JAVA

# Introduction et Définitions

**Jacky Melloul mellouljacky@hotmail.com**

**Plateforme/Framework :**

* Plateforme englobe framework (.Net vs JAVA) + langages + serveurs etc.
* .net: plateforme MS incluant le FK, plusieurs langages. Un peu une copie de JAVA, apparu après JAVA, 2002
* JAVA : plateforme oracle system, un seul langage JAVA, concurrent, 90’s
* GAP entre les deux = minuscule… On peut apprendre l’un et connaître l’autre.

**Environnement d’Exécution/de développement :**

* **Dev :** outil de dev SDK/JDK, Eclipse = IDE = env de dev. Permet de compiler.
  + **JDK :** Java Development Kit : compilateur + outils (debug…) + doc (source) + JRE (= env d’exec, pour .net = framework dotnet)
  + **Eclipse :** environnement visuel pour la progra et la compilation (au lieu de notepad + shell)
* **Exec : JRE** Java Runtime Environment à installer sur les machines au préalable. Valable pour le dev et pour l’utilisateur. = Programme chargé en mémoire (et éventuelle apparition à l’écran).
  + **JRE = JIT compiler** Just In Time **+ JVM (+GC)** Java Virtual Machine (CLR commun language runtime en dotnet) **+ JRE library** (ressource) fonctionalités

**Compilé/Semi-compilé/Interprété :**

* **Java, « révolution » :** solution à certains problèmes, comme en en C++ par ex.
* **Langage compilé :** Langage vers code machine. Pb : liaison trop étroite à l’OS/proc… Un lang. Compilé sur windows ne sera pas compris sur linux/mac. Fast, chargé direct du DD vers la RAM.
* **Langage semi-compilé :** Palier à ce problème de restriction d’OS. Ex : Transforme le lang .JAVA en un .class (pas du code machine, mais du byte code/code intermédiaire, msil (MS intermediate language) en dotnet .exe ou .dll, compilation partielle) via JAVAC.exe., plusieurs .class en .JAR. Puis vient l’exécution. Moins rapide que le compilé, car need une compil supplémentaire à l’exec.

**Exécution/CPU/RAM :**

* L’exécution = chargement du programme dans la mémoire.
* RAM = Heap + Managed Heap + Stack
* JIT compiler transforme .class en code final (machine) et le monte directement en mémoire. D’où le ralentissement de la montée de l’appli.
* Heap = C++, donc ne nous intéresse pas. La Stack se gère auto (FiFo), se pile et se dépile. Tout se charge dans le Heap managé (managé par JVM)
* Ensuite JVM = Gestion de la mémoire, allocation/déallocation = Process. Sous-process **GC** = Garbage Collector, ramasse-miettes, supprime tous les éléments inutiles dans la mémoire = vidage/swipe. On ne s’occupe de rien, la gestion est auto. Le GC supprime les objets sans références/non utilisés.

**Plateforme :**

* J2E appartient à la plateforme JAVA. Le langage est JAVA. La plateforme .net contient plusieurs langages VB.net, C#, J# (copie java maintenant supprimé car pb copyright avec Oracle), F#, C++ managé.
* Code portable.
* Tous les langages sont similaires à 99%, du moins on peut faire la même chose aussi performante. La multitude de langages du dotnet est seulement pour les vieux de la vieille habitués à leur code, et aussi regrouper les dev d’horizons différentes = fusion simplifiée vers MSIL (exe ou dll).
* + langages avec versions compatibles dotnet, comme python, pb…. Donc tout le monde peut coder pour un même projet et c’est la fête. Plus d’une trentaine de langage.

**Client lourd / Léger :**

* Client lourd : applications à installer sur le DD. 2 dépendances : .class/jar installé + JRE installé. Type de client lourd : appli à fenêtres, appli consoles.
* Client léger : type site web, besoin de rien à part le NAV et la connexion internet pour accéder à l’application.

