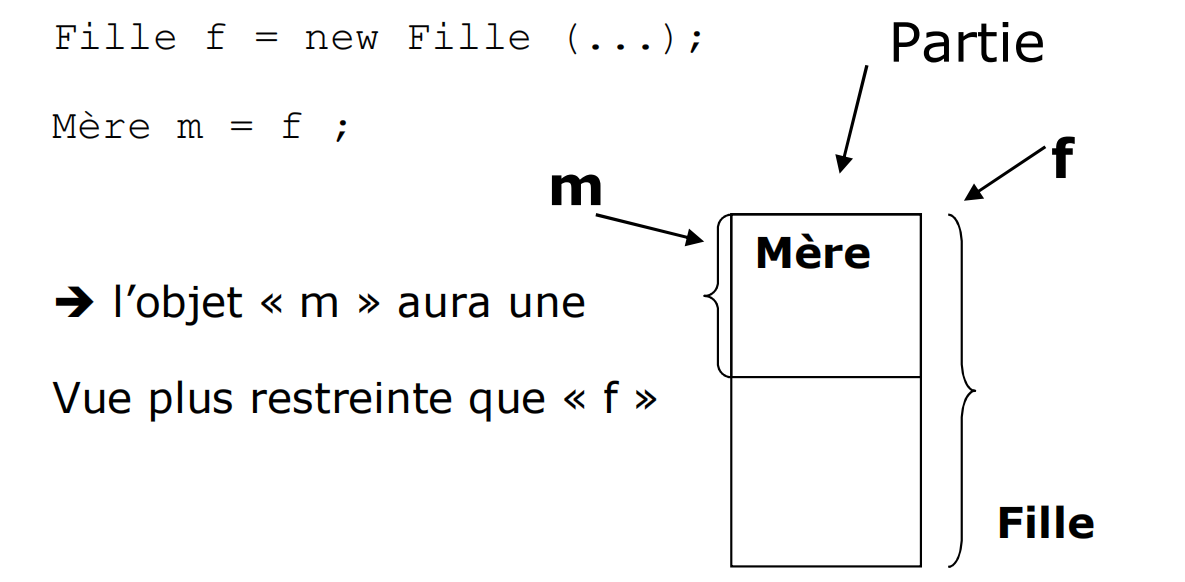
Pour dérouter l’appel á un autre constructeur mère, il faut l’appeler dans la première instruction du constructeur fille **super(**/\*paramètres\*/**) ;**

Utilisation de super appelle la classe mère immédiatement supérieur.

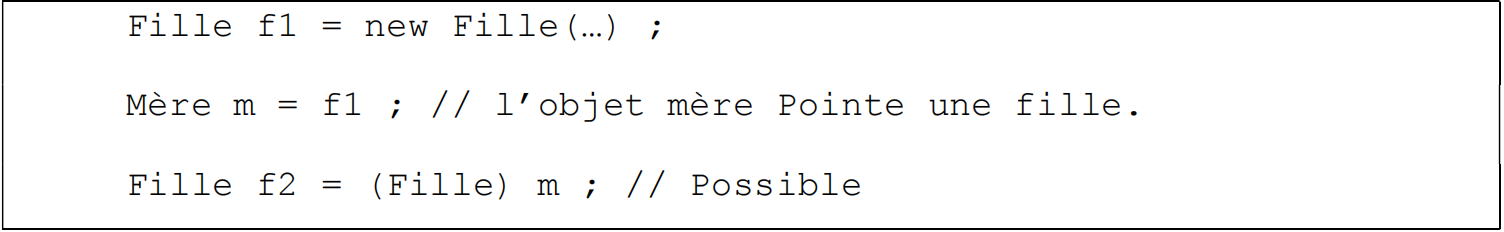
Un constructeur peut appeler un constructeur de sa même classe á l’aide de mot clé this (/\*paramètres\*/) ;

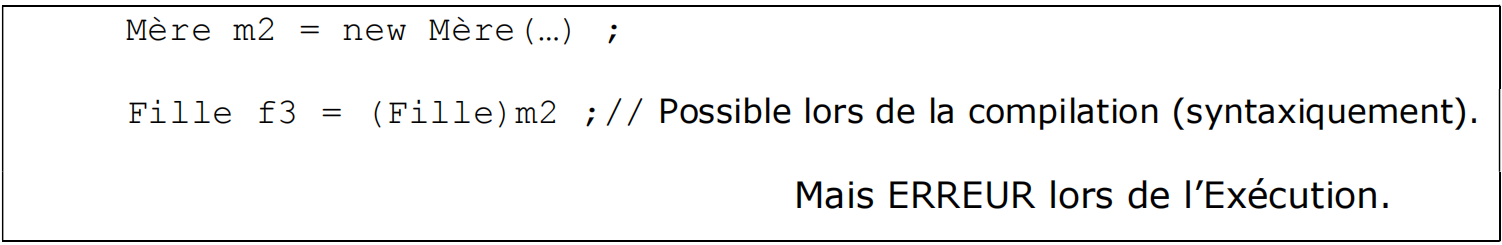
Pour appeler une méthode de la classe mère qui était redéfinie dans la classe fille, on utilise le code suivant : super.methode() ;

