notes Java:

Compiler/Exécuter un programme depuis la ligne de commande :

- javac nom_fichier.java (on doit d'abord se positionner dans la répertoire contenant le fichier à compiler)
- java nom_fichier arg1 arg2 arg3… pour exécuter le programme en envoyant des arguments à son main , ces arguments seront reçu dans le tableau de main (appelé noramlement args) tel que args[0]=arg1 , args[1]=arg2… (au contraire de C , java ne reserve pas la premier case du tableau args pour le nom du fichier)

Les méthodes static :

- les méthodes de classe (méthodes static) ne peuvent pas accéder directement aux variables d'instance ni aux méthodes d'instance (non static), ils doivent utiliser un objet de référence pour y accéder
- les méthodes static ne peuvent pas utiliser le mot clé « this » puisqu'une méthode static est habituellement appelé à travers le nom de la classe elle-même.

variables primitifs (simple :int,char…) ainsi que les objets sont passé aux méthodes par référence :

- Quand un objet est envoyé à une méthode, il est passé par valeur, si on change la référence de l'objet, ce changement sera perdu quand on sort de classe, cependant si on change les valeurs des attributs de l'objet, ces changements vont persister.

En java, les paramètres pour les types primitifs sont passés par valeur, et pour les types objets c'est une copie de la référence Java (un **Handle**) qui est transmise.

Java MythBusters

Scanner

- Il n'existe pas de méthode pour extraire un caracètre (~~nextChar()~~), on peut utiliser nextLine() et charAt(0)
- nextLine() récupère le contenu de toute la ligne saisie et replace la « tête de lecture » au début d'une autre ligne, si on l'appelle après un autre next**Type** (sauf nextLine) , elle ne va rien récupéré.

Les listes : (ArrayList et LinkedListe)

- les types de listes qu'on a fait :
    - ArrayList
    - LinkedList

  les deux sont manipulés de la même façon, la différence entre les deux c'est leur façon de manipuler les données : le premier type (Arraylist) permet un accès rapide aux données (c'est un tableau dynamique) , le deuxième est plus rapide en gestion de données (une liste chainée)
- la déclaration :
    - List<type> nom_list = new **ArrayList**<type>()
    - List<type> nom_list = new **LinkedList**<type>()
- la manipulation :
    - public void add(Object element) ;
    - public void add(int index,Object element);
    - public boolean addAll(int index,Collection c);
    - public object get(int Index_position);
    - public object set(int index,Object element);
    - public int size() ;
    - public int indexOf(Object o) ;
    - public int lastIndexOf(Object o) ;
    - public object remove(int index);
    - public ListIterator listIterator();
    - public ListIterator listIterator(int i);
    - System.out.println(nom_list) : affiche toute la liste sous la forme [element1, element2, …]
- listIterator permet de parcourir une liste : (il existe 'iterator' mais il ne permet pas de parcourir la liste dans le sens inverse)
    - Déclaration : ListIterator<type> nom= nom_list.listIterator() ; (*)
        - la liste doit être déjà créée et remplie , après chaque modification l'affectation précédente (*) doit être refait , sinon LinkedList jettera une exception
    - Manipulation :
        - public boolean hasNext();
        - public Object next();
        - public boolean hasPrevious() ;

```
public static void
main(String[]
args) {
    Duo<String,
Boolean> dual =
new Duo<String,
Boolean>("toto",
true);
    dual = new
Duo<Double,
Character>();
```

- public Object previous() ;

- 

La généricité :

- Le principe de la généricité est de faire des classes qui n'acceptent qu'un certain type d'objets ou de données de façon dynamique (i.e le type de donnée reçu par les méthodes d'une classe peut être préciser à l'appel de la méthode)
- <T> ou <T1,T2,..,Tn>
- Une fois qu'on déclare un objet avec un ou plusieurs types précises en utilisant la généricité, on ne peut pas affecter un nouveau objet de différent type(s) à cet objet.

  ce code est faux :

- on ne peut pas affecter une classe fille à une classe mère (covariance) si les deux sont de type générique différent

La boucle for « améliorée »

- pour les tableaux :
  - soit un tableau « tabA » de type « typeA »
    - pour parcourir le tableau on faisait :

```
for(int i=0;i<tabA.length ;i++)
        //instructions
```

- avec la boucle for « amélioré » on peut faire :

  for (int i : tabA)

