Java théorie oral

**(Array)List :**

C’est une classe qui implémente *List*. Une arrayList n’a pas de taille limite et accepte n'importe quel type de données, y compris null. Elles sont rapides en lecture, même avec un gros volume d'objets.

Une collection représente un groupe d’objets, ses éléments.

* Tous les éléments sont de même type
* Certaines collections sont triées et d’autres ordonnées
* Certaines collections permettent des doublons

De nombreuses méthodes sont fournies avec cet objet :

|  |
| --- |
| Exemple : |
| List <String> fruits = **new** ArrayList<> ();  fruits.add("Banana") ;  fruits.add("Apple") ;  system.out.println(fuits) ; //affiche [« Banana »,  « Apple »] |
| Fruits.add(1, “Kiwi”) ; //ajoute à la position d’index 1 le mot « Kiwi » |

* *Add() :* ajoute un élément
* *Get (int index) :* retourne l'élément à l'indice demandé

|  |
| --- |
| Exemple : |
| public static void display (ArrayList <String> fruits) {  for (int i=0; i <fruits. size (); i++) {  system.out.println ( i + ": " + fruits.get( i ) ) ;  }  } |

* *Remove(int index)* : efface l'entrée à l'indice demandé
* *IsEmpty() :* renvoie « vrai » si l'objet est vide
* *RemoveAll() :* efface tout le contenu de l'objet
* *Contains(Object element) :* retourne « vrai » si l'élément passé en paramètre est dans l'ArrayList

Si c’est pour effectuer beaucoup de lectures sans se soucier de l'ordre des éléments, c’est judicieux d’optez pour une ArrayList.

Puisqu’une liste est itérable, elle peut être parcourue par un foreach :

|  |
| --- |
| Exemple : |
| for ( String mot : dictionnaire ){  system.out.println (mot); //mot prend chaque valeur de dictionnaire  } |

🡪 Comme la position de **mot** est inconnue, on ne peut pas supprimer ni remplacer un élement de l’arrayList.

Implémentation de l’interface List . L’arrayList a pour particularité d’être en fait un tableau. Au début, la taille de ce tableau est de 10 et il ne contient rien. Ensuite quand on souhaite ajouter un élément, on crée un nouveau tableau d’une taille qui est double par rapport à la taille de l’ancien tableau, on met les éléments de l’ancien tableau dans le nouveau tableau et finalement on rajoute le dernier élément.

Par conséquent, plus on va ajouter d’éléments plus la taille du tableau de notre List sera grande. Si on ajoute un élément à une arrayList dont la taille du tableau est 5000 et que toutes les 5000 cases du tableau sont occupées, on va passer d’un tableau de taille 5000 à un tableau de taille 10000. Ce qui peut être une grosse perte de mémoire.

D’un autre côté supprimer et refaire un tableau à chaque fois que le tableau est plein est coûteux en écriture.

**Assignation :**

L’assignation est avant tout une expression avec un type (celui de la variable) et une valeur (celle du *left hand side*). On en fait une instruction avec le ;

|  |
| --- |
| Assignment :  LeftHandSide AssignmentOperator Expression  LeftHandSide :  Identifier  ArrayAccess |

Il existe d’autres opérateurs d’assignation :

|  |
| --- |
| Assignment :  LeftHandSide AssignmentOperator Expression  AssignmentOperator : one of  = \*= /= %= += -= |

|  |
| --- |
| Exemple : |
| i+=1 ; //équivaut à i = i + 1 |

Il existe deux sortes d’assignation :

* assignation simple ( *=* )
* assignation composé (%=)

L’assignation est une expression (qui comprend une opération, ex : + - \* / …).//encore définir expression.

En java, on l’utilise avec un opérateur d’assignation pour attribuer une valeur à une variable.

Elle a un type (celui de la variable) et une valeur (celle du left hand side) . On considère la variable de gauche comme étant le premier nombre de l'opération mais aussi la variable dans laquelle il faudra mettre le résultat.

|  |
| --- |
| Exemple : |
| int a = 1;  int b = 2;  int c = 3;  a = b = c; |

L’assignation s’opère de droite à gauche. Donc : a = (b=c) Comme c = 3, b = c donc b = 3 et

a = b donc a = 3.

DONC le résultat du premier opérande de l’assignement doit-être une variable, sinon cela ne compilera pas ( a = 2 = c, ne compile pas). Si une des variables n’est pas déclaré, ça ne compile pas, pareil pour une constante qui ne se trouve pas tout à fait à droite.

DONC l’assignation est avant tout une expression :

* elle a un type (celui de la variable)
* elle a une valeur (celle lefthandside)

Comme il est possible de faire en une ligne plusieurs déclaration (séparée par une virgule), il est aussi possible de faire plusieurs assignations en une ligne ;

**Break/continue :**

Tout d’abord, il faut introduire la notion de label. Toutes les instructions peuvent recevoir une étiquette (label). Il permet de nommer une instruction et n’est connue que dans l’instruction qui la suit. Le label permettra de quitter brutalement (break) ou de réitérer (continue) certaines instructions.

|  |
| --- |
| LabeledStatement :  Identifier **:** Statement |

Le break permet d’arrêter brutalement une instruction. Si il a une étiquette, il arrête l’instruction étiquetée. Sinon, il arrête la boucle englobante.

|  |
| --- |
| BreakStatement :  break Identifier(opt) ; |

* Il est associé à une structure conditionnelle (boucle for, while), sans laquelle la boucle ne ferait jamais plus d'un tour.
* L'identifier est optionnel car si on ne le met pas, il arrête l'instruction en cours. Si on met l'identifier il arrête le label qui est trouvé en remontant le code, cette utilisation peut être intéressante lorsqu'on veut arrêter des boucles imbriquées.

Le continue permet de passer directement à l’itération suivante. Si il n’as pas d’étiquette, recommence la première instruction répétitive englobante. Sinon, il recommence la boucle étiquetée.

|  |
| --- |
| ContinueStatement :  continue Identifier(opt) ; |

Le continue est utilisé dans le cas d'une boucle (for et while) et d'un arrêt inconditionnel.

S’il n'y a pas de label il réitère l'instruction en cours, s’il y'en a un, il réitérer l'instruction étiquetée par le label. Continue est un saut inconditionnel associé à une structure conditionnelle. Il permet de passer aux instructions suivantes sans mettre fin à la boucle, ou la suite d'instruction.

**Collections :**

Collection est une Interface qui représente une collection.

Une collection représente un ensemble d’objets (ceux-ci seront ses éléments).

* tous les éléments d’une collection sont de même type
* certaines collection sont triées et/ou ordonnées
* certaines permettent les doublons

Comme une collection n’admet que des objets, il est impossible d’y placer directement des

types primitifs. Pour baiser le système, on va utiliser des wrappers .

Attention Collection != Collections !!!

Les méthodes de cette classe lancent une NullPointerException si les collections ou objet (!!!) (class objects) sont null.