**L’affection entre objet :** affectation Mère 🡨 fille est possible ;

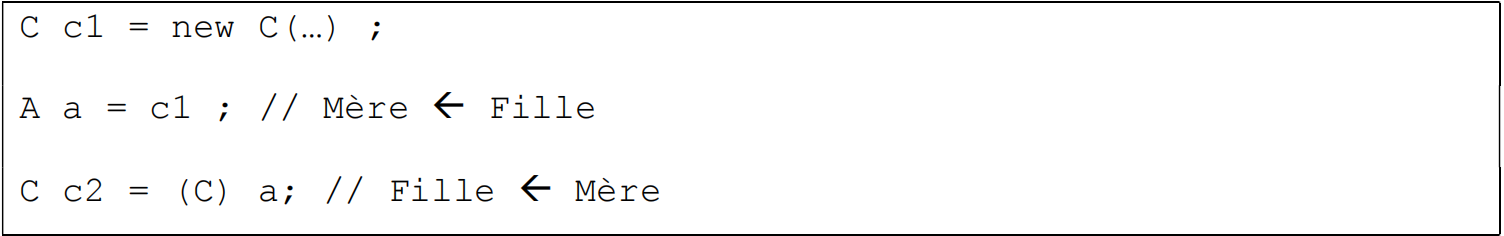


Affectation Fille 🡨 Mère n’est possible que : si on utilise le casting, et si effectivement le pointeur mère point sur une instance fille.



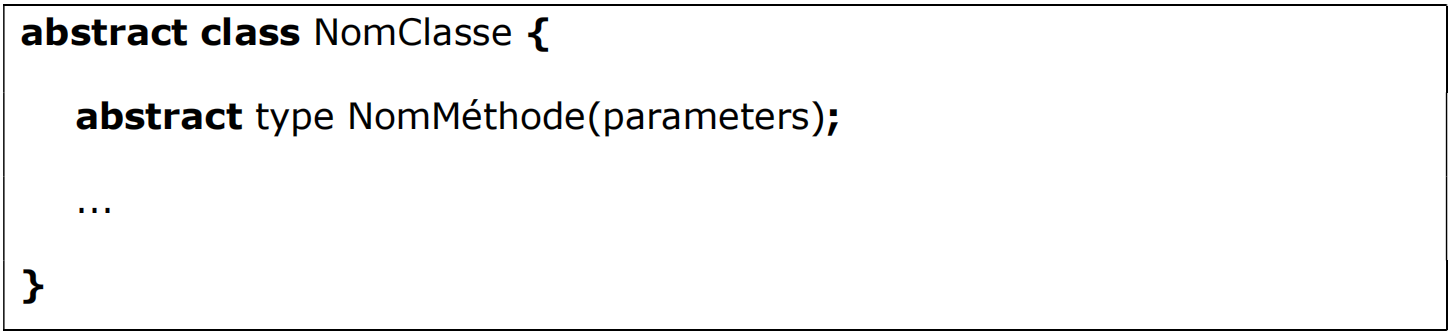


Si C étend de B et B étend de A, on peut faire :