  //instructions

- pour les listes :
  - Soit la liste listeA de **TYPEA (la liste doit être d'un type précis)**
    - pour que la liste soit d'un type précis (on utilise les génériques :

      Liste <TYPEA> listeA)
  - on peut parcourir la « listeA » comme suit :

    for(TYPEA el_list : listeA)

    //instructions

It's important to note that the enhanced for loop can't be used everywhere. You can't use the enhanced for loop:

- *To remove elements as you traverse collections*
- *To modify the current slot in an array or list*
- *To iterate over multiple collections or arrays*

Héritage :

- objet instanceof classe : permet de savoir si un objet est une instance d'une classe ou non
  - si on veut par exemple tester si un objet contient une référence d'une classe mère ou d'une classe fille en utilisant instanceof , il faut d'abord tester si c'est une classe fille car classe_fille instanceof classe_mère renvoie toujours vrai !
  - applying instanceof on a null reference variable returns false
- classe mere = new classe fille () : aucun probleme (cast implicite), dans ce cas on ne peut accéder qu'au méthode commune entre la classe mère et la classe fille et non pas ceux qui ne sont défini que dans la classe fille.
- classe fille =(classe fille) new classe mère () : n'est possible que si la classe mère contient une référence de classe fille (on peut utiliser instanceof pour verifier) , dans ce cas il faut faire un cast explicite.
- en affectant une classe fille à une classe mère , toutes les méthodes de la classe mère seront écraser, sauf ceux qui sont static, ils sont seulement cachés :
  - classeMere objetMere = new ClasseFille() ;
    - Appeler n'importe quelle méthode redéfinie dans la classe fille renverra celle définie dans la classe fille, sauf pour les méthodes static, ils sont seulement cachés, donc c'est la méthode de classe mère qui sera appelé

- Si une classe implémente plusieurs interfaces et hérite d'une classe supérieure, et chacune de ces interfaces définie une méthode default qui a la même signature dans toutes les interfaces, et la classe supérieure aussi défini une méthode avec la même signature, c'est cette dernière qui sera hérité par la classe fille
- Static methods in interfaces are never inherited.
- The access specifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the superclass can be made public, but not private, in the subclass.

	Superclass Instance Method	**Superclass Static Method**
Subclass Instance Method	Overrides	Generates a compile-time error
Subclass Static Method	Generates a compile-time error	Hides

- Une classe abstraite est une classe qui ne peut pas être instancier, une méthode abstraite est une méthode qui n'est pas défini (protoype seulement)
- une méthode abstraite ne peut exister que dans une classe abstraite
- public class_fille **extends** classe_mere
- la classe fille hérite tous les membres (méthodes et variables) qui sont déclarés public ou protected dans la classe mère
- this([paramètres]) appelle le constructeurs de la classe déjà défini .
- pour les constructeurs , on peut appelé super() ou super(les arguments …) dans les constructeurs de la classe fille pour appeler ceux de la mère ex :

```
public class Capitale extends Ville {
 private String monument;
 //Constructeur par défaut
 public Capitale(){
  //Ce mot clé appelle le constructeur de la classe
mère
  super();
  monument = "aucun";
 }
 //Constructeur d'initialisation de capitale
 public Capitale(String nom, int hab, String pays,
String monument){
  super(nom, hab, pays);
  this.monument = monument;
 }
```

  - le constructeur de la fonction mère que super() va appeler dépend des paramètres envoyer à super (si on donne 0 paramètre c'est le constructeur sans parametre qui sera appelé (s'il existe), si on deux 1 paramètre c'est le constructeur avec 1 paramètre(s'il existe)…)

- Attention : super([parametre(s)]) et this([parametre(s)]) ne peuvent être qu'au début du constructeur

- super.fonction() permet d'appeler une fonction définit dans la fonction mère :

```
public String decrisToi(){
   String str = super.decrisToi() + "\n \t ==>>" +
this.monument + "en est un monument";
   return str;
   }
```

- Attention à ne pas confondre la surcharge de méthode avec une méthode polymorphe.
  - Une méthode surchargée diffère de la méthode originale par le nombre ou le type des paramètres qu'elle prend en entrée.
  - Une méthode polymorphe a un squelette identique à la méthode de base, mais traite les choses différemment. Cette méthode se trouve dans une autre classe et donc, par extension, dans une autre instance de cette autre classe.