Cette classe possède aussi des algorithmes destructifs, ces algorithmes sont des algorithmes qui modifient la collection qu’ils reçoivent (exemple tri), ils lanceront une

UnsupportedOperationException si la collection ne supporte pas l’opération, la méthode peut (mais le ne fera pas toujours) lancer cette exception si l’opération ne changera rien à la collection.

Cette class fait partie de Java Collections Framework.

Package (à importer): java.util.Collections;

Les collections n'ont pas de taille prédéfinie. Il est donc impossible de dépasser leur capacité!

Les méthodes de collections proposent des service pour des collection (comme les listes).

java . util . Collections propose des services pour les listes :

* max (List l ) donne le maximum d’une liste
* sort ( List l ) trie une liste
* reverse ( List l ) inverse une liste
* shuffle ( List l ) mélange une liste

**Constructeur :**

Un constructeur en Java est une **méthode spéciale** qui a pour but d’instancier (construire en mémoire) des objets :

* lui réserver de l’espace en mémoire et donc, par extension, d'en réserver pour toutes ses variables.
* initialiser son état (ses attributs)

Pour instancier un objet, il faudrait faire appel au constructeur via la combinaison de

mots-clé : new NomClasseObjet(param).

Le constructeur aura pour but de définir l’état de l’objet via ses attributs. Il s’occupera donc

de définir la valeur de ces attributs.

Un constructeur se définit comme une méthode standard, mais ne renvoie aucune valeur.

Ainsi, le constructeur d'un objet porte le même nom que la classe et ne possède aucune

valeur de retour (même pas void).

* un constructeur porte le même nom que la classe dans laquelle il est défini.
* un constructeur n'a pas de type de retour (même pas void).
* un constructeur peut avoir des arguments.

Les attributs ont une valeur par défaut :

* 0 pour les nombres ;
* null pour les références ;
* false pour les booléens.

Il existe un constructeur par défaut sans paramètre et qui ne fait rien.

*Remarque :* Il n’est plus disponible si un autre constructeur est fourni.

On peut fournir plusieurs constructeurs :

* On appelle cela la surcharge (overloading) ;
* Doivent se différencier par : nombre ou type des paramètres.

On peut placer des “sécurités” dans un constructeur pour le rendre robuste.

Le getter (accesseur) permet de retourner la valeur d’un attribut private.

Le setter (mutateur) permet de modifier la valeur d’un attribut static. L’intérêt d’avoir un setter est de pouvoir appliquer son action que d’une certaine façon, définie par le programme. Si un setter est défini, on ne pourra pas modifier l’attribut autrement que par son setter.

**Encapsulation :**

Le sens de l'encapsulation est de s'assurer que les données "sensibles" sont cachées aux utilisateurs. Pour y parvenir, vous devez :

* déclarer les variables / attributs de classe comme private ;
* fournir des méthodes get et set publiques pour accéder et mettre à jour la valeur d'une private variable.

Les private variables ne sont accessibles que dans la même classe. Cependant, il est possible d'y accéder si nous fournissons des méthodes publiques get et set. La get méthode renvoie la valeur variable et la set méthode définit la valeur.

* les attributs sont privés ;
* les méthodes permettant de modifier l’état de l’objet sont publiques ; elles s’assurent que l’objet reste cohérent.

L'encapsulation est un concept consistant à rassembler des données et des méthodes dans une structure en cachant l'implémentation de l'objet, c'est-à-dire en empêchant l'accès aux données par un autre moyen que les méthodes proposées (getter, setter ou méthode définie par le programmeur). Pour empêcher l’accès des attributs on utilisera des visibilités différentes : public, private, protected, package (cf: Visibilité )

L'encapsulation permet de garantir/contrôler l'intégrité des attributs contenus dans l'objet. Garantir/contrôler veut dire : faire en sorte qu’on ne puisse pas mettre n’importe quoi dans un attribut. Par exemple pour une heure, on ne peut pas avoir être 10h *83* . Car on n’écrit pas un nombre de minute supérieur ou égal à 60.

**Enumération :**

*Définition :* Une énumération est un ensemble fixe et relativement petit de valeurs ayant un nom et étant sémantiquement liées. Réunir ces valeurs permet de mieux structurer le code et d’y accéder plus facilement. Les valeurs de l’énumération ne changeront pas durant la durée de vie du programme.

En java, l’énumération est un type à part entière. Les instances sont décrites dans la classe. Elles portent un nom et sont constantes (nom en majuscule, séparés d’une virgule et la dernière est suivit par un point-virgule). C’est impossible d’en créer d’autres par la suite (constructeur privé).

|  |
| --- |
| ClassDeclaration:  NormalClassDeclaration  EnumDeclaration  EnumDeclaration:  {ClassModifier} enum Identifier [Superinterfaces] EnumBody |

L’énumération est comme une classe avec des fonctionnalités en plus :

* *Classe == énum :*
* c’est un type à part entière ;
* elle peut avoir des attributs, des méthodes et même un constructeur ;
* *Classe != énum :*
* conversion automatique vers une chaine ;
* peut apparaitre dans un switch ou un foreach ;
* fournit un tableau des valeurs de l’énumération ;

**Equals :**

La méthode equals() permet de vérifier si deux objets sont identiques. C'est avec cette fonction que vous effectuerez vos tests de condition. Elle return un booléen donc deux valeurs possibles true (si même type, même valeur et même hashcode() ) ou false.

Une méthode equals() est fournie par défaut dans la classe Object mais elle ne compare que les références des objets. Ainsi, lors de la création d’une nouvelle classe, il est de bonne pratique de réécrire la méthode equals (avec un @Override) pour l’adapter. Lorsque la méthode equals est adaptée il ne faut pas oublier d’également adapter le hashcode().

Cette méthode est surtout utilisée pour comparer des objets (notamment dans les listes, les

méthodes contains() et remove() utilisent la méthode equals(),…) mais aussi pour comparer

des String vu qu’il ne s’agit pas d’un type primitif (comparer à l’aide de « == » ne suffit pas).

La méthode equals() doit être :

* réflexive : un objet est toujours égal à lui-même.
* symétrique : si a = b alors b = a.
* transitive : si a = b et b= c alors a = c.
* consistante : peu importe le nombre de fois que la méthode est utilisée sur les mêmes objets, tant que ceux-ci ne sont pas modifiés le résultat de la méthode sera toujours le même.
* gérer les null : tout est différent de null (et donc chaque fois que quelque chose est comparé à null la méthode renverra false ou un nullPointerException). Deux null comparés ensemble renverront également un nullPointerException ou un false. Pour comparer deux null il vaut mieux utiliser “==”.

**Expression régulière :**

Les expressions régulières, appelées « expressions rationnelles » ou encore « regex », est une chaîne de caractères avec une syntaxe précise permettant d’obtenir un ensemble de chaînes composés de caractères possibles.