**Choix d’une nouvelle technologie dotnet VS Java:**

* Choix historique et politique selon le chef et la boîte… pas trop réfléchi
* Choix économique : Si chaque environnement (BO-MO-FO) à ses propres projets, ont fait une équipe transverse. Elle monte des briques pour chaque environnement/équipe. Si la boîte fait déjà plus de JAVA, il est préférable de rester/continuer sur du Java…

**Eclipse et bases :**

* Workspace contient plusieurs projets.
* Un projet contient dossier src et bin (.class compilé à la volée par Eclipse).
* Src contient les dossiers package qui contiennent les codes source des class.java
* Les package nommés avec des points -> sous-dossiers ex pack1.pack2 = dossier pack2 dans dossier pack1
* New Java Project
* Project/src new Class -> cocher public static void main()
* Une class = nom du fichier et vice-versa. 1 class par fichier seulement !
* Main() = point d’entrée de l’application, méthode spécifique exécutée en premier, OBLIGATOIRE. Là où se déroule/démarre toute l’appli. UN SEUL main qui s’exécute dans un projet (.exe en dotnet, le reste .dll).
* Pour voir les sources JAVA il faut les rattacher : « attach source » dossier programmes/Java/JDK/src.zip
* Une class possède un nom court (dans le fichier lorsqu’on la créer) et un nom complet « nomPackage.nomClass » **Ce nom complet empêche les conflits de classes qui ont le même nom dans des packages différents. Package1.class1 != Package2.class1. Par contre il est impossible de faire un import des 2 dans notre fichier puisque même nom court utilisé par la suite. Il faut donc les utiliser, sans import, en spécifiant leur nom long.** Ex : Package1.class1()… directement
* On importe les class spé, ou directement les package nomPackage.\*
* Pour chercher où sont les packages d’une fonction spé non importée, click droit sur l’erreur, ou ctrl+espace (auto-completion au passage), ou ctrl+shift+o (inspecte tout le code et toutes les classes présentes et importe tout d’un coup).
* Le package java.lang est importé de base car un des plus utilisé, contient notamment le System.out.println()

# Fonctionnalités et progra de base

**Types / Primitives :**

* *c.f. PDF POO-1 p.24*
* String n’est pas une primitive mais une classe.
* Les primitives sont enveloppées dans des objets enveloppeurs (Wrappers). PDF p.29
* Val par défaut pour une variable locale : AUCUNE, il faut les initier avec un val

**Opérations / Tests / Transformations :**

* **Transformation / parsing :** int a = Integer.parseInt(b) ; transforme b en int. Ici **Integer est un wrapper**
* **Conversion / casting :** implicit a = b ou explicit a = (long)b
* **Opérations : + - \* / %** + fait aussi concaténation de strings
* **Incrémentation/décrémentation**: a++ a+= a-- a-= a\*= tout ça tout ça, p’tite diff avec –a ++a qui incrémente la var avant. En gros : b = a++ c’est comme b=a puis a=a+1, alors que b = ++a c’est b = a = a +1
* **Opérateurs de condition**: **== <= >= != > <,**  mais pour les string c’est un bordel différent puisque c’est pas une primitive.
* **Opérateurs conditions string**: string a et b : a.equals(b) -> true ou false, pour la casse : a.equalsIgnoreCase(b). Inverse d’une condition : ajouter **«** **! »**
* **Opérations ternaires : ((**condition)?expression\_si\_vrai:expression\_si\_faux). Exemple : (a>b)?a:b si a > b alors affiche a, sinon affiche b
* **Switch case :** Selon condition, transforme la valeur de la var en autre chose : Exemple :
  + switch(var) {

case val1 : instruction1 ;

Instruction 2 ;

Break ;

case val2 : instruction ;break ;

case val 3:

case val 4: instruction pour val 4 et 3; break;

default : instruction ;break ;

}

**Boucles :**

* **for** (int i=0 ; i < valMax ; i++) { instructions; }
* **while** (condition) {instructions;}
* **do** {instructions;} **while** (condition); Ici la condition est vérifiée après les instructions, donc on entrera au moins une fois dans la boucle, même avec une condition fausse.

**Contrôle/Sortir d’une boucle:**

* **Break :** sortie franche.
* **Continue :** saute l’itération en cours et passe à la suivante.