- La **covariance** des variables signifie qu'une variable objet peut contenir un objet qui hérite du type de cette variable. (objet classe mère objet classe fille)

System.out.println(objet.toString()); **Est équivalent à** System.out.println(objet);

Transformation des types primitives :

- Transformer n'importe quel objet en string en utilisant la méthode static : valueOf() :
  - String nom = String.valueOf(objet)
    - l'objet peut être int , char , double ….
  - int nom = Integer.valueOf(String a) ; (ou bien int a)
  - on peut aussi utiliser valueOf pour extraire une valeur (decimal,binaire,hexa ou octa) d'un string en précisant un deuxieme arguement à valueof qui est le radix (2, 8 , 10 ou 16) [la méthode decode fait la même chose)
- toString aussi fait la même chose sauf qu'elle doit être redéfinit dans la classe de l'objet
  - pour les classes Integer Long … on a deux méthodes toString :
    - static String toString(type nom) // type peut être Integer ,Long …
    - String toString()
- transformer un string en n'importe quelle type de variable primitive en utilisant l'ensemble des méthodes static parseTYPE_PRIMITIVE.

    - Exemple :
        - String s = "123456 "
        - int a = Integer.parseInt(s) // a= 123456
        - double b = Double.parseDouble(s) // b = 123456.0
- Integer.valueOf(char) retourne le code ASCII du caractère, donc il vaut mieux utiliser Integer.parseInt(String) (""+char = string)

String :

- String text= new(char [] array) permet de créer un nouveau string composé des caractères du tableau « array »
- indexOf permet de recherche un caractère ou un substring dans un string , retourne l'index de la premiere occurrence de ce dernier , retourne -1 s'il n'existe pas.
    - cette fonction est surchargé , on peut soit chercher depuis le début (indexOf(str)) ou bien depuis un index définit (indexOf(str, intdepuisOuChercher)
    - il y a aussi int lastIndexOf()
- String substring(int premierIndex[,int dernierIndex]) : retourne un substring depuis le premierIndex **inclut** jusqu'à la fin ou jusqu'à dernierIndex **exclut** (s'il est donnée)
- String toLowerCase() et toUperCase()
- String replace (ancien ,nouveau) : remplace l'ancien caracetere ou substring par un nouveau
- String toCharArray() pour convertir un string en tableau de caractère
    - attention le type de retour est char[] est non pas Character[] (impossible d'affecter à un tableau de character !)
- String equals(String) ; Ex : a.equals(b)
- Objects.equals(string1,string2) permet de tester si deux strings sont égaux , mais teste d'abord si l'une des string est null (si oui elle retourne NULL sinon elle appelle le equals normal)
- String equalsIgnoreCase(string)
- string1.toLowerCase() ;
- string1.contains(string2) ;
- string1.trim() : supprime les whitespaces (les espaces, les tabulations, les retour à la ligne et les retours chariots qui se trouvent au début ou à la fin d'une chaine de caracètre)

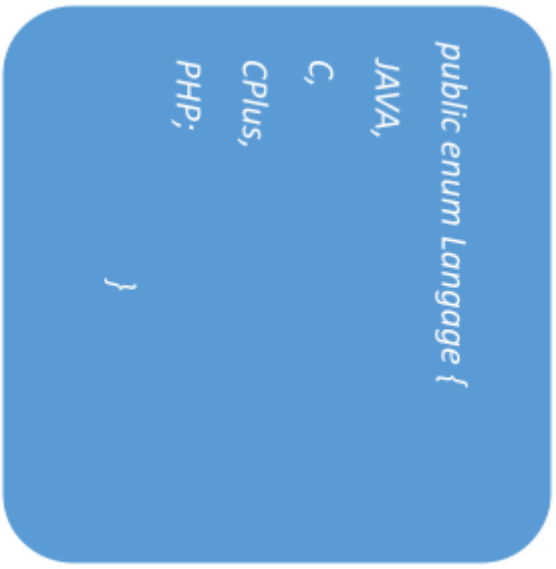
StringBuffer

- Est de taille dynamique (modifiable) et peut être modifié (au contraire de string)
- stbuilder1.setCharAt(int index,char char_to_set) ;
- stbuilder2.append(String)
- toString()
- setLength(int new_length) ;

- il n y a pas le search and replace , il faut passer par string , il y a par contre :
    - sb.replace(int i_deb_inclu,int i_fin_exclu,String string)

Enumeration:

- Une énumération se déclare comme une classe, mais en remplaçant le mot-clé class par enum. Autre différence : les énumérations héritent de la classe java.lang.Enum.
- Exemple :
- il n'y a pas de déclaration de portée, ni de type : les énumérations s'utilisent comme

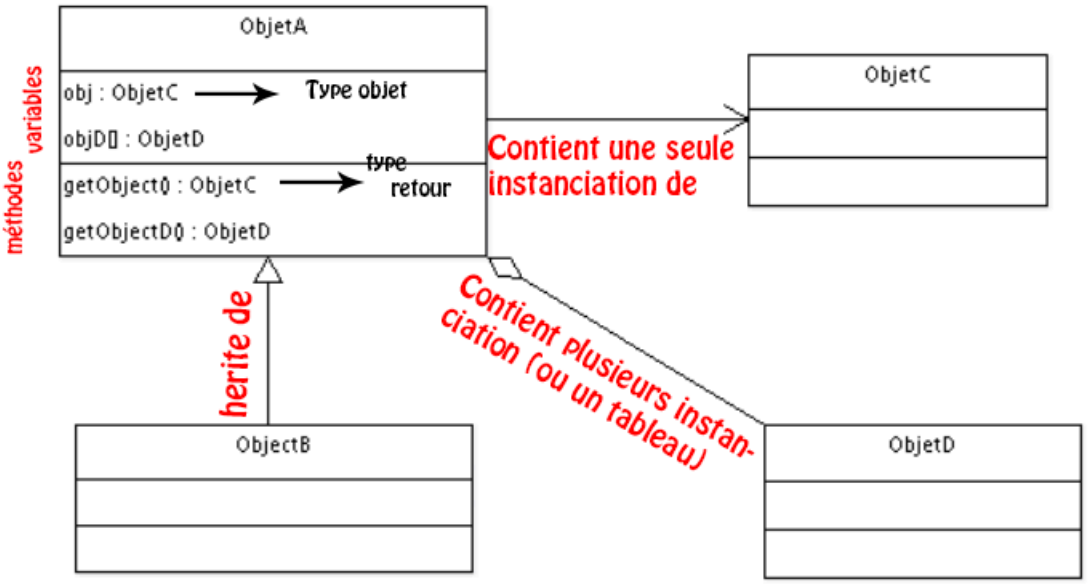


des variables statiques déclarées public

Packages :

- pour ajouter une classe à un package , il faut ajouter au début de son fichier : package nom_package ; (au début du fichier)
- pour utiliser les classes et les interfaces d'un autre package on doit d'abord l'importer comme suit : import package_name ; (au début du fichier)
- on peut faire un import static d'un package , cela nous permet d'utiliser les méthodes statiques sans les préfixé avec le nom de leurs classes mères
- on peut soit apporter une seule classe d'un package : import package_name.classe ou bien importer tout le package : import package_name.*
    - To use a public type that's in a different package, you have three choices: (1) use the fully qualified naeme of the type, (2) import the type, or (3) import the entire package of which the type is a member.
- « by convention a company uses its reversed Internet domain name for its package names »

UML (Unified Modeling Language)



Exceptions :

- try{ instructions susceptible de générer une (des) exception(s) }catch (exceptionType nom){ traitement} (on peut ajouter plusieurs catch)
    - si une exception est déclenché le reste des instructions dans le try ne sera pas exécuté
- nom_exception.getMessage() retourne un String contenant le type d'erreur
- nom_exception.printStackTrace() affiche l'erreur qui aurait était afficher si on a pas intercepté l'exception en rouge)

- `throws` : ce mot clé permet de signaler à la JVM qu'un morceau de code, une méthode, une classe… est potentiellement dangereux et qu'il faut utiliser un bloc `try{…}catch{…}`. Il est suivi du nom de la classe qui va gérer l'exception.
- on ajoute un bloc finally{ instructions } pour exécuter des instructions quoiqu'il arrive (avec ou sans déclenchement d'exception)
- `throw` : celui-ci permet tout simplement de lever une exception manuellement en instanciant un objet de type `Exception` (ou un objet hérité). Dans l'exemple de notre `ArithmeticException`, il y a quelque part dans les méandres de Java un `throw new ArithmeticException()`.
- après un throw new …() ; la méthode est quittée
- Créer une exception personnalisée :
    - class nom_class extends Exception{ définition constructeur } ;