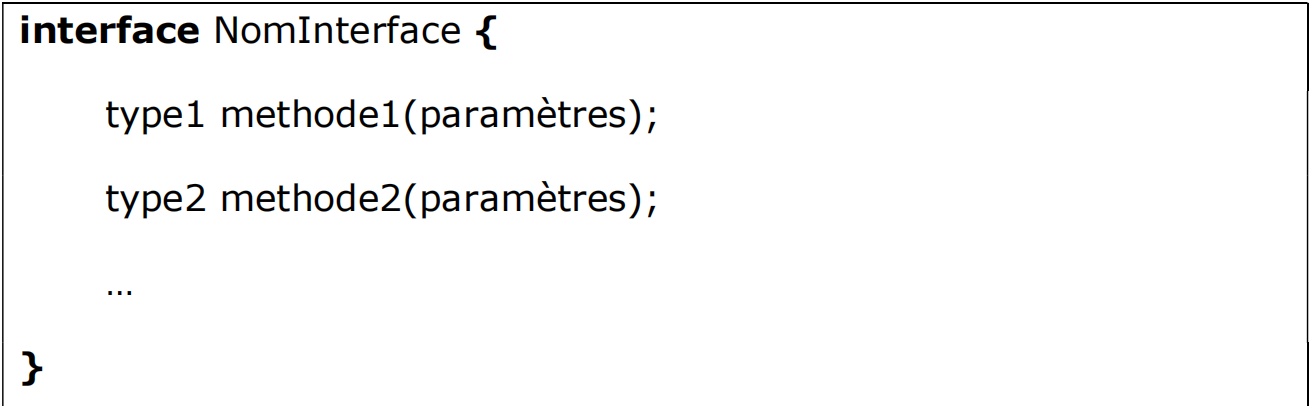


**Classes Abstraites et interfaces :**

Une classe abstraite est une classe qui ne peut pas avoir d’instance, car il contient au moins une méthode abstraite et dont le corps n’est pas encore défini, et c’est aux classes filles de redéfinir cette méthode, sinon si une classe fille n’a pas redéfinis cette méthode, alors elle aussi devra être abstraite.



L’interface est un cas particulier d’une classe abstraite, on peut dire que c’est une classe ou toutes les méthodes sont abstraites (contient que les prototypes des méthodes), c’est une structure totalement abstraite permettant la spécification et l’abstraction de service, l’héritage depuis une interface s’appelle l’implémentation et il se fait á l’aide du code suivant :





Une classe qui hérite d’une interface doit implémenter toutes ces méthodes, sinon elle sera aussi abstraite.

Toute propriété définie á l’intérieur de l’interface est implicitement public, static, final, ce qui nécessite alors une initialisation immédiate.

Une classe implémente une interface, mais une interface étend une autre interface.

Une classe peut implémenter plusieurs interfaces avec le code :

**public** **class** A **implements** IntA, IntB {}

classe A doit implémenter les méthodes de IntA et IntB.

Une interface peut étendre plusieurs interfaces, la classe qui l’implémente doit définir le corps des méthodes venant de l’interfaces et les méthodes venant du supers interfaces :

**interface** IntC **extends** IntA, IntB{}

On ne peut ni redéfinir une méthode statique, ni redéfinir une propriété, si on n’essaye de le faire il s’agit que du masquage, donc lorsqu’on utilise un objet mère sur une instance fille, les propriétés et les méthodes static de la classe mère qui seront invoqué.

**Les tableaux Java : Declaration d’un certain type des tableau en Java.**

int tab1[] = new int[10] ; String tab2[] = new String[64] ; Point tab3[] = new Point[20] ;

les tableaux herite aussi de la classe Object, et ils ont une attribut length qui indique la taille du tableau.

lorsqu’un tableau est crée il est impossible de le redimensioner, la solution est de crée un nouveau tableau avec la taille souhaitée, et d’y copier l’ancien tableau, puis referencer le pointeur ancien sur le nouveau tableau.

**Vector et LinkedList :** Vector se base sur la structure des tableaux, chaque élément est indexé, ce qui offre l’accès direct aux données du Vector, par contre le LinkedList est une liste chainée dont chaque élément possède un pointeur sur l’élément immédiatement suivant, l’accès est séquentiel n’est pas direct, mais l’ajoute des élément est très facile par-rapport á la structure des tableaux, Vector et LinkedList ont une mère commune nommée List qui est une interface abstraite qui contient les méthodes implémenter dans les 2 structures get(index),add(),remove() ….

Pour ajouter un Vector ou LinkedList en utilise la syntaxe suivante :

Vector<Object> v = new Vector</\*Object\*/>(/\*taille\*/) ; v.add(new Point), v.get(5), …..

LinkedList<Object> v = new LinkedList</\*Object\*/>(/\*taille\*/) ;

Quelque méthode des listes :

***boolean add(E élément)*** 🡪 permet d’ajouter un élément á la fin de la liste.

***boolean add(int indice, E o)*** 🡪 permet d’ajouter un élément á un indice donnée.

***boolean addAll(Collection<E> c)*** 🡪 permet d’ajouter une collection á la fin de la liste.

***void clear()*** 🡪 supprimer tous les éléments de la liste.

***boolean contains(Object o)*** 🡪 retourne true si l’élément existe dans la liste.

***void forEach(Consumer< ? super E> action)🡪*** appliquer l’action sur tous les éléments de la liste (action peut être une expressions lambda).