**Normes et syntaxes :**

* Une Classe commence par une Majuscule
* Une méthode (fonction) commence par une minuscule
* Une variable n’a pas de valeur par défaut dans une méthode (comme main()), car variable locale, quedal. Il faut l’initialiser avec quelque chose.
* Le main devrait être épuré, c’est-à-dire qu’il ne contient que des exécutions de méthodes, définies par la suite, après le main. En appuyant sur ctrl sur la méthode, on peut aller à sa définition

**Méthodes courantes :**

* System.out.print() ; affiche dans la console sans retour à la ligne
* System.out.println() ; affiche dans la console avec retour à la ligne
* String : variable char
* Scanner nom=new Scanner(System.in); : entrée utilisateur clavier, on l’enregistre avec String entree=nom.nextLine(); next prend premier string sans espace, nextLine prend toute la ligne.
* Déclaration de méthode : static void **methName**() { }

**Définition de méthodes :**

* Static **[type] methodName**(**[[typeArg] [Arg]]**) { }
* Le type = primitifs que doit retourner la méthode, ou « void » qui ne retourne rien.
* Les arguments à passer à la méthode à déclarer, comme int a, String b…

**Méthode swap et passage de paramètres :**

* Méthode swap : intervertie deux valeurs. Par contre ça ne marche pas ? Car le passage de paramètre par référence ne marche pas. Uniquement passage par valeur = par copie.
* C’est au niveau de la mémoire stack, dans une méthode les valeurs passées en argument sont des copies et non les ref originelles.
* Une solution est d’utiliser des tableaux ou objets. Cf. exos car ces éléments sont des adresses pointant à l’endroit des variables, donc on copie l’adresse qui pointe aux même variables dans la heap managé, et non une copie des variables dans la stack (qui disparait après la fin d’exécution de la méthode).

**Surcharge de méthode et varargs :**

* Méthode implémentée plusieurs fois (même nom) mais avec des paramètres d’entrée différents (signature), que ce soit en type, nombre ou ordre. Ce qui détermine alors la méthode est sa signature (ses paramètres), elle doit être **unique** pour chaque surcharge.
* Pour pouvoir passer en paramètre un nombre indéfini de type, on écrit « … » exemple : static void **methode**(int… x)  C’est un tableau à la volée / varargs ;
* Pour pouvoir également passer en argument n’importe quel type d’argument, on utilise le supertype / superclasse **Object** (qui inclus en fait tous les autres types = héritage). Exemple : static void **methode**(Object… **x**)

**Les Tableaux :**

* POO-1 p.116
* Liste d’éléments de même type
* Caractérisé par une taille et un type. Non modifiables par la suite.
* Déclaration :
  + int [] **Tab** = new int[**taille**]; ou,
  + int[] **Tab**; puis **Tab** = new int[**taille**];
* Val par défaut, cette fois vu que Tab est un objet, initiées par la JVM :
  + Booléen : false
  + Int, double,float :0[.0]
  + String : null
* Parcourir le tableau :
  + Boucle for, avec **Tab**.length
  + Boucle foreach, spé aux tableaux : for(int **el:tab)**
  + Indice Tab[**indice**]

**Récursivité :**

* Méthode s’appelant elle-même, au sein de la méthode…
* Cf Algo, fibonacci, factorielles.

# Classes et Objets

**Classes :**

* Entité qui définit un nouveau type = **déclaration**
* C’est un modèle, qui sert ensuite à créer des trucs… = **affectation**
* Déclaration : **Classe nom ;**
* Instanciation (créer en mémoire) **constructeur** : **nom =** new **Class**()**;**
* En une fois **Class nom =** new **Class**()**;**
* Affectation : **nom.attribut = … ;**
* Les classes peuvent contenir :
  + Attributs : data member, variable de class, chaque instance a ses propres attributs en mémoire. **nom.attribut ;**
  + Méthodes : Spé à la classe, appelée par le nom de sa référence : **nom.methode() ;**
  + Constructeurs : new **Class() ;** mise en mémoire + créer un objet + lien + val par défaut.

**Constructeurs :**

* Il en existe 4 types :
  + Par défaut de la JVM : Quand il n’y a aucun constructeur codé dans notre classe, c’est celui par défaut utilisé pour la création.
  + Par défaut : constructeur **sans paramètres**. Porte le même nom que la classe et ressemble à une méthode, sans void : **Class() { }**.
  + D’initialisation : On applique une val par défaut **Class() { attr = val ; }**. Il faut attribuer une val à **tous** les attributs.
  + Par copie : Clone, il faut le coder. Prend comme paramètre lui-même : **Class(Class var) {** this.**attr = var.attr }.** Prend donc une instance déjà créée pour en faire la copie : **Class nom2 =** new **Class(nom1) ;**
* Il est possible d’initialiser des valeurs par défaut dans l’initialisation des attributs, ou à l’aide d’une méthode, qui pourra alors être appelée plusieurs fois **nom.methode() ;**
* Pour clarifier le nom des attributs on utilise **this** pour faire référence au nom de l’attribut de la classe initialisée. **this.attribut**
* Egalement, chaque constructeur peut appeler un autre constructeur, une seule fois, en première ligne : **Class([Type] [args]){** this**([args])}**