Interfaces :

- une interface extends une autre
- une interface public doit être déclarer sur son propre fichier

- toutes les classes sont par défaut abstract et public (cela ne peut pas être modifié et on a pas besoin de le préciser)

Abstract classes :

- abstract class can not be instanciated
- une classe abstrait n'a pas de corps , elle est composé d'un modificateur , une signature et un point-virgule
- Dans une classe abstraite, on peut trouver des méthodes abstraites, c'est à dire des méthodes qui n'ont pas d'implémentation possible dans la classe abstraite, mais qui doivent obligatoirement être implémentées différemment dans toutes les classes filles
- si une classe abstraite implémente une interface, l'implémentation ou même l'écriture des signatures des méthodes n'est pas obligatoire.
- Une classe doit être abstraite s'elle contient une méthode abstraite
    - classe contient une méthode abstraite => classe doit être abstraite

with abstract classes, you can declare fields that are not static and final, and define public, protected, and private concrete methods. With interfaces, all fields are automatically public, static, and final, and all methods that you declare or define (as default methods) are public. In addition, you can extend only one class, whether or not it is abstract, whereas you can implement any number of interfaces.

~~I/O :~~

- ~~Quand on parle de «Input» et « Output » , on se réfère à l'application et non pas à l'utilisateur (au contraire de ce qu'on l'on peut croire) , donc Input c'est tout ce que sera lu par le programme (lecture depuis clavier, depuis un fichier…) et Output c'est ce que le programme sortira (affichage à l'écran, écriture dans un fichier…)~~
- ~~Chacune des classes qu'on va utiliser utilise read pour lire depuis un fichier et write pour écrire dans un fichier.~~
- ~~On doit capturer les exceptions (IOException) qui peuvent survenir (donc on doit importer aussi **java.io.IOException**)~~
- ~~Les classes de « output » crée automatiquement le fichier s'il n'existe pas, par contre les classes « input » jette une exception FileNotFoundException~~
- ~~—~~

~~Lecture byte par byte (Byte Streams):~~

- ~~il faut apporter les packages : **java.io.FileInputStream** et **java.io.FileOutputStream**~~
- ~~on doit capturer les exceptions (IOException) qui peuvent survenir (donc on doit importer aussi **java.io.IOException**)~~
- ~~On instancie la classe FileInputStream (pour lecture , ou FileOutputStream pour l'écriture) avec le nom de fichier (à lire/à modifier)~~
- ~~read renvoie -1 quand elle atteint la fin du fichier (EOF)~~

~~Lecture char par char (Character Streams):~~

- il faut apporter les packages : **java.io.FileReader** et **java.io.FileWriter**
- read renvoie -1 quand elle atteint la fin du fichier (EOF)

BufferedStream (Meilleur performance):

- lire un caractère par caractère ou byte par byte peut être couteux puisque chaque demande est gérer par le SE directement et demande l'accès au disque etc… pour cela on utilise le « buffer » , dans lequel on stock temporairement les données et on les « flush » au fichier qu'on veut une fois qu'on a suffisamment de données.
- BufferedReader et BufferedWriter permette de « Buffer » les « Character Streams »
    - FileReader name = new BufferReader(new FileReader("file_name"))
    - FileReader name = new BufferedWriter(new FileWriter("file_name"))
- Si on « buffer » un stream de caractères on peut utiliser la méthode readLine() pour lire une ligne (jusqu'à le premier \n et/ou \r

Remarques :

- On ne peut pas mettre des instructions hors les méthodes (on peut seulement déclarer des variables)
- Attention une variable déclaré dans un bloc (notamment try{}) est valable seulement dans ce bloc, c'est pour cela il ne faut jamais créer une variable dans un bloc try{}, on peut par contre la créer hors le bloc et l'initialisé dedans.
- Une classe final est une classe qui ne peut pas avoir de filles. Une classe final ne peut pas être étendue : le mécanisme d'héritage est bloqué. Mais une classe final peut évidemment être la fille d'une autre classe(non final!).
- Transtypage = cast (mère=fille implicite et fille=mère explicite et n'est possible que si la mère contient une référence de fille)
- Vector implements a dynamic array. It is similar to ArrayList, but with two differences:
    - Vector is synchronized.
    - Vector contains many legacy methods that are not part of the collections framework.

sources :
https://openclassrooms.com/courses/apprenez-a-programmer-en-java/les-exceptions