***E get(int indice) 🡪*** retourne l’élément á la position indice.

***int indexOf(Object o)*** 🡪 retourne la 1ère position de l’élément O de la liste.

***boolean isEmpty() 🡪*** retourne true si la liste ne contient aucun élément.

***Iterator iterator() 🡪*** retourne un Iterator pointant sur le début de la liste.

***boolean remove(int pos) 🡪*** supprime l’élément á la positon pos, si un élément passer au lieu de int on supprime son 1ère occurrence de la liste.

***E set(int indice, E element) 🡪*** insérer l’élément E á la position E.

***int size() 🡪*** retourne le nombres d’élément contenant dans une liste.

***void sort(Comparator< ? super E> c) 🡪*** trie la liste selon l’order spécifier dans le comparateur c.

***List<E> subList(int fromIndex, int toIndex)*** 🡪 retourne une sous liste entre les 2 positions spécifier dans les paramètres.

Collection.reverseOrder() ;

**Les chaines de caractère sous Java :**

Les chaines déclarer String sont immuable (reste inchangeable après leur création), par contre StringBuffer sont mutable, les 2 appartient á la même classe mère CharSequence.

**String :**

En effet lorsqu’on utilise += sur string, il s’agit de la création d’une nouvelle instance contenant les 2 chaine concaténées, car il est impossible de changer le contenu d’un string.

***Quelque Constructeurs de String :***

[**String**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#String(char[])) (char[] value),([**StringBuffer**](https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuffer.html) buffer),[**String**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#String(char[]))(String string).

***Quelques méthodes de la classe String :***

***char charAt(int index) 🡪*** retourne le caractère á la positon index.

***int compareTo(String s) 🡪*** retourne une valeur positive si la chaine appelant est plus grand lexicalement que la chaine s (sert pour le tri des chaines), 0 en cas d’égalité.

***String concat(String s) 🡪*** retourne la concaténation de la chaine appelante avec s.

***boolean contains(CharSequence s) 🡪*** permet de savoir est ce que la chaine appelante contient la chaine s ou pas.

***boolean endWith(String s) 🡪*** true au cas où chaine termine avec s.

***boolean startWith(String s) 🡪*** true au cas où la chaine commence avec s.

***boolean isEmpty(String s) 🡪*** retourne true si la chaine est vide.

***int length() 🡪*** return le nombre des caractères de la chaine.

***boolean matches(String regex) 🡪*** utiliser pour teste si la chaine respect une expression regulier.

***String replace(String old,String new)*** 🡪 remplace old sous-chaine avec new sous chaine.

***String[] split(String regex)*** 🡪 divise la chaine en sous chaine en se basant sur un séparateur regex, résultat un tableau des strings.

String ***substring(int beginIndex, int endIndex)*** 🡪 retourne la chaine existante entre beginIndex et endIndex.

***String toUpperCase() 🡪*** converti la chaine en majuscule.

***String toLowerCase() 🡪*** converti la chaine en minuscule.

***String trim() 🡪*** supprime les espaces au début et fin de la chaine.

***String String.valueOf([int, double ..])*** 🡪 retourne une chaine de caractères de nombre entré en paramètres.

***Classe StringBuffer*** :

*Quelques constructeurs :*

***StringBuffer(int taille) :*** crée une instance vide mais réserve une taille fixe pour la chaine.

***StringBuffer(String str), StringBuffer(CharSequence seq);***

***Quelques méthodes:***

***StringBuffer append([types primitifs, CharSequence]) 🡪*** ajouter le paramètre passé á la fin de la chaine.

***int capacity() 🡪*** retourne le nombre d’octet réserver pour la chaine.

***StringBuffer delete(int start, int end) 🡪*** supprime la sous chaine existante entre start et end.

***StringBuffer deleteCharAt(int index) 🡪*** supprime le caractere existant á l’indice index.

***StringBuffer insert(int offset, Object obj) 🡪*** insérer l’objet á l’indice offset.

***StringBuffer reverse() 🡪 renverse la chaine.***

***StringBuffer setCharAt(int index, char c) 🡪 modifier le caractère á l’indice index.***

***StringSequence(int start, int end) 🡪*** retourne la sous chaine entre start et end.

charAt, subString, lastIndexOf, length, et replace aussi valable avec StringBuffer.

**Collection Set :**

Il ressemble á l’interface List mais á condition qu’elle n’accepte pas des doublons.

Set<String> l = **new** HashSet<>();l.add("boussaid") 🡪 HashSet est une classe qui implémente l’interface Set.

HashSet : lors de récupération des éléments, l’ordre d’insertion n’est pas respecté, l’élément le plus proche au niveau de la mémoire sera affiché.

TreeSet : les éléments sont triés.

