Java est similaire à C++ avec certaines choses supposées être plus simple. Il possède plusieurs librairies standard, donc beaucoup de code est déjà écrit pour nous. Typiquement un langage de programmation très léger et beaucoup utilisé pour le développement web, mobiles, etc.  
  
public class NameOfClass{} Contiendra le code de notre class. Détermine s’il sera accessible à tous les fichiers du projet ou seulement celui où ladite class a été déclaré. Cela est déterminé par le terme public ou private.   
  
Un code de sortie 0 indique qu’il n’y a pas eu d’erreur!  
  
Les packages sont entièrement en minuscules. Les class commencent par une majuscule. Les champs et méthodes commencent par une minuscule.  
  
Important : Tant que l’on ne place pas un égal suite à un attribut, sa valeur ne sera pas modifier par la méthode l’utilisant.   
  
Pour exécuter un programme à l’extérieur de notre éditeur de code, il nous faut utiliser CMD. On place en premier lieu le fichier source de la console comme la root de notre console. Ex :   
cd C:\Users\paule\Desktop\School-moment\Prog\_session\_3\Prog\Projet\l01a-bonjour\out\production\l01a-bonjour  
Ensuite on indique quel fichier il faut exécuter de type .class Ex "C:\Users\paule\.jdks\openjdk-20.0.2\bin\java.exe" Bonjour  
Cela peut aussi être fait à l’aide d’un artefact lorsque l’on possède des packages comme vu à la fin du labo l01c. Pour cela il nous faut premier lieu le créer. Dans IntelliJ IDEA, on s’assure d’avoir ouvert notre projet, ensuite on clique sur  
 File -> Project Structure -> Artifacts. On clique ensuite sur le + -> JAR -> From modules with dependencies. Pour Main class : on sélectionne le nom du fichier qui contient notre fonction main. On s’assure que le nom de la classe commence bien par le nom du package et on clique sur OK. L’artefact devrait être conserver dans le dossier \out\artifacts\ si c’est bien le cas on peut cliquer sur OK. Pour compiler l’application dans le fichier JAR on va dans le menu Build -> Build Artifacts… -> nomDuJar.jar -> Build.   
Pour exécuter l’artefact on utilisera CMD. On commence par positionner notre fichier directeur dans le fichier où se trouve l’artefact comme dans l’exemple suivant :  
cd C:\Users\paule\Desktop\School-moment\Prog\_session\_3\Prog\Projet\l01c-packages-java\out\artifacts\l01c\_packages\_java\_jar  
Ensuite on utilise la commande suivant avec le nom du fichier jar approprié à notre situation (dans ce cas -ci l01c-packages-java.jar) :  
java -jar l01c-packages-java.jar  
  
public class Bonjour {  
 public static void main(String args[]) {  
 System.out.println(‘’Bonjour le monde!’’);  
 }  
}  
Ce petit bout de code permet d’afficher la phrase « Bonjour le monde! »  
Il est important de mettre le mot public devant nos déclarations de champs ainsi que de méthodes.  
Il n’y a qu’une fonction main dans un projet en Java et celle-ci est déclaré par   
public static void main(String args[]){}  
  
Quand on compile du code en Java, il peut être exécuté sur différents systèmes d’exploitation sans être recompilé. Une JVM est employé pour exécuter les programmes java, il utilise un compilateur JIT pour y arriver et le transformer en code que la machine pourra comprendre. Un fichier Java avec l’extension .java compilé porte le nom de bytecode qui aura l’extension .class. Un JVM doit être installé pour pouvoir faire tourner des programmes .class. Généralement le JVM est installé à travers un JRE qui correspond au système d’exploitation voulu. Si on cherche aussi à développer des applications Java, on va plutôt installer un JDK ou un java development kit, qui contient lui un JRE ainsi que d’autres outils.   
  
La programmation orienté objet permet de regrouper ensemble les données d’une entité et les actions qu’on peut effectuer sur cette entité dans une même structure. Dans un langage de programmation fonctionnel, les données seraient dans des variables alors que les actions seraient des fonctions. Au contraire, dans un langage de programmation orienté objet ces choses sont regroupées.   
  
En java, un langage de programmation orienté objet, les données sont regroupées dans ce que l’on nomme des champs/attribut. Les actions, ce qui aurait été une fonction en C++, qui peuvent être effectuées sur un objet sont nommées méthodes. Une classe regroupe des champs, des méthodes et quelques autres, ceux-ci se trouvent être des membres de notre classe. Donc la définition de la structure d’un objet et le code qui y est rattaché s’appelle une classe. La classe est le modèle à suivre pour construire des objets/des instances.   
  