Une utilisation récurrente des regex consiste en la recherche de mots-clés dans des fichiers ou dans une base de données ou encore en la vérification des données saisies par l'utilisateur afin de s'assurer qu'elles respectent un format prédéfini, ou même d'opérer des conversions de format.

Chaque caractère permettant de réaliser notre regex tels que ^ , $ , \* , + , {} , [] , … sont appelés métadonnées.

|  |
| --- |
| ^[a-z0-9.\_-]+@[a-z0-9.\_-]{2,}\\.[a-z]{2,4}$ |

^ ce caractère spécifie le début d'une chaîne de caractères. $ celui-ci y spécifie la fin. Le + se trouvant à la fin de l'intervalle signifie que ce qu'il le précède (donc l'intervalle) peut

être répété 1 ou plusieurs fois. Le **\\.** qui permet de définir le point pour domaine

Les expressions régulières (*regex*) permettent de vérifier qu’une chaine correspond à un certain schéma (*pattern*).

|  |
| --- |
| Exemple : |
| **import** java.util.regex.Pattern;  String regex = "^[a-z0-9.\_-]+@[a-z0-9.\_-]{2,}\\.[a-z]{2,4}$" ;  String mail = "nom.prenom@domaine.com" ;  System.out.println(Pattern.matches(regex, mail)); // retourne vrai |

**Extends :**

Le mot extends permet à une classe d’hériter d’une autre classe. C’est-à-dire, qu’une classe enfant a la possibilité d’utiliser les méthodes ainsi que les attributs de sa classe parent. La classe enfant peut modifier les attributs et méthodes dont elle hérite et peut même en ajouter (mais ne peux y accéder que si la visibilité de ces membres sont protected et non private).

|  |
| --- |
| Exemple : |
| public class Animal { //parent  }  public class butterfly extends Animal { //enfant extends parent  } |

L’héritage multiple est illégal en java. On ne pourra donc pas avoir un Enfant avec deux parents. Pour pallier ce problème, on va utiliser des interfaces.

**Fichier texte/binaire :**

Ce sont des entrées/sorties de bas niveau. Java fournit des classes de bas niveau pour lire/écrire des **byte**/**char**. En java les fichiers sont des flux.

*Binaire :*

L'avantage des fichiers binaire est qu’ils possèdent une taille réduite. Nous pouvons grâce à des classes de bas niveau tels que FileInputStream , FileOutputStream (Bas niveau) ou encore DataInputStream , DataOutputStream (Haut niveau) écrire/lire dans un fichier binaire.

La différence entre ses classes est que DataInputStream / DataOutputStream permettent de gérer les données directement en tant que types de données primitifs (int, long, char, …) contrairement à FileInputStream / FileOutputStream qui gère les octets du flux.

FileInputStream et FileOutputStream héritent respectivement de InputStream et

OutputStream. Ces deux classes s’occupent donc des lectures et écritures des fichiers.

Il n’est pas possible d’instancier InputStream et OutputStream directement, cependant il existe des classes qui le font comme Files (classe utilitaire) avec ça méthode

newInputStream(Path, option) et newOutputStream(Path, option) .

*Texte :*

Les fichiers texte sont sur le même principe que les fichiers binaires mais avec des classes adaptées :

* FileReader (pour lire un fichier texte) : int read() → lit un caractère (-1 si fin du fichier)
* FileWriter (pour écrire un fichier texte) : void write(int c ) → écrire le caractère stocké dans c.
* BufferedReader (lire ligne par ligne ( et pas seulement byte par byte ) →

**Filtrer (fonctionnel) :**

On utilise un stream pour simplifier des traitements sur des collections ou des tableaux ! (Stream est un objet !)

Un stream se construit à partir d’une source de données et possède un certain nombre de propriétés spécifiques :

* Le stream ne stocke pas de données. Il ne fait que transférer d’une source vers une suite d’opérations.
* Un stream peut ne pas être borné (il n’a pas de limite, il est infini). (Mais il faudra cependant veiller à ce que nos opérations se terminent en un temps fini)

Un stream n’est pas réutilisable. Si on veut réutiliser les données de la source sur laquelle il avait été construit, nous serons obligés de reconstruire un nouveau stream sur cette même source. Il ne modifie pas les données de la source. S’il doit modifier les données pour les réutiliser, il va construire un nouveau stream à partir du stream initial. (Important pour la cohérence du parallélisme !!)

Le chargement des données pour des opérations sur un stream s’effectue de façon

lazy. Cela permet d’optimiser les performances de nos applications. Par exemple, si l’on recherche dans un stream de chaînes de caractères une chaîne correspondant à un certain pattern, cela nous permettra de ne charger que les éléments nécessaires pour trouver une chaîne qui conviendrait, et le reste des données n’aura alors pas à être chargé.

Les données passant par le stream peuvent être parcouru de 2 manières différentes.

* Séquentiellement : un à un
* Parallèlement : plusieurs à la fois

En fonctionnel il existe 2 types de méthode :

* Intermédiaire :

Les opérations intermédiaires (Stream.map ou Stream.filter par exemple) sont effectuées de façon lazy et renvoient un nouveau stream, ce qui crée une succession de streams que l’on appelle stream pipelines. Tant qu’aucune opération terminale n’aura été appelée sur un stream pipelines, les opérations intermédiaires ne seront pas réellement effectuées.

* Terminal :

Quand une opération terminale sera appelée (Stream.reduce ou Stream.collect par exemple), on va alors traverser tous les streams créés par les opérations intermédiaires, appliquer les différentes opérations aux données puis ajouter l’opération terminale.

On peut appliquer à un stream plusieurs méthodes intermédiaires mais une seule terminale.

**For :**

C'est un mot réservé du langage qui permet de définir des boucles. Il sert à coder le "pour". Si on connaît le nombre de récurrence, on utilisera souvent un for classique. Sinon, on se contentera en général d'un while.

Il existe deux variantes pour le for : le for classique et le foreach.

*Le for basic*

|  |
| --- |
| For (forInit ; expression ; forUpdate) {  Statement ;  } |

Le for est composé de trois parties : le forInit, l’expression et le forUpdate.

Dans le forInit, on initialise la boucle en déclarant la variable. Il est facultatif si l’on a déjà déclaré une variable locale que l’on souhaite utiliser à la place. On ne met pas de forInit si on veut que la variable incrémentée par la boucle puisse être utilisée dans un autre endroit que la boucle.

L’expression, c’est le test que l’on réalise avec une expression booléenne. Si true, Statement, ForUpdate et recommencer, sinon si false, instruction suivant la boucle for. Enfin, dans le forUpdate, il actualise la variable.