**Contraintes de nommage & tricks :**

* Il existe certaine norme de nommage pour des fonctions qui effectuent le même type d’action sur une class, comme par exemple le println. Ce sont des classes **métiers**
* Pour pouvoir utiliser le println() sur notre class créée, il faut ajouter la méthode **public String toString().** Ensuite il suffit de printer l’objet.
* Eclipse possède des outils pour généré auto les constructeurs et le toString, en cliquant sur « Source » puis « Generate… »
* On peut créer une méthode create() pour créer auto un objet de notre classe…
* Les instances anonymes permettent de déclencher des méthodes « one shot ». Peut être utile pour déclencher plusieurs choses d’un coup via celle-ci, puis de ne plus l’avoir en mémoire par la suite car elle ne servira plus.

**Visibilité Private, Public, protected, default/package :**

* Public : accessible à l’extérieur de la classe depuis une instance. On peut la déclencher. Les méthodes principales et constructeurs sont généralement public.
* Private : **non** accessible à l’extérieur. Instance impossible à générer. Bloquer l’accès à certaines méthodes à l’utilisateur par exemple pour privilégier l’utilisation d’autres…. Les attributs sont généralement en private car ce sont les objets les plus importants et sensibles des classes, ils sont définis et non voués à modification propre. **Les attributs private peuvent être créés avec des valeurs que l’on passe dans le constructeur (car le constructeur est public, et dans la classe il voit tout). Mais pas modifiable NI PRINTABLE directement par l’utilisateur.**
* Protected : lié à l’héritage
* Default : lorsque l’on ne spécifie pas la visibilité, c’est la visib par défaut : package. Elle est public dans le même package, mais private dans un autre package. **A ne pas faire, il faut spécifier explicitement la visib.** Sinon conflit chez le client qui veut ajouter à son package.

**Methodes Static ou Non static) :**

* **Static =** lié à la classe. on peut l’appeler sans créer d’instance : **Class.methode1() ;** Le contenu static d’une classe est tout de suite chargée en mémoire par le **ClassLoader** (dans la heap managed) donc disponible. **Une méthode Static ne peut appeler directement une méthode non static, puisque pas dans une instance.**

Utilisées lorsque ces méthodes sont utilisées très fréquemment, elle sollicite alors qu’une fois le CPU (2 lors du vidage), et sollicite la RAM.

* **Non Static =** lié à l’objet / l’instance, il faut créer l’objet avant de pouvoir l’appeler : new **Class().méthode1() ;** Déclencheable à partir du nom de l’instance. **Une méthode non statique peut appeler directement une méthode static ou non.**

Utilisée lorsque ces méthodes sont appelées spécifiquement pour une instance, on sollicite donc le CPU à la création et suppression d’instance, mais peu la RAM.

**Accesseurs Getter & Setter :**

* Permet de modifier la visibilité d’éléments. Ce sont en fait des normes d’écriture de méthodes, publiques, que l’on code, permettant la modification ou la lecture d’attributs. set**Attr()** et get**Attr()**.

**Imbriquations / Associations**

* Les classes peuvent contenir des attributs de types simples, mais également de types complexes, ie d’autres entités classes, qui elles contiennent des attributs, classes etc.
* On peut donc factoriser des attributs qui nécessitent de contenir également plusieurs types d’informations. Exemple : Un salarié composé d’un nom, prénom et d’un RIB, et le RIB est composé d’un nom de banque et numéro de compte…
* Pour faciliter la visualisation de tout ce bordel on fait des diagrammes de classes (cf UML).
* Il existe deux types d’associations : Forte et Faible. Elles sont définies par le cahier des charges. Elles déterminent le lien entre deux classes, si on supprime l’une doit-on supprimé l’autre également ou non ?

Classe Salarié

* Nom
* prénom
* RIB
* constructeur
* Méthode1
* Méthode 2
* toString

Classe RIB

* Nom banque
* N° compte
* constructeur
* Méthode1
* toString

**Association forte et faible :**

* Forte : composition. On appelle un constructeur à l’intérieur d’un autre constructeur = Instance au sein d’une autre instance.
* Faible : aggrégation. On passe juste une ref, valeur, les deux instances sont distinctes dans la mémoire.