**Tableau associatif(map) :**

Map<String, String> phoneBook = new HashMap<>();

Map<String, String> phoneBook = **new** HashMap<>() 🡪 pour créer un tableau associatif.

phoneBook.put("Boussaid", "0624160055") 🡪 ajouter un éléments au tableau.

phoneBook.get("Boussaid") 🡪 récupérer la valeur de clés entrées

Map.Entry : Entry est une interface á l’intérieur de l’interface Map, contient 2 méthodes qui récupère clé et valeur d’un tableau associatif:

Set<Map.Entry<String,String>> values = phoneBook.entrySet();

**for**(Map.Entry<String, String> e : values) {

System.***out***.println(e.getKey() + " : " + e.getValue());

}

***Fichier en Java :***

**Écriture dans un fichier nécessite 2 objets :**

Objet de type ***FileOutputStream*** pour ouvrir le fichier, et un tampon de type ***DataOutputStream*** (ou bien ObjectOutputStream pour écriture des objets) qui est utiliser pour écrire dans l’objet pointant sur le fichier, une fois l’écriture est terminé il faut fermer les 2 objets par close().

Remarque : l’objet DataOutputStream on doit lui passer un objet de type FileOuputStream lors de sa création, et également ce dernier on doit lui passer l’emplacement du fichier.

* FileOutputStream fos = **new** FileOutputStream("tst");//permet d’ouvrir un fichier en écriture.
* DataOutputStream dos = **new** DataOutputStream(fos);//permet d’écrire les données primitives dans un fichier ouvert en écriture.
* ObjectOutputStream oos = **new** ObjectOutputStream(fos); );//permet l’ecriture des objets dans un fichier ouvert en écriture.

dos.writeUTF("toto is a good man \n done");//ecrit la chaine toto … dans le fichier.

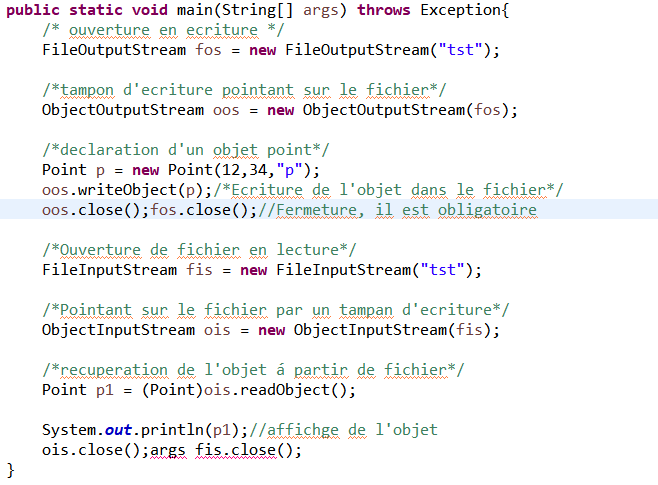
***Différents méthodes de DataOuputStream*** 🡪 writeUtf(), writeDouble, et writeXxx() avec xxx tous les types primimitifs.

***ObjectOutPutStream*** contient tous les méthodes de DataOutputStream en plus de writeObject() qui permet de stocker un objet dans le fichier.

**Lecture á partir d’un fichier :**

Presque les mêmes objets sauf qu’on remplace Output par input au nom des classes, et pour récupérer les données on utilise readUtf, readObject …

* FileInputStream fis = **new** FileInputStream("tst"); //fichier ouvert en lecture.
* DataInputStream dis = **new** DataInputStream(fis); //tampon utiliser pour récupérer les types primitifs depuis le fichier pointé par l’objet fis.
* ObjectInputStream ois = **new** ObjectInputStream(fis); //tampon utiliser pour récupérer les objets et tous types primitifs depuis le fichier pointé par