Le Statement est le corps de la boucle et est obligatoire. Ça représente une instruction ou un bloc d'instructions.

|  |
| --- |
| Exemple : |
| For(int i =0 ; i<tab.length ; i++){  System.out.println(tab[i]);  } |

Les seules raisons acceptées pour utiliser un for au lieu d’un foreach sont lorsque :

* L’on a besoin de l’index
* L’élément parcouru doit être modifier

**For each:**

*Le for amélioré*

Ce for permet de parcourir un tableau ou un itérable. Ce for est plus concis et plus rapide mais on n’a pas accès à l’indice et il est impossible de modifier un élément. C’est un raccourci pour le for basic.

|  |
| --- |
| for(Type Identifier : Expression) {  Statement  } |

* Type : Doit être du même type que ce qui compose l'élément à parcourir
* Identifier : Le nom de la variable
* Expression : Une expression itérable, comme un tableau qui a été initialisé précédemment ou encore une liste.

|  |
| --- |
| Exemple : |
| For(int chiffre : tab){  System.out.println(chiffre);  } |

Mais qu'on ne peut pas avoir accès au compteur dans le foreach, ni aux indices et on ne peut pas modifier le contenu de l'élément iterable dans le foreach.

**Grammaire :**

La grammaire est décrite dans The Java Language Specification. Il existe la grammaire lexicale, syntaxique et sémantique. Elle permet de définir ce qui est autoriser ou non d’écrire.

Un code est correct s’il peut être produit par la grammaire.

Une grammaire est une description finie de l’infinité des programmes :

* Un mot (token) doit être légal, grammaire lexicale
* Une séquence de mots doit être légale, grammaire syntaxique
* Le tout doit avoir un sens, sémantique
* Exemple : “ *Loïc est modeste* ” n’as pas de sens, erreur sémantique.
* Grammaire lexicale :
* Les symboles terminaux sont les caractères (characters of Unicode characters set).
* Les règles de production forment les mots (tokens), éléments d’entrée (input éléments) de la grammaire syntaxique.
* Exemple :

*“dkfjklsdjflds”* n’est pas un mot légal, il y a erreur de grammaire lexicale.

* Grammaire syntaxique :
* Les symboles terminaux sont les tokens.
* Les règles de production permettent de définir ce qu’est un programme syntaxiquement correct.
* Parmi les éléments importants d’un programme, on trouve :
* Les expressions :
* Représentent les « *calculs* »
* Ont une valeur et un type
* On y reviendra continuellement sans jamais tout faire en une fois
* Les instructions
* Les expressions-instructions
* Exemple :

“ *Ce que tu as appris, désapprendre tu dois !* ” n’est pas une séquence légale, on ne respecte pas la grammaire

**If :**

|  |
| --- |
| If (expression) {  Statement ;  }Else{  Statement2 ;  } |

Le if permet de mettre des conditions grâce à des expressions booléennes. Si l’expression booléenne contenue dans les parenthèses suivant le if est vrai, le statement est exécutée. Sinon le statement2 est exécuté. Le else est un peu considéré comme le cas par défaut.

Si on veut on peut mettre plus d’une condition. Alors, il faut utiliser le if – else if. Si il y a trop de else if il vaut mieux utiliser un switch. Sinon, le else if fonctionne comme pour le if.

Les accolades sont optionnelles, si on ne les met pas, ça compile quand même et les instructions sont exécuté́. On met donc les accolades par convention.

Quand on exécute plus d'une instruction et qu'on omet les accolades, seule la première instruction est considérée dans le if, ce qui n'est pas le but si par exemple, on les utilise afin de rassembler un bloc d'instruction.

Lorsque le code à l'intérieur des accolades n'est composé que d'une seule ligne, elles deviennent facultatives.

**Implements :**

Pour expliquer ce que signifie le mot implements. Il faut d’abord expliquer ce qu’est une interface.

Une interface est une classe totalement abstraite. Elle ne contient que la signature des méthodes abstraite. La définition du code des méthodes de l’interface est la responsabilité de la classe qui va implémenter celle-ci. Une interface ne peut posséder de constructeur. Elle ne donc peut être instanciée.

|  |
| --- |
| Exemple : |
| public interface Model { } |

Une interface :

* définit un type de données
* on peut donc déclarer un objet de ce type
* mais il faut instancier une classe concrète

Une interface peut hériter d’UNE SEULE autre interface (attention hériatge multiple interdit en java) :

|  |
| --- |
| Exemple : |
| public interface int1 extends int2 |

Ce mot permet à une classe d’implémenter des interfaces et définir le code de toutes les méthodes. Une classe implémentant une interface doit implémenter (définir le code de) chaque méthode de l’interface (elle peut cependant en rajouter).

Pour implémenter une méthode, il suffit de la définir comme on le ferait pour une méthode en général. On pourra ajouter l’annotation @Override mais elle est facultative (vu qu’une méthode abstraite est toujours override).

|  |
| --- |
| Exemple : |
| public class Game implements Model { } |

Une classe peut implémenter plusieurs interfaces :

|  |
| --- |
| Exemple : |
| public class Game implements Model, Jeu { } |

Une classe qui implémente une interface possède toutes les méthodes et attributs de l’interface implémentée.

Si on veut de l’héritage et de l’implémentation pour une classe, il faut toujours donner l’héritage d’abord et ensuite les interfaces à implémenter :

|  |
| --- |
| Exemple : |
| public class Butterfly extends Animal implements Animaux { } |

**Import :**

Le mot-clé import permet, comme son nom l’indique, d’importer dans une classe une ou plusieurs classes contenues dans un autre package que celui courant. Cela permet par la suite d’appeler des méthodes sans devoir spécifier leur chemin complet. Import est donc une sorte de raccourci améliorant la lisibilité du code, il n’est pas obligatoire de l’utiliser (mais c’est une bonne pratique).

Toujours dans ce but d’amélioration de la lisibilité, il faut faire attention de ne pas avoir des import inutiles.

Les import se placent au début du programme, après avoir (optionnel) déclaré le package dans lequel la classe se trouve.

Il est possible d’importer une classe spécifique ou un ensemble de classes. Le caractère \* étant un joker signifiant que toutes les classes du package doivent être importées. Il est obligatoire de spécifier le nom du package que l’on souhaite importer, « import \* » ne compilera pas.

Il est possible d’importer en utilisant le mot-clé static. Attention, l’import static doit désigner un membre de la classe, pas la classe en elle-même.

Utiliser ce mot-clé permet par la suite de pouvoir utiliser le membre directement sans devoir spécifier la classe à laquelle il appartient.