**Attributs Static ou Non Static :**

* **Static = Attribut partagé (shared)**: permet d’éviter la redondance en fixant l’attribut une seule fois. Même valeur définie pour tous les objets créés. Gain de place/mémoire ;  **Ce n’est pas une constance**, car modifiable si l’on veut.
* **Non static**: Spé à l’objet/instance, valeurs différentes.

**Block Static (ou constructeur static en dotnet) :**

* Instructions sur la zone static…..
* Effectuer des traitements…
* S’appelle tout seul comme un constructeur
* Contrairement au constructeur (qui peut faire que l’attribution), il peut appeler, modifier, traiter les attributs static.
* Static { insructions/attributs }

**Variables non modifiables « final »:**

* On ajoute « final » après le static void/type
* On peut du coup le rendre public, donc lecture seule
* Fixer une valeur, comme pi, constante unique non modifiable.

**Enumérés :**

* Public enum **nomEnum {**

**nom1(val1,val2),**

**nom2,**

**nom3(val3) ;**

attribs

methodes

**}**

* Liste d’option avec valeurs associées, ou non.
* Possibilité de récup nom, vals ou les deux
* Possibilité d’ajout d’atttributs, constructeurs etc… Logique de classes
* Le constructeur va charger en mémoire **chaque noms(vals)** = une instance pour chaque.

# Design Pattern

**Définition :**

* Solution à une problématique récurrente dans la programmation objet.
* Il existe un certain nombre de solutions objet préétablies à des problèmes rencontrés quasi systématiquement dans la programmation objet.
* Il existe plusieurs « types » de design pattern, solutions « toutes prêtes », comme le **GoF (Gang of Four) :** Liste des patterns les plus connus et plus fréquemment utilisés.
* Il existe des sites qui listent ou proposent des design patterns comme **dofactory.com**

**Singleton – Creation Pattern :**

* Permet de créer une seule instance d’une classe et de créer les objets à partir de celle-ci uniquement. Ex : spool d’impression qui reçoit les demandes des postes et les dispatch intelligemment entre les imprimantes disponibles.

**Observer - Behavioral Patterns :**

* Système d’abonnement et notification.
* A la disponibilité d’une variable ou autre, les abonnés peuvent la récupérer.
* Ex : un groupe d’élève (**observer**) attendant la note d’un prof (**observable**). C’est quand le prof donnera la note que les élèves seront notifiés/notés, et non les élèves qui solliciteront le prof jusqu’à ce qu’il est noté…

# JAR et travail de groupe

**Création :**

* Création du projet, avec package et classe (tester la classe)
* Génération du JAR : clique droit sur projet / export / Java / JAR File NEXT selectionner les classes à exporter

**Utilisation :**

* Création d’un projet avec le main
* Ajout du JAR : clic droit sur le projet / Propriétés / Java Build PAth / Libraries / Add External JAR OK
* Import du package dans notre fichier : import nomPackage.\* ;
* Utiliser la classe, w00t !

# Héritages

**Définitions :**

* Contrairement à l’association (composition ou agrégation), l’héritage s’effectue entre objets similaires. En fait si B **est une sorte de** A, alors B hérite de A. Du coup le contenu de A est dispo dans B directement.
* C’est le fait de **récupérer les données** du parent.
* A = classe de base ou classe mère
* B = classe dérivée ou classe fille. on écrit : class **B** extends **A { }**
* A 🡪 B : spécialisation/enrichissement
* B 🡪 A : généralisation
* C’est aussi **mettre en relation** des classes, qui deviennent sœurs via le parent (cf. classes abstraites)

**Visibilité :**

* La visibilité (public/private/protected) fixe ce qui va être hérité, ou plutôt l’accessibilité sera la même.
* **Protected** se comporte comme **public** pour l’héritage et au sein du package, mais **private** à l’extérieur du package (utilisé généralement pour les attributs)

**Redéfinitions :**

* Après l’héritage, on peut redéfinir les méthodes ou attributs dans l’héritier, qui écrasera alors celles du donnateur.
* Si l’on veut récupérer une méthode du parent, on utilise **super.methodeName().** On peut ensuite y ajouter des choses à son return, super.**methName + « lol »**