**Classe interne, locale et anonyme:**

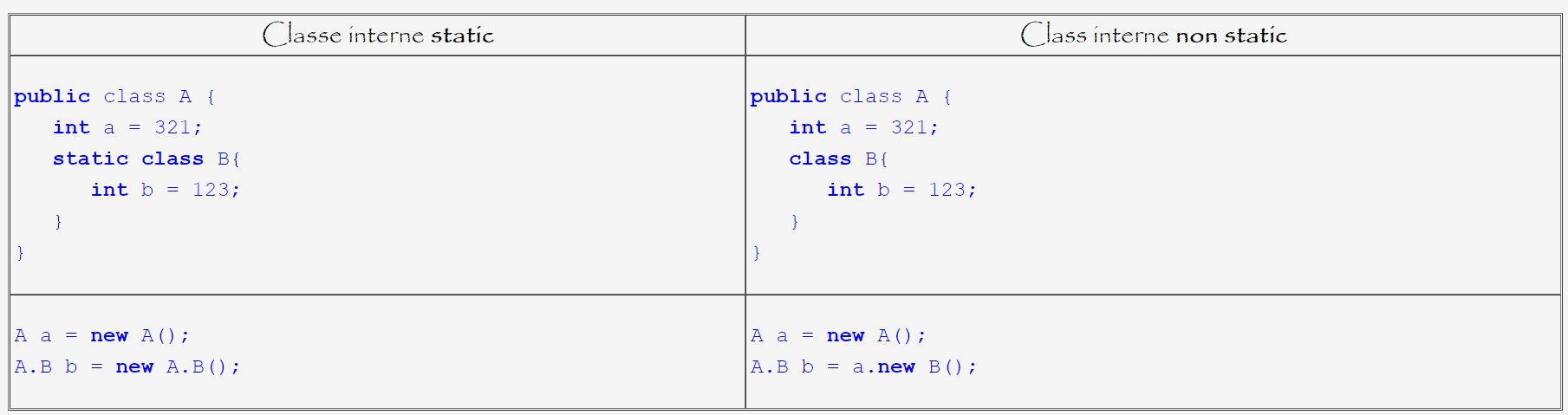
**Classe interne :** c’est une classe définit á l’intérieur d’une autre, la classe interne a la possibilité d’accéder au membres de la classe qui l’englobe, pour crée une instance de la classe interne il faut passer par une instance de la classe englobante :

Englobante.Interne i = **new** Englobante().**new** Interne();

Si la classe interne est déclarée **static** on peut alors créer une instance de la classe interne, sans nécessité d’instancier la classe englobante 🡪 Englobante.Interne i = **new** Englobante.Interne(); Dans ce cas, on ne peut pas évidemment accéder aux attributs et aux méthodes autres que *static*, de la classe englobante.

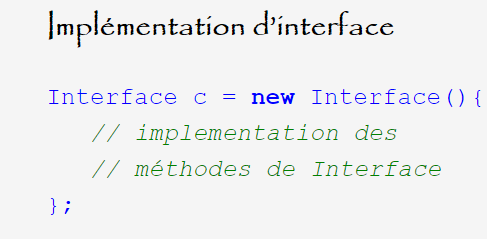
Lors de la compilation en aura 2 fichiers :

Externe.class et Externe$Interne.class



**Classe locale :** est une classe interne, mais qui est définit á l’intérieur d’une méthode, elle a l’accès au attribut de la classe englobante ainsi qu’aux paramètres et variables locale de la méthode ou elle est définie (á condition que ceux-ci soit final). Une classe interne locale ne peut pas être ni static.

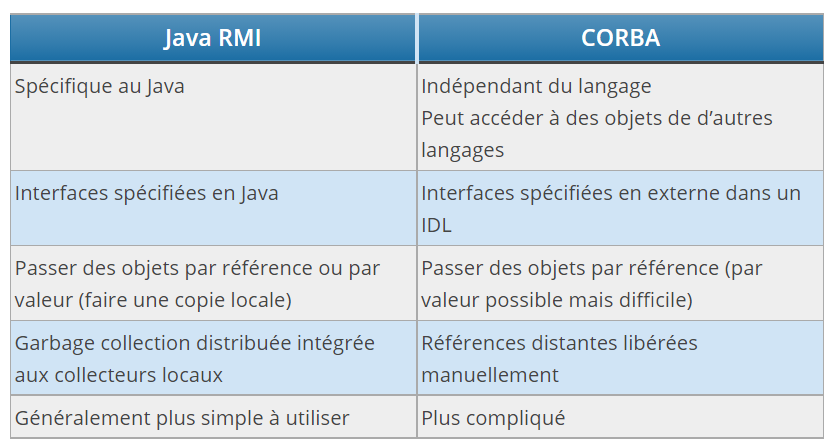
**Classe anonyme :** est une classe interne qui est définie sans lui donner un nom, utiliser souvent pour implémenter les méthodes d’une interface, il a également l’accès au attributs et au méthode de la classe englobante.



Lors de la compilation, le compilateur génère un fichier constitué du nom de la classe englobante puis un ‘$’, puis un nombre puis “.class”.

**RMI** et **CORBA**, les deux prennent en charge la programmation avec des objets distribués. Les deux permettent à un programmeur de traiter les objets distants comme s’elles sont des objets locaux. La différence principale entre les deux est la suivante :

**RMI** utilise que le langage Java, tandis que **CORBA** est indépendant du langage.



IDL (Interface Definition Language) : contenant des interfaces indépendantes des langages de programmation, peuvent être implémenter que ce soit par Java ou C++.