**Itérer (fonctionnel) :**

Supposons qu’on veuille liker toutes les vidéos d’une liste :

|  |
| --- |
| Exemple : |
| List<Video> videos = Arrays.asList(  new Video("Alexandre Astier", "Kaamelott", true, 1\_236\_722),  new Video("Dominique A", "Au revoir mon amour", true, 455\_262),  new Video("Michael Launey", "Dimensions Stade foot", true, 64\_598)  ); |

On peut utiliser un for each :

|  |
| --- |
| Exemple : |
| public void likerAll ( List<Video> videos) {  for (Video v : videos) {  v. liker ();  }  } |

Java 8 : méthode forEach et lambda :

|  |
| --- |
| Exemple : |
| public void likerAll ( List<Video> videos) {  videos .forEach( v −> v.liker () );  } |

Qu’on peut aussi raccourcir en (method reference) :

|  |
| --- |
| Exemple : |
| public void likerAll ( List<Video> videos) {  videos .forEach( Video:: liker );  } |

Code précédent plus compact mais pas plus rapide. On peut paralléliser l’exécution en utilisant un stream :

|  |
| --- |
| Exemple : |
| public void likerAll ( List<Video> videos) {  videos . parallelStream (). forEach( Video:: liker );  } |

**Object :**

Un langage orienté objet permet de créer ses propres types, liés au problème à résoudre. Un programme peut créer et manipuler les objets. Un type de données OO est défini par une classe. Un objet est une instance d’une classe :

* construit à partir de la définition donnée par la classe ;
* appartenant au type défini par la classe.

On appelle instance d'une classe, un objet avec un comportement et un état, tous deux définis programmation orientée objet s par la classe. Il s'agit donc d'un objet constituant un exemplaire de la classe.

Un objet se caractérise par :

* son état :
* stocké dans des attributs (des variables liées à l’objet)
* son comportement (ce qu’on peut en faire):
* défini par des méthodes (du code)

Instancier un objet c’est le construire en mémoire :

* lui réserver de l’espace en mémoire (sur le tas)
* initialiser son état (ses attributs)

Initialisation prise en charge par un constructeur. (voir mot constructeur)

Pour instancier un objet :

* On utilise l’opérateur new
* On fournit les paramètres au constructeur

Une classe est un type référence. Le constructeur peut vérifier la validité des paramètres ;

Null :

* littéral de type référence.
* indique qu’il n’y a pas d’objet (référence vers rien) ne pas confondre null avec " ".

Il existe même un constructeur par défaut, il est sans paramètre et ne fait rien. Il n'est plus disponible si un autre constructeur est fourni. Une classe d'objet peut posséder plusieurs constructeurs, on appelle ça la surcharge (overloading). Le constructeur, les méthodes et les attributs d'un objet peuvent être représenté par "this".

*Class Object :*

Java.lang.Object

Public class Object

La classe parent de toutes les classes est Object. Toutes les classes héritent de cette classe et donc de :

* String toString () représentation textuelle :
* cette méthode « doit » être réécrite (*override*)
* par défaut retourne le type et le hash de l’objet
* boolean equals(Object o) égalité sémantique :
* permet de définir quand deux objets sont égaux
* par défaut, retourne ==
* réflexive, symétrique et transitive
* L’implémentation de equals est polémique :
* utilisation de getClass
* utilisation de instanceof
* rendre la classe ou la méthode equals final
* int hashCode() hash associé à chaque objet :
* deux objets « equals » -> même hashcode
* hashcodes différents -> objets differents (contraposée)
* objets différents - /-> hashcodes différents (mais c’est mieux)
* Object clone() retourne une copie de l’objet :
* méthode mal conçue dont l’usage est polémique
* il est préférable d’utiliser un constructeur par copie

**Objects :**

La classe Objects est une classe utilitaire (static) comprenant uniquement des méthodes qui servent à la manipulation d’objets :

* static boolean deepEquals(Object o1, Object o2) :

Retourne vrai si les deux objets sont profondément égaux. Deux objets null sont profondément égaux.

Si les deux arguments sont des Array, alors c’est la méthode Arrays.deepEquals() qui va déterminer l’égalité profonde. Sinon l’égalité est testée par l’equals du premier argument. Deux Arrays sont profondément égaux s’ils référencent vers des tableaux de même taille et dont les éléments aux même indices sont deepEquals.

|  |
| --- |
| Exemple : |
| String st1 = "coucou" ;  String st2 = "coucou" ;  System.out.println(Objects.deepEquals(st2, st1)); //affiche vrai |

* static boolean equals(Object o1, Object o2) :

Le equals, lui va comparer strictement les éléments d’un tableau (s’il tombe sur des références ,à l’intérieur, du tableau, il va comparer les références et non les éléments pointés par les références).

|  |
| --- |
| Exemple : |
| **int** [] tab1 = {1, 2, 3};  **int** [] tab2 = {1, 2, 3};  System.out.println(Arrays.equals(tab1, tab2)); //affiche vrai |

* static T requireNonNull(T t) :

Renvoie true si la référence fournie est non nulle sinon renvoie false.

* static int hash(Object ... values ) :

Génère un hash code pour les valeurs entrée en paramètre. Le code de hachage est généré comme si toutes les valeurs d'entrée étaient placées dans un tableau, et que ce tableau était haché en appelant Arrays.hashCode (Object []).

|  |
| --- |
| Exemple : |
| @Override  **public int hashCode** () {  **return** Objects.hash(x, y, z);  } |

**Polymorphisme :**

*Traduction :* plusieurs formes

Il permet de manipuler des objets sans vraiment connaître leur type. Le polymorphisme est la capacité d’une méthode à avoir plusieurs formes. C’est à dire qu’elle peut avoir plusieurs comportements. Une méthode est appelée de la même manière mais réagit différemment en fonction de l’objet (et surtout de la classe qui l’invoque).

|  |
| --- |
| Exemple : |
| ***On crée une classe Voiture :***  Classe Voiture{  }  ***Puis on crée d’autre classe de marques différentes qui héritent de Voiture:***  Class Audi extends Voiture{  }  Class Bmw extends Voiture{  }  ***Faire ça n’est pas du polymorphisme :***  Audi a1 = new Audi() ;  Bmw b1 = new Bmw() ;  ***Faire ça est du polymorphisme :***  Voiture a1 = new Audi() ;  Voiture b1 = new Bmw() ; |

**Post-incrémentation :**

i++ (post-incrémentation) et ++i (pré-incrémentation) sont des expressions.

++ est un opérateur qui permet l’incrémentation d’une variable.

Dans la post-incrémentation, i donne d’abord sa valeur à l’expression i++ puis i est incrémentée. Pour la pré-incrémentation, les étapes sont inversées.

|  |
| --- |
| i++ = i; // l'expression (i++ prend la valeur de i)  i = i + 1; //i est incrémentée  // post incrémentation => l'incrémentation se fait POSTérieurement à l'assignation de l'expression |

Post-incrémentation(i++) i donne sa valeur à l’expression avant d’être incrémenté.

**/ ! \ Remarquez que c’est toujours le i qui donne sa valeur à l’expression, jamais l’inverse.**

|  |
| --- |
| Exemple : |
| public static void main (String[] args) {  int i = 1;  System.out.println(i++); //affiche 1  System.out.println(i); //affiche 2  } |

**Static :**

Static est un mot-clef utilisé pour la gestion de la mémoire. Il peut être appliqué aux membres, c’est-à-dire aux attributs, méthodes et constructeurs.

Lorsqu’un membre static est utilisé, cela indique que le membre en question appartient à la classe plutôt qu’à une instance de la classe ; ce qui veut dire qu’un seul exemplaire du membre est créé et partagé à travers toutes les instances de la classe.