**Constructeurs :**

* Egalement pour appeler le constructeur du parent : **super([params])** ;
* Cela permet de modifier des attributs **au niveau du parent** même si non accessibles…
* **Si** le constructeur (avec ou sans paramètres) du père n’est **pas appelé explicitement** dans le fils, **le constructeur du fils** **appelle implicitement** le constructeur **implicite (non def) ou par defaut()** du père.
* En mémoire on a donc un objet Y contenant un objet X, systématiquement.
* **Donc il faut impérativement déclarer un constructeur par defaut() dans le père, c’est mieux surtout si on définit des constructeurs avec paramètres** (qui eux ne seront pas appelés implicitement même si présents)

**Instances « Hybrides » (pas le nom correcte) :**

* On peut créer un objet fils de type de son parent, ce qui restreint les fonctionnalités à celles du parent :
  + **P obj1 = new F([params]) ; ou F obj2 = new F(); puis P obj1 = obj2;**
  + **Cast implicite**
  + Cela créer un objet F contenant P, mais les fonctionnalités de F ne seront pas accessibles (masquées)
  + Il faudra recaster l’objet pour recouvrir les fonctionnalités : **F obj3 =** **(F) obj1**

**Polymorphisme :**

* On parle de **polymorphisme** dans l’héritage ou interfaces puisque **obj1** qui est de type F, est aussi du type de son P (et de ses P), obj1 utilise les methodes de F et P et les autres P.
* On peut donc dans un tableau de type P regrouper les objets de types F (et leur fils), mais cela donnera un **cast implicite**

**Multiples Héritages :**

* Le **multiple héritage**, qui consiste à hériter de plusieurs classes, est impossible an Java. Le risque est de retrouver deux méthodes de deux classes différentes avec le même nom = bordel. La solution est d’utiliser les **interfaces**.
* Toutes les classes héritent explicitement ou implicitement de la classe Objet, par défaut et définition.
* Les classes peuvent être **scellées** pour bloquer l’héritage de celle-ci, avec **final**, on ne pourra pas la redéfinir.

# Classes Abstraites

* On peut déclarer une variable de classe **Object**, qui est la classe englobant tous les autres types (qui héritent donc d’object), donc globale = fourre-tout
* Une classe **abstract** **class A { }** ne peut plus être instanciée, on ne peut plus manipuler son contenu avec var.meth() ou var.attr… On peut toujours utiliser les méthodes static, qui sont chargées direct en mémoire, pas en instance, type A.meth()
* Une classe est automatiquement abstract si une de ses méthodes est abstract
* Une méthode abstract est simplement déclarée : **public abstract void Methode() ;**
* Si on hérite d’une classe abstraite, on est obligé de définir/d’implémenter les méthodes abstract que possède le parent (et ses parents) pour que cette classe soit de nouveau normale, sinon on ne peut rien utiliser (sauf static) car instances bloquées.
* Cela oblige l’utilisateur à implémenter sa/ses propre(s) méthode(s) 🡪 force un modèle cohérent qui suit celui de la classe abstraite.
* On peut créer un tableau de type A et y mettre ses fils pour déclencher leur méthodes redéfinies différemment…
* On peut également utiliser les classes abstraites juste de façon **« générique »** afin de mettre en commun des classes différentes, via **héritage**, qui n’a ni sens concret, ni besoin de donner d’infos, mais juste de faire le **« pivot »** entre les classes filles (qui deviennent sœurs).

# Interfaces

* Entité qui définit un nouveau type
* **public interface I1 { }**
* Une classe peut ensuite implémenter l’interface : **class A implements I1 { }**
* C’est un peu un contrat entre une classe et une interface : la classe doit coder tout ce qui est présent dans l’interface. L’interface ne possède que des déclarations de méthodes. La class est donc également de type de l’interface.
* Utilisée quand la classe **adopte le comportement de**  l’interface
* En mémoire = zones au sein de la classe correspondant aux interfaces
* La création d’une classe de type de son interface 🡪 masque le contenu de la classe, mais pas celui de l’interface : **I1 obj1 = (I1) new A() ;** En fait pour être propre et être sure que A est un type I1, il faut tester avant le cast :
  + **If (new A() instanceof I1) I1 obj1 = (I1) new A() ;**
* C’est un peu la solution pour le **multiple héritage :**
  + Une classe peut implémenter plusieurs interfaces : **class A implements I1, I2,.. { }**
  + Problème quand dans les 2 interfaces on a le même nom d’une méthode et les mêmes paramètres… Solution : **Pattern Façade** qui consiste à créer des fonctions qui appellent d’autres fonctions d’autres classes…………….
  + Une interface peut hériter d’une autre, l’une étend l’autre.
  + Une interface **peut hériter de plusieurs interfaces** : **extends I1, I2,… { }**
  + On peut aussi créer des **pivots** entre différentes classes, qui ne contiennent pas forcément de contenu ou d’info utilisées.