***RMI Remote Method Invocation:***

Prend en charge la programmation avec des objets distribué sous java, l’intérêt de RMI c’est de ne pas réinventer la roue si des classes sont déjà prédéfinit quelque part sur le réseau, il faut juste traiter les objets distants comme il sont des objets locaux.

Rmi se base sur Architecture Client(Stub) /Server(Skelton).

**Les étapes de configuration de RMI :**

Création de l’interface distant. L’interface doit étendre ***Remote*** qui est une interface propre á RMI.

Fournir l’implémentation de l’interface (création d’une classe qui implémente l’interface).

Compiler, créer stub and Skelton. Pour créer le skeleton il que la classe qui implémente l’interface aussi étendre d’une classe ***UnicastRemoteObject*** et pour gérer les exceptions il faut définir le constructeur de cette classe en jetant l’exception á la classe mère pour le gérer.

Lancer le registre. Permet d’identifier sur le serveur les objets concernant par RMI en leur donnant un identificateur que va être utiliser sur le client pour y accéder.

Créer et lancer un serveur. D’abord on instance l’objet ensuite on lui donne un identificateur avec Naming.rebind("ADD",obj),

Créer et lancer le client. Instancier l’interface avec la méthode Naming.lookup("ADD"),

L’interface doit être dans les 2 côtés, par contre son implémentation doit être que dans le serveur, et il faut être d’accord sur le nom de l’objet accessible.

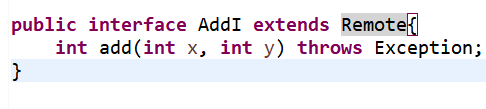
***# rmic nomInterface*** 🡪 commande de génération de stub et Skelton.

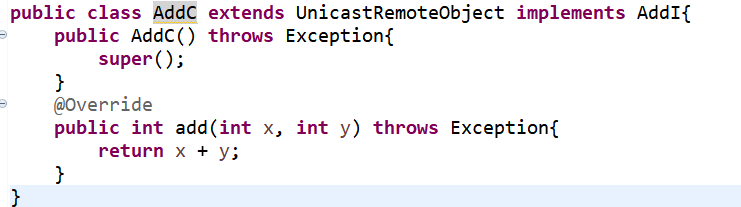
***# start rmiregistry*** 🡪 permet de lancer le registre.