*Attribut statique :*

Si un attribut est static cela signifie que, peu importe le nombre de fois que la classe sera initialisée, une seule copie de cet attribut sera créée. La valeur de l’attribut sera partagée entre tous les objets. Il est donc intéressant de déclarer un attribut static lorsque sa valeur est indépendante des objets et qu’elle sera partagée à travers eux.

On accède à un attribut static directement à partir du nom de la classe, il n’est pas nécessaire d’initialiser un objet ou d’en utiliser un déjà créé (et c’est même déconseillé, même si cela reste possible, pour éviter les confusions entre les attributs de classe et d’instance de classe).

* utilisation courante : constantes

*Méthode statique :*

Appartiennent à la classe plutôt qu’à l’objet. Encore une fois, pas besoin de créer un objet pour l’utiliser, la méthode est indépendante.

Attention, les méthodes static ne peuvent pas être Override car elles sont lancées lors de la compilation et un Override est utilisé lors de l’exécution. Les méthodes abstract ne peuvent pas être static non plus étant donné car le but d’une méthode abstract est d’être Override. Elles ne peuvent pas utiliser les mots-clefs this ou super.

Les méthodes static peuvent accéder directement aux attributs et méthodes static mais pas les non static (elles ont besoin d’un objet pour ça). Les méthodes d’instance peuvent accéder directement à tout, static ou non.

* ne peut pas accéder aux membres des instances
* utilisation courante : méthodes non objets

*Constructeur static :*

Un constructeur static est utilisé pour initialiser des attributs static. Même si il est possible de les initialiser static lors de la déclaration.

*Import static :*

Un import static permet d’accéder aux membres d’une classe sans devoir spécifier le nom de la classe. Donc, crée un raccourci pour l’accès aux membres statiques.

*L’opérateur conditionnel ? :*

Utilise la valeur booléenne de la première expression pour décider laquelle des deux autres expressions sera évaluée.

|  |
| --- |
| Exemple : |
| int value = n < 0 ? −n : n;  EST EQUIVALENT A:  int value ;  if (n < 0){  value = −n;  } else {  value = n;  } |

*Appel d’une méthode :*

* statique : par le biais de la classe
* non statique : par le biais d’une instance de la classe

**Switch :**

Le switch c’est comme plusieurs if imbriqués. La variable dans le switch peut être byte, char, short, int, Byte, Character, Short, Integer, String ou enum.

|  |
| --- |
| SwitchStatement:  switch ( Expression ) SwitchBlock  SwitchBlock:  { {SwitchBlockStatementGroup} {SwitchLabel} }  SwitchBlockStatementGroup:  SwitchLabels BlockStatements  SwitchLabel:  case ConstantExpression :  case EnumConstantName :  default : |

* Switch :

Le mot clé switch est suivi d’une expression mise en paramètre. Cette expression est testée avec les différents cas.

* Case :

Le mot clé case est suivit par la valeur avec laquelle l’expression doit être comparée. Sa valeur est du même type que celle de l’expression exécutée. Il n’y a pas un nombre de case limité.

* Default :

C’est le cas par défaut qui sera exécuté si aucun case ne correspond. En général, il est écrit en après le dernier case et n’est pas suivi d’un break. S’il n’y a pas de default après le dernier case, le switch passera à la prochaine instruction (une fois toutes les case parcourus).

* Break :

Le break permet de sortir du switch quand on passe dessus. Tant qu’il n’y a pas de break, même si on a trouvé un case correspondant on va exécuter les instructions du cas suivant.

|  |
| --- |
| Exemple : |
| public enum Direction {  NORTH(-1, 0),  SOUTH(1, 0),  EAST(0, 1),  WEST(0, -1);  public Direction opposite() {  switch (this) {  case NORTH:  return SOUTH;  case SOUTH:  return NORTH;  case EAST:  return WEST;  case WEST:  return EAST;  default:  return null;  }  }  } |

**Tableau 1D :**

Un tableau est un type de référence. Il est une structure de donnée regroupant plusieurs éléments du même type accessible via un indice. Le premier indice est 0 et le dernier étant TAILLE - 1.

Un tableau est un objet puisqu’il hérite de java.lang.Object. On peut créer un tableau en donnant les valeurs et les tailles. Une fois l’espace alloué en mémoire pour n composantes de type T, on ne peut pas la changer la taille du tableau.

*Déclaration :*

|  |
| --- |
| TypeName [] Name |

Les crochets indiquent au compilateur que la variable représente un tableau. A ce moment-là, le compilateur réserve un espace mémoire portant le nom du tableau. Après la déclaration, la variable tab existe, mais n’est pas encore initialisée à un tableau. tab ne référence pas encore, elle contient la valeur null.

*Création :*

Deux manières de créer un tableau :

* Manuellement :

|  |
| --- |
| Exemple : |
| int[ ] tab = {5, 6, 34} ; |

* En lui donnant sa taille initiale :

|  |
| --- |
| Exemple : |
| int [ ] tab = new int[3] ; |

Les tableaux peuvent être parcourir par un for ou un foreach.

Il existe une classe utilitaire pour les tableaux qui a des méthodes tels que :

* binarySearch () recherche dans le tableau
* copyOf copie tout ou partie du tableau
* fill () remplit le tableau
* sort () trie le tableau
* toString représentation du tableau
* deepToString représentation « en profondeur »
* equals égalité des valeurs de deux tableaux
* deepEquals égalité « profonde »

L'utilisation d'un tableau pour lequel l'espace n'a pas été alloué provoque la levée d'une exception: *NullPointerException*

L’accès à un élément du tableau en dehors des bornes provoque la levée d’une exception :

*ArrayIndexOutOfBoundsException*

Les éléments d’un tableau sont indicés à partir de 0. Chaque élément peut être accédé individuellement en donnant le nom du tableau suivi de l’indice entre [ ].

**Tableau 2D :**

Le tableau 2D est un tableau qui contient des tableaux. Ils gardent les mêmes caractéristiques que les tableaux à une dimension. Lors de la définition de tableau, le nombre de de crochets indique le nombre de dimensions du tableau. Un tableau à deux dimensions s’organise non plus sur une seule ligne, mais sur des lignes et des colonnes.

La déclaration et initialisation se base sur le même principe que le tableau à 1D :

|  |
| --- |
| TypeTab [][] nomTab = new TypeTab[nbLignes][nbColonnes]; |

Pour trouver le nombre de ligne → monTab.length

Alors que pour trouver le nombre de colonnes → monTab[0].length

|  |
| --- |
| Exemple : |
| int [][] test = new int [2][3]; |

**This/super :**

*This :*

this est une référence à l’instance de l’objet en cours d’utilisation (à soi-même).

Elle apparait dans différents contextes :

* constructeur this () ; Lorsque qu’un constructeur en appel un autre, l’instruction doit bien être la première instruction du constructeur.
* attribut this.titre ;
* méthode this.liker ().