# Interfaces génériques, Tris et comparaisons

**Généricité :**

* Certaines interfaces génériques acceptent tout type d’objet : interface **I1<T>**
* Ou sont de type dictionnaires (clef, valeur) **I1<K, V>**

**Sort :**

* La méthode **Arrays.sort(obj)** permet de trier un tableau de n’importe quoi, **à condition** que les objets du tableau contiennent la méthode **public int** **compareTo(Obj),** qui doit retourner un int de type -1 plus petit, 0 egal, 1 plus grand **ET** la classe doit implémenter l’interface **Comparable<typeClass> puisqu’elle contient la méthode compareTo(T o)**
* Si on reçoit un package non modifiable (jar) dont la méthode compareTo n’est pas implémentée, on peut créer notre propre classe ClassCompare pour le faire. Il faut importer **java.util.Comparator**, implementer **Comparator<typeObj>** et coder la méthode **compare()** en gettant les attributs. Ensuite à l’appel du sort on fait : **Arrays.sort(obj, new ClassCompare) ;**

**Comparaisons :**

* La comparaison == entre deux string compare la **référence** de l’objet, pas le contenu.
* La méthode **equals()** entre deux string compare d’abord la ref, si la même forcément même contenu, sinon compare le **contenu** (la chaine de caractère)
* Pour coder la méthode equals(), il faut aussi coder la méthode **hashCode()**, qui est une méthode de génération d’identifiant unique d’un objet. Ces deux méthodes proviennent de la classe **Object**

# Collections Dynamiques

## ArrayList

* C’est un peu un tableau dynamique d’objets, une liste **indexée** sans taille fixe, qui s’adapte au contenu.
* Elle peut accepter n’importe quel objet **ArrayList obj = new ArrayList() ;** ou on peut fixer le type d’objet **ArrayList<type>**
* En fait au niveau de la mémoire ce n’est pas un vrai tableau dynamique. C’est un tableau de taille initiale, qui lorsque son contenu atteint la capacité max, la JVM créer un nouveau tableau de **taille double** puis copie le contenu de l’ancien, et ainsi de suite.
* Il faut donc fixer la taille initiale de l’arraylist de sorte à ce que la JVM ne boss pas trop fréquemment (min CPU) mais on peut perdre en RAM.
* **.add(obj)**  pour ajouter un truc
* Le .**size()** de l’arraylist retourne le nombre concret d’élement. On ne peut pas connaitre la capacité du tableau (à part à l’initialisation).
* Le .**clear()**  supprime le contenu/vide le ta0bleau.
* Le **.get(index)** retourne le contenu à l’indice index.
* Le **.remove(index)** pour supprimer (null) un element.
* **.toArray()** permet de transformer l’arraylist en array simple, taille et type fixe.
* **.sort(comparateur)** trie l’arraylist à l’aide d’un comparateur
* **Un foreach** d’une arraylist n’a pas besoin de la vérification d’élement null, car elle ne bouclera que sur les élements présents.

## Liste liées – LinkList – File d’attente

* **LinkList<Type> nom = new LinkList<Obj>();**
* Principe **FiFo – First in First out**
* **.peek()** premier arrive
* **.poll()** enlève premier arrive
* **.poll()** enlève deuxième
* **isEmpty()**  voir si la file est vide

## HashMap – Dictionnaires

* java.util.HashMap;
* **HashMap<Type, Type> nom = new HashMap<Type, Type>**
* Forme Key, Value
* Interdiction de doubloon sur les clefs
* **.put(key,val)** ajoute
* **.get(key)** recup la valeur à la clef demandée
* **.keySet()** recup toutes les clefs
* **.values()** recup toutes les valeurs
* Si la clef est un objet on peut spécifier les attributs d’objet à tester pour correspondance, il faut coder les fonctions **public boolean equals(Object X) qui teste la clef, et public int hashCode() pour hacher les parametres de la clef**