. la natation du point permet d’identifier l’attribut ou la méthode d’un objet. On manipule généralement les objets par l’entremise d’une variable. Fonctionne un peu de la même manière que lorsque l’on souhaite appeler certaines portions d’une structure en c++. En java, le point permet d’appeler les champs de notre objet (données), mais aussi pour nos méthodes (actions). Les objets/les instances sont associés à des classes et hérite donc de ses caractéristiques.   
  
System.out.println(‘’Text’’); Est l’équivalent d’un cout en Java. On utilise un + pour assembler des éléments à l’intérieur d’un même System.out.println();  
  
Lorsque l’on déclare un champ, il est important d’inclure si elle se trouve à être public ou private. Ex : public double prix;   
Lorsque l’on déclare une méthode, il est important d’inclure si elle se trouve à être public ou private. Ex :  
public void AfficherPrix(){}   
  
L’expression this. est utilisé comme nom place holder à la place du nom de l’objet qui appellera la fonction. Évite de la confusion, indique spécifiquement le champ.   
  
new est utilisé lors de la création d’un objet de manière à l’associé à une classe. Dans l’exemple suivant, la classe se trouve à être Modele et l’objet se nommera m1 : Modele m1 = new Modele(); Important de mettre les parenthèses après le deuxième Modele, sinon on associe m1 à l’adresse en mémoire de la classe, on ne lui associe alors pas une adresse, on fait simplement overwrite l’autre adresse.   
  
Un programme Java aura toujours besoin d’une classe principale ainsi que d’une méthode principale nommé main (en minuscule)  
  
Un Packages regroupent tous les fichiers compilés dans un fichier de type JAR, un peu comme une archive de type zip. Cela est nécessaire car toutes les classes en Java ont leur propre fichier.   
  
import nomDePackage.classe; À mettre au début du code pour importer une classe provenant d’un autre package dont on aurait besoin dans notre package. Sinon on peut aussi importer toutes les classes d’un package en utilisant la commande import nomDePackage.\*; Les packages ont pour convention d’être nommé uniquement en lettre majuscule. Les classes faisant partie d’un package possèdent une ligne au début indiquant à quel package elles appartiennent. Il suffit de la modifier pour transférer notre classe d’un package à un autre.   
  
Les modules Java sont une façon d’organiser le code en regroupement de plusieurs package. L’objectif est de rendre le code encore plus réutilisable.   
  
En orienté objet, il existe quelque chose qui se nomme sous-classe. C’est en soit une classe qui hérite des caractéristiques des membres d’une autre classe. La classe qui hérite, soit la sous-classe, peut aussi être appelé classe enfant. La classe qui fournit les caractéristique de base se fait appeler superclasse ou classe parente. Par exemple :  
public class Personne{  
 private String nom;  
}  
  
class Client extends Personne{  
 private String numeroClient  
}  
Un client aura donc un nom ainsi qu’un numéro client, alors qu’une personne n’aurait qu’un nom. Tout cela est possible grâce au mot clé extends. On peut même faire des sous-classes de sous-classes.   
  
public, private, default, protected sont tous des modificateur d’accès. On peut avoir des modificateurs d’accès pour la classe elle-même, les membres d’une classe peuvent aussi avoir des modificateurs d’accès. C’est l’accès de nos champs et méthodes dans d’autres classes et packages qui est géré par nos modificateurs d’accès. Il faut que nos modificateurs d’accès soit le plus restrictif possible tout en répondant au besoin du code. Le champ public ne devrait être utiliser que pour les constantes la majorité du temps.   
Si on suit les meilleures pratiques, les champs seront private et il faudra des méthodes public pour y accéder. Une méthode qui permet de récupérer/lire la valeur d’un champ s’appelle : méthode d’accès (getter/accessor). Une méthode qui permet de modifier la valeur d’un champ s’appelle méthode de mutation (setter/mutator). Les conventions de nom suivantes sont universellement reconnues :   
**Méthode d’accès** : getNomDeChamp  
**Méthode de mutation** : setNomDeChamp  
  
**default et public** sont les seuls modificateurs possibles pour une classe. Une classe ne peut pas être private ou protected.   
  
**private** n’est accessible qu’à l’intérieur de cette classe seulement. Il n’y a que des méthodes à l’intérieur de la classe qui pourront l’utiliser. Une méthode présente dans main ne pourrait pas l’utiliser. Main ne pourrait enfaite même pas voir cet élément.  
  