Il permet aussi de passer l’objet en paramètre. Enfin il rend le code plus lisible car c'est un raccourci de langage !

On peut également utiliser this pour appeler le constructeur de la classe courante.

Lorsqu’une classe hérite d’une autre, on utilise super au lieu de this.

*Super :*

Il fait référence au constructeur parent et doit être la première instruction dans un constructeur enfant. On parle ici d’héritage car pour référencer d’un constructeur parent on doit hériter de celui-ci, et on peut le faire grâce au mot clé extends où on va récupérer les attributs et les méthodes du parent.

On peut également utiliser this comme pour appeler le constructeur de la classe.

Dans un constructeur enfant , super(...) doit être la première instruction. Même si le constructeur n’a pas de paramètres → super(...) est obligatoire. Tout ceci apparaît dans le concept d’héritage.(seul les classes en public et en protected peuvent être utilisée dans une classe héritée.

**Throw(s) :**

*Throw :*

Le throw permet de déclencher une exception.

|  |
| --- |
| Exemple : |
| If(number<0){  throw new IllegalStateException("The number is negative.");  } |

*Throws:*

Le throws permet à une méthode de lancer une exception contrôlée. Le throws doit apparaître dans l’entête de la méthode juste avant l’ouverture d’accolade.

Le fait de déclarer une exception dans la signature de la méthode indique au compilateur que la méthode peut renvoyer une exception, et il obligera les développeurs qui utilisent cette exception à la traiter (ou à la renvoyer).

Quand on appelle une méthode qui peut lancer une exception contrôlée, on doit soit la gérerpar un “try-catch” soit déclarer qu’on la lance par un “throws” (on laisse passer l’exception).

*En résumé :*

Throws permet de prévenir le compilateur qu’une erreur pourrait être levée dans la méthode en question.

Throw par contre va permettre de lever et de gérer une exception “manuellement”.

**ToString :**

En Java, la méthode toString() retourne une représentation d’objet sous forme de chaîne de caractères.

Elle est définie dans la classe java.lang.Object mais elle peut être Overridé dans une classe enfant.

System.out.println( ) va rechercher la méthode toString() qui a été créée dans la classe.

La méthode toString() d’un objet sera déclarée dans sa classe et va donc permettre la même notation pour chacun de ses objets. Un System.out.println() par contre se placera généralement dans la main (ou dans la view) afin d’afficher un texte personnalisable pour chacun des objets grâce à des getters.

String toString ()

* fournit une représentation textuelle basique de l’état
* nom standardisé
* appelée automatiquement par println ou lors d’une concaténation
* version par défaut existe mais pas intéressante

**Trier :**

L’API fournit des méthodes pour trier tableaux et listes. Java permet déjà de trier des listes ou des tableaux mais si on cherche à trier des objets non standardisés c’est plus compliqué.

*Tri d’objet non standards :*

Il faut que la classe soit Comparable (cf. API)

* Interface : int compareTo(T o)
* Définit l’ordre naturel
* Utilisée par l’algorithme pour comparer deux éléments

*Tri selon un ordre personnalisé :*

Il faut fournir un Comparator (cf. API)

* Interface : int compare(T o1, T o2)
* Définir une classe implémentant cette interface
* Passer une instance à une autre version de sort void sort (T[] t , Comparator<T> c)
* Utilisée par l’algorithme pour comparer deux éléments

**Try-catch :**

|  |
| --- |
| try {  // code pouvant lancer une exception  } catch (IllegalArgumentException e){  // gestion de l ’ erreur  } |

Permet de traiter et gérer une exception avant qu’elle ne soit lancé. Un try catch peut avoir plusieurs catch ou un catch multiple en utilisant le séparateur | (ou).

On met les instructions qui peuvent mal se passer dans le bloc try{ } et si une exception survient, il y'a dans le catch { } les instructions qui vont régler le problème. Ces deux mots clés sont indissociables les uns des autres, il ne peut y avoir de try sans catch et inversement.

|  |
| --- |
| Exemple : |
| int a = 100 ;  try {  int b = scanner.nextInt() ;  int c = a/b //si on a rentré 0, il y'aura une exception car on ne peut pas diviser par zéro  } catch (Exception e) {  system.out.println(« Division par 0 impossible ») ;  } |

Quand une instruction se passe mal dans le try, le code du try est interrompu, et le code du catch est exécuté. Dans le catch, il faut préciser le type et le nom de t’exception qui doit être attrapée, sinon tout type d’exception sera attrapée. Pour préciser, il suffit de modifier ce qui se trouve entre parenthèse après le catch.

Il existe aussi le try with ressources :

Le try instruction-with-resources est une try instruction qui déclare une ou plusieurs ressources. Une ressource est un objet qui doit être fermé une fois le programme terminé. Le try instruction-with-resources garantit que chaque ressource est fermée à la fin de l'instruction. A noter qu'il est obligatoire de définir les ressources à fermer dans les parenthèses qui suivent le try, sinon, erreur de compilation.

**Var :**

L’inférence de type pour les déclarations de variables locales avec initialiseurs est la possibilité de ne pas répéter le type d’une variable locale lors de sa déclaration et d’utiliser le mot réservé var.

* Il faut choisir un nom de variable qui donne de l’information utile.
* Il faut minimiser la portée des variables.
* Utiliser var quand l’initialiseur donne suffisamment d’informations.
* Ne pas trop s’inquiéter de l’utilisation des interfaces :

Dans le cadre d’une variable locale et non d’un attribut, de paramètres de méthodes ou d’un type de retour, ce n’est pas important.

* Être attentif avec l’opérateur diamond et les génériques.

**Var args :**

Il est possible de définir une méthode dont le nombre d’arguments est variable. Varargs donne une flexibilité sur la façon dont on utilise une méthode.

Varargs :

* indiquent un nombre variable d’arguments
* reçoit les arguments sous forme d’un tableau

La notation « … » utilisée dans un tel contexte est appelée « ellipse ». L’ellipse concerne des arguments d’un type quelconque, primitif ou objet et peut être accompagnée d’arguments classiques.

|  |
| --- |
| Exemple : |
| public void function ( char c, int ... x) {...} |

* Il ne peut pas y avoir trop de var args et il doit être le dernier argument.

Une ellipse :

* est placée après le type de l’objet
* est unique dans une méthode
* est toujours le dernier argument de la méthode (cela permet de ne pas avoir d'ambiguïté sur les arguments mis en paramètre

Le type suivit de l’ellipse indique que la méthode peut accepter un nombre variable d’arguments. L’avantage est la facilité d’écriture dans l’appel.