***Les différents classes :***

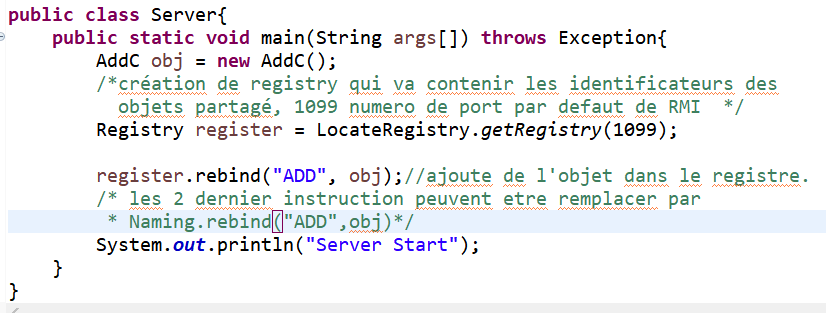
***Côte server :***

***L’interface :***



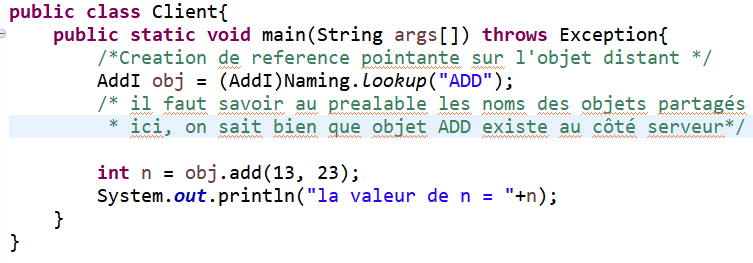
***L’implémentation de l’interface :***

***Classe serveur:***



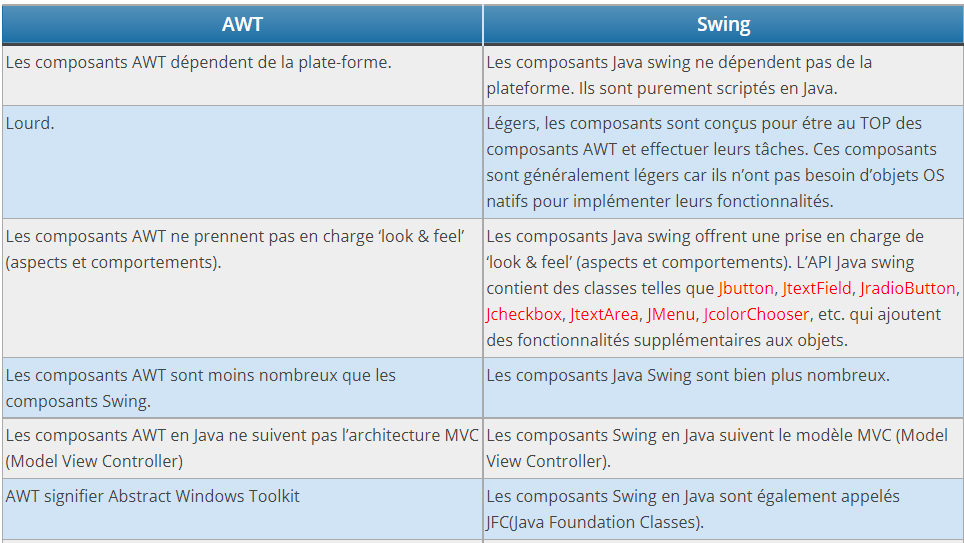
***Côté Client :*** il faut d’abord avoir une même interface que celle de serveur.

***Classe client :***



**Swing : interface graphique en Java**

**Swing** est une collection de composants qui ont la capacité de développer des objets d’interface graphique (GUI) indépendamment de la plate-forme, les composants **AWT** dépendent de la plate-forme et fonctionnent différemment sur différentes plates-formes.



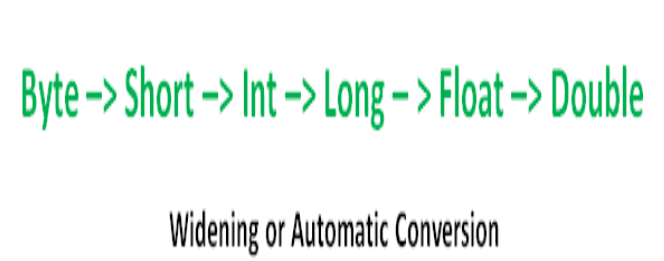
Remarque :

1- Tous les objets en Java sont dynamique (Notion de pointeur et d’instance).

2- la Machine virtuelle en Java dépend de la plate-forme sur laquelle elle tourne.

3-Java est basé sur la compilation séparer, la compilation d’un fichier .java qui contient 2 classes génère 2 fichiers.class.

Type primitifs en JAVA: si les type de données sont compatible, java effectuera automatiquement la conversion, sinon il faut utiliser le casting ou bien une conversion explicite.



De gauche á droit conversion automatique, de droit á gauche la troncation de donnée en utilisant le casting.

**double** x = 3; 🡪 POSSIBLE ,MAIS AVEC LES WRAPPERS NON;;

**float** x = 3; 🡪 POSSIBLE MAIS AVEC LES WRAPPERS NON;;

**long** l = 34 🡪 POSSIBLE MAIS AVEC LES WRAPPERS NON;

int est compatible avec double et float avec long, mais l’inverse est incorrect.

**float** x = 3.4;🡪 N’EST PAS POSSIBLE, CAR 3.4 EST CONSIDIRÉ COMME DOUBLE, IL FAUT UTILISER SOIT LE CASTING, SOIT 3.4F

Float est compatible avec double, mais double n’est pas compatible avec float.

**int** x = 'x';🡪 int et char sont compatible.

Int est compatible avec Long, mais l’inverse est incorrect (raison Long á 8octet, int 4 octet).

Les types wrappers : sont compatible qu’avec les types primitifs de même type.

Double x = 343.34 //Autoboxing ;

Double x = 343; // incorrecte 🡪 il faut utiliser Double x = 343d;

Long x = 34;// incorrect 🡪 il faut Long x = 34L;

Float f = 343.3f;//sans f cette affection est impossible.

Quelque fonction nécessaire en Java :

***int n = (int) (Math.random()\*10 + 1);*** 🡪 génère une valeur aléatoire entre 1 et 1 + 10 ;

***Double.parseDouble(“193,3”)*** 🡪 convertir une chaine vers un double.

***Integer.parseInt("24")*** 🡪 convertir une chaine vers un entier.

***Math.floor(double d)*** 🡪 partie entier.

***Math.pow(x,y) 🡪*** retourne x á la puissance y.

***Math.sqrt(x) 🡪*** retourne la racine de x.