**public** est accessible à partir d’autres classes, dans main et même dans d’autres packages.  
  
**default** est accessible à l’intérieur du package où la classe se situe seulement. Donc si main se situe dans le même package que la classe il pourra l’utiliser, mais pas s’il se trouve dans un autre package.   
  
**protected** est accessible dans son package seulement, un peu comme default, ou dans une sous classe de cette classe d’un autre package. Donc si une classe de mon package possède une sous classe qui se situe dans un autre package, cette sous classe pourra utiliser mon élément protected.   
  
**foreach equivalent**  
for(Objet i : tableau) Objet étant le type contenu dans le tableau, i représentant l’élément manipulé dans le tableau. Va exécuter notre code pour tout les éléments du tableau, à part si on inclue un break.  
  
**break** permet de sortir d’un loop ou d’une fonction   
  
-------------------------------------------**Principes fondamentaux orienté objet**--------------------------------------------  
  
**Encapsulation** consiste à rassembler les données et les actions d’une entité dans une même structure et à contrôler l’accès aux détails de cette structure. Permet de cacher les détails du fonctionnement d’un objet aux autres objets. Permet de rendre des informations confidentielles non disponible à l’extérieur d’une classe. Permet de contrôler comment les données d’un objet sont traitées. Permet de contrôler comment les données sont manipulées.   
  
**Abstraction** consiste à cacher les détails d’une partie du code pour en simplifier l’utilisation. Le code requis pour chaque méthode est conservé dans la classe. La complexité du code requis pour réaliser une méthode est cachée dans la classe, on ne voit donc pas tout le code à partir du programme principal, le code se trouve ailleurs simplifiant le code principal, le rendant plus net. À pour résultat du code plus simple et mieux organisé, gestion plus facile. Réduction de la complexité du programme, au moment de la conception et du codage. Maintenance du code pus facile, on peut modifier des parties du code sans en affecter d’autres.   
  
**Héritage** permet de configurer des caractéristiques à partir d’un endroit unique et de les utiliser dans des sou-composants. Pour simplifier le code en conservant les propriétés d’une classe au moins d’endroits possibles. On profite de la réutilisation du code. Assure une cohérence entre des classes de même nature.   
  
**Polymorphisme** consiste à permettre à du code de se comporter différemment dépendant du contexte. Les classes permettent l’utilisation de méthodes qui portent le même nom, mais qui acceptent des paramètres différents. Permet d’améliorer la réutilisation du code, de simplifier le code ainsi que de fournir une interface plus uniforme.   
  
------------------------------------------------------------------------------------------------------------------------------------------  
  
**Modélisation** est de choisir comment on va former nos classes et comment elles vont interagir ensemble. L’analyse est une des étapes les plus importantes du début d’un projet de programmation. Faite avant de commencer à coder, elle permet de planifier et de distribuer les tâches et d’éviter de refaire des parties du projet. La modélisation est un des outils qui permet de faire cette identification de départ, elle consiste à exprimer la conception d’un projet dans une représentation facile à comprendre. UML (Unified Modeling Language) est un des outils qui permet de faire de la modélisation. Offre une structure standard de présenter les éléments de conception qui est utilisée et comprise par tous. Offre en autres des diagrammes pour représenter graphiquement les éléments de conception du logiciel. On va se concentrer sur le Diagramme de classes. On va créer les diagrammes à partir d’une logique de modèle conceptuel, en omettant certains des détails spécifiques pour se concentrer sur les informations et leurs relations. Les composants de nos diagrammes seront les classes, les champs, les méthodes, les liens de relations ainsi que les modificateurs d’accès. Les chiffres dans un diagramme représente la quantité de chaque chose.   
  
Les spécialisations (héritage) seront les liens de hiérarchie entre classe et sous-classe de notre programme représenté par une flèche vide.   
L’association est un lien d’interaction entre 2 classes, généralement identifié par une action représentée par une ligne.  
L’agrégation est un élément qui fait partie d’un tout, l’élément peut exister sans le tout et est représenté par un losange vide.   
La composition est un élément qui fait partie d’un tout, et ne peut pas exister sans ce tout, il fait partie de et est un composant de, sera représenté par un losange plein.   
Une image contenant diagramme, texte, ligne, Plan

Description générée automatiquement  
   
  
  
  
  
  
  
  
  
  
  
  
  
**Les variables** sont fortement typées, autrement les variables sont déclarées de la même manière qu’en c++. Les variables doit être nommée comme suit minusculeMajusculeMajusculeEtc.   
Les champs sont une forme de variable, ce sont en soit des variables d’instance, elle appartient à l’instance d’une classe.  
À l’intérieur de nos méthodes, les variables se trouvent être des variables locales.   
Les paramètres passés à une méthode sont aussi une catégorie de variable.  
  