**Visibilité :**

Chaque membre peut avoir comme visibilité :

* privé :

Quand l’attribut private est appliquée sur un champ, il n’est pas accessible en dehors de cette même classe sauf via un accesseur (getter) ou un mutateur (setter) (puisqu’ils sont public).

|  |
| --- |
| Exemple : |
| private int nombre; |

* paqueté :

Pour la visibilité Package il n’y a pas de mot-clé. Elle permet de rendre les membres accessibles par l’entièreté du même package.

|  |
| --- |
| Exemple : |
| public class Esi {  boolean isReussi;  int nombre;  int somme ( int value) {  return value+value;  }  } |

* protégé :

La visibilité protected est utilisée en héritage lorsqu'on veut que les attributs et les méthodes de la classe mère soit visibles dans la classe fille. Du coup, au lieu de mettre le membre comme étant private dans la classe mère, on le met protected.

|  |
| --- |
| Exemple : |
| public class Employe {  protected String nom;  protected String prénom;  protected int matricule;  protected int salaire;  } |

Lorsqu'on utilisera le mot clé extends, les attributs de la classe mère seront accessible pour la classe fille et elle va donc pouvoir y accéder sans restriction via l'héritage, ce qui n'aurait pas été possible si l'attribut était private.

* public :

La visibilité Public permet d'accéder aux champs à partir de toutes les autres classes, on peut y accéder comme on le souhaite autant pour lire que pour modifier avec : NomDeLaClasse.nomDuChampPublic

|  |
| --- |
| Exemple : |
| public int nombre; |

**While/do-while :**

*While:*

Contrairement au for il est utilisé lorsqu'on veut arrêter le traitement à un moment donné mais qu'on ne connait pas le nombre d'itération à l'avance. On quittera le while quand la condition ne sera plus respectée. Le while peut aussi être utilisé comme un for.

|  |
| --- |
| while ( Expression ) Statement |

* Expression est booléen, si elle return true, on rentre dans le while
* C’est Statement qui se charge de « l’incrément »

Une boucle commence par l'utilisation du mot clef, ici : while. Cela veut dire, « tant que ». Puis nous avons une condition : c'est elle qui permet à la boucle de s'arrêter. Une boucle n'est utile que lorsque nous pouvons la contrôler, et donc lui faire répéter une instruction un certain nombre de fois. C'est à ça que servent les conditions.

Contrairement au *for* il est utilisé lorsqu'on veut arrêter le traitement à un moment donné mais qu'on ne connait pas le nombre d'itération à l'avance. Ensuite nous avons une ou plusieurs instructions : c'est ce que va répéter notre boucle (il peut même y avoir des boucles dans une boucle).

|  |
| --- |
| Exemple : |
| int indice = 0;  while ( indice < tableau.length && tableau[ indice ] == 0) {  indice++;  } |

*Do-while :*

Le do while fonctionne comme le while mais fait au moins une itération avant de vérifier la condition.

|  |
| --- |
| do Statement while ( Expression ) |

* C’est un répéter tant que

|  |
| --- |
| Exemple : |
| int nb;  do {  nb = clavier.nextInt ();  } while (nb%2==0) ; |

*Les différences entre while et do-while :*

* La boucle do… while s'exécutera au moins une fois, contrairement à while . C'est-à-dire que la phase de test de la condition se fait à la fin, car la condition se met après le while.
* C'est une différence de syntaxe, qui se situe après la condition du while. Il y a un « ; » après le while.

**Wrapper/boxing :**

Les listes n’admettent que des objets comme éléments. Il est alors impossible de mettre un type primitif dans une liste. Comment faire pour tout de même avoir des listes de booléen, d’int ou de char? La solution est de transformer un type primitif en objet. C’est le wrapping!

*Wrapper :*

Seuls les objets sont permis dans les listes.

Peut-on avoir une liste de int ?

* existence de wrapper (enveloppe)
* englobe une valeur primitive dans un objet
* boolean : Boolean
* int : Integer
* byte : Byte
* long : Long
* char : Character
* float : Float
* short : Short
* double : Double

Les conversions du type primitif vers son wrapper et vice versa sont automatiques (boxing / unboxing). Les wrappers sont aussi des classes utilitaires. Les classes wrapper sont moins efficaces que les types primitifs. Donc, il ne faut pas utiliser les wrappers quand on peut utiliser un type primitif.

*Boxing/unboxing :*

Le boxing est l’enveloppement d’un type primitif dans un Objet. L’unboxing est le phénomène inverse. C’est quand on va sortir le type primitif de son wrapper.

*Résumé :*

|  |
| --- |
| Exemple : |
| int n1 = 7; // valeur 7 stocké directement dans la pile. Integer n2 = 8 // Integer est un type référence. |

La valeur 8 est stockés dans les attributs de l'instance n2 de type Integer. L'objet est stocké sur le tas et une référence pointant vers cet objet est stockée dans la pile.

On dit que la classe Integer est une classe "wrapper" pour le type int. (to wrap = emballer)

On vient d'écrire :

|  |
| --- |
| Integer n2 = 8; |

Normalement on aurait dû écrire:

|  |
| --- |
| Integer n2 = new Integer(8); |

Ou encore :

|  |
| --- |
| Integer n2 = Integer.valueOf(8) |

Mais le compilateur s'en occupe automatiquement : c'est l'auto-boxing.

L’auto-boxing a 2 scénarios possibles:

1. Quand une valeur primitive (7, true, …) est entrée en paramètre dans une méthode qui s'attend à recevoir un objet de type wrapper correspondant (Integer, Boolean, …).

|  |
| --- |
| Exemple : |
| List<Integer> nombres = new ArrayList<>();numbers.add(42);//add() s'attend à recevoir Integer mais reçoit int.// -> autoboxing |

1. Quand une valeur primitive est assignée à une variable de type wrapper correspondant.

|  |
| --- |
| Exemple : |
| Integer nombre = 42;// l'opérateur d'assignation = s'attend à Integer mais trouve int.// -> autoboxing |

unboxing:

|  |
| --- |
| Integer n1 = 17; // auto-boxing |
| int n2 = n1.intValue(); // unboxing |
| int n3 = n1; // auto-unboxing |

|  |
| --- |
| Exemple: |
| int sommePairs (List<Integer> nombres) {    int somme = 0;    for (Integer n : nombres) {        if (n % 2 == 0) { // auto-unboxing de n            somme += n; // auto-unboxing de n        }    }    return somme;} |

Valeurs possibles:

|  |  |
| --- | --- |
| **Type primitif** | **Classe Wrapper** |
| type donnée/valeur | type référence |
| plage fixée de valeurs possibles (exemple [-128;127] pour un byte) | même plage que type primitif correspondant + null |

Hiérarchie des classes :

Les classes wrapper sont toutes "final". On peut les utiliser tel quel mais on ne peut pas en hériter.

|  |
| --- |
| Exemple : |
| public final class Integer extends Number implements Comparable < Integer > {  } |

Utilité :

* Fonctionnalités utiles:

|  |
| --- |
| Exemple : |
| int number = Integer.parseInt("1337"); |

* Programmation générique : accepte uniquement les types références.

|  |
| --- |
| Exemple : |
| List<int> myInts = new ArrayList<>(); // erreur compilation  List<Integers> myInts = new ArrayList<>(); // compile |