**Retour de méthodes**  
Une méthode peut retourner un résultat ou non. S’il n’y a pas de valeur retourné la méthode doit être de type void. Si on a une valeur de retour dans notre méthode, alors on peut l’utiliser comme du code et placé le résultat dans une variable.   
Les méthodes peuvent être appelées dans les différents blocs de code, incluant à partir d’autres méthodes, d’une autre classe ou encore de la même classe.   
  
**Math**  
Math.min est une fonction de la librairie math qui va retourner le chiffre le plus petit de ses deux paramètres. Donc min(5, 6.23); retournerait la valeur 5.  
Math.pow(chiffre,exposant) est une fonction de la librairie math qui va nous permettre de faire des exposants sur nos chiffre tel que pow(3, 2) retournerait 9.   
  
**Switch**Switch fonctionne comme en c++, c’est-à-dire que l’on fait :  
switch(expression){  
 case x :  
 System.println(‘’Hello’’);  
 break;  
 case y :  
 System.println(‘’World!’’);  
 break;  
 default :  
 System.println(‘’Cas par défaut!’’);  
Il existe par contre des enhanced switch case qui ressemble plutôt à ceci :  
switch(expression) {  
 case x -> System.println(‘’Hello’’);  
 case y -> System.println(‘’World!’’);  
 default -> System.println(‘’Cas par défaut!’’);   
Ce type de switch n’utilise pas de fallthrough, c’est-à-dire que l’on a pas besoin d’inclure de break car il arrête de chercher pour des cas valide dès qu’il en trouve un. Si on veut intégrer plus qu’une ligne de code dans notre enhanced switch, il est important d’inclure {} à la suite de ->. À noter que l’on peut aussi mettre {} à la suite de : d’un switch normal.   
Peut importe le type de switch, il est possible de relier plusieurs cas au même bloc de code comme suit :  
 case x, y :  
  
**DecimalFormat**Lors de sa création, nous demande un pattern qui peut être écris comme suit ‘’#.00’’ où même ‘’#.###’’ ce qui va nous permettre de plus tard limiter le nombre de décimal présente dans notre long. On peut aussi placer des caractères à la suite ou avant notre chiffre tel que ‘’#%’’ ou même ‘’$#’’. On utilise ensuite la méthode format à l’intérieur de laquelle on place le long à formatter.   
  
**Il existe 8 types primitifs (commence par une minuscule)**:  
-bytes : entier signé 1 octet  
-short : entier signé 2 octets  
-int : entier signé 4 octets  
-long : entier signé 8 octets  
-float : réel 4 octets  
-double : réel 8 octets  
-boolean : vrai ou faux, 1 bit  
-char : caractère, 3 octets, 1 caractère/lettre ou valeur ascii (normalement un caractère prend un octet)  
  
**Type Non-primitifs (commence par une majuscule)**:  
-String, Array, Class, Interface  
-Tout autre type défini par le programmeur  
-Peut être null  
-Sont des objets (peuvent être instanciés avec new)  
  
**Conversion de types**  
-(int) permet de convertir un char en un int ayant la valeur ascii dudit char  
-(char) permet de convertir un int en un char ayant la valeur ascii dudit int  
  
**Un wrapper** est une classe prédéfinies qui offrent des fonctionnalités supplémentaires aux type primitifs. Par exemple Integer qui est le wrapper du int nous permet d’utiliser des fonctions supplémentaires tel que :  
String nombreTextuel(‘’42’’);  
Integer var = new Integer();  
var.toString(8); transforme en en string un chiffre entier  
var.Integer(nombreTextuel); transforme un string contenant un chiffre textuel en entier numérique.  
  
**Liste de wrappers** :  
-Character char   
-Byte byte  
-Short short  
-Integer int  
-Long long  
-Float float  
-Double double  
-Boolean Boolean  
  
**Méthode des wrappers** (ces méthodes ne peuvent être utiliser qu’avec des wrappers):  
.toString() Permet de convertir une donnée en chaine de caractère.  
.valueOf() Ne fonction qu’avec des String. Prend entre ses parenthèses une valeur numérique et retourne la valeur en texte.   
.intValue() Doit stocker le résultat dans un int, mais permet de convertir un Integer en int.   
.parseInt() Méthode static de Integer qui doit être stocker dans un int. Prend en paramètre un string qui est transformer en int.   
.doubleValue() Ne fonctionne que lors de l’initialisation d’un double avec new Double(aString), retourne le bool équivalent au string donner en paramètre tel Double(aString).doubleValue()  
.lenght() Ne fonctionne qu’avec des String ou des tableaux, permet de trouver le nombre de caractères de la chaine  
.compareTo() Ne fonctionne qu’avec des String, permet de comparer deux chaines de caractères pour savoir si elle sont pareille, retourne un int  
.indexOf() Ne fonctionne qu’avec des String, permet de trouver la position de la première occurrence d’un caractère dans la chaîne  
.lastIndexOf() Ne fonctionne qu’avec des String, permet de trouve la position de la dernière occurrence d’un caractère dans la chaîne.   
.charAt() Ne fonctionne qu’avec des String, permet de trouver le caractère à une position donné dans la chaîne  
.isEmpty() Ne fonctionne qu’avec des String, permet de déterminer si une chaine de caractère est vide. Retourne un bool.   
.contains() ou .matches() Ne fonctionne qu’avec des String, cherche si une chaine de caractère est présente dans le string. Fonctionne avec des Regex. Retourne un bool.   
.equals() Ne fonctionne qu’avec des String, permet de déterminer si une chaine est égal à ce qui se trouve entre les parenthèse. Retourne un bool.   
.split() Ne fonctionne qu’avec des String, permet de séparer une chaine de caractères en plusieurs chaines selon un caractère séparateur. Tel que String[] tab = ‘’asdf4pa4osi’’.split(‘’\\d’’); qui séparerait tab en un tableau de String, car le string originale aura été séparé à chaque chiffre présent dans le String.   
.intern() Ne fonctionne qu’avec des objets string, nous permet de convertir un objet string en chaine de texte littérale.   
.getText() Ne fonctionne qu’avec des String, nous retourne le contenu de la chaine de texte, si on déclare un String pour stocker l’information celui sera un objet String et non une chaine littérale.   
.instanceOf() permet de déterminer le type d’une variable   
.equals() permet de comparer deux objets, car on ne peut pas utiliser le == dans des comparaisons en java, retourne un bool   
.valueOf() permet de convertir un String qui contient de l’information numérique en int   
.isDigits() Ne fonctionne qu’avec Character, permet de vérifier si un char est un chiffre ou non  
.isLetter() Ne fonctionne qu’avec Character, permet de vérifier si un char est une lettre ou non  
.toUpperCase() et toLowerCase() Ne fonctionne qu’avec des String, permet de rendre notre chaine majuscule ou minuscule  
.getNumericValue() Ne fonctionne qu’avec Character, permet de convertir un char qui est un nombre en int  
.isEmpty() Ne fonctionne qu’avec String, permet de vérifier si un String est vide non, retourne un bool   
.trim() Ne fonctionne qu’avec String, permet de retirer les espaces vide d’un String  
  
**Modificateurs de non-accès** : ne définissent pas l’accessibilité des variables et des méthodes mais leurs fournissent des propriétés spéciales.   
-static indique qu’un champ ou méthode existe indépendamment d’une instance (pas besoin de créer un objet). Cette méthode appartient donc à la classe. Une variable membre d’une classe qui est static est appelée variable de classe. Une variable locale ne peut pas être static. Pour l’appelé il nous suffit de faire NomDeLaClasse.NomDuChamp ou même NomDeLaClasse.NomDeMethode().  
-final interdit la modification de la valeur d’une variable, méthode ou classe. Bloque l’héritage des classes. Les méthodes finales ne peuvent pas être surchargée dans une sous-classe. Une variable finale représente une constante, sa valeur ne peut pas être modifier. Une variable avec static et final est la façon typique de définir une constante pour une classe. La convention de nom est tout en majuscule et les mots sont séparés par des barres de soulignement.   
-abstract pour une classe indique qu’elle ne peut pas être insanciée, pour une méthode, indique qu’elle est sans définition, la méthode sera plutôt définie dans une sous-classe.   
-synchronized interdit l’accès à une méthode par plusieurs threads en même temps.  
-transient indique qu’un membre d’une classe ne doit pas être stocké en mémoire ou envoyé sur le réseau avec le reste de l’objet. Utilisé par exemple pour des mots de passe.   
-scrictfp assure la production d’un résultat d’opération point flottant identique sur toute plateforme  
-native permet d’écrire du code dans une méthode utilisant un autre langage que Java  
  
**Surcharge de méthodes**  
-Java autorise l’utilisation de méthodes qui portent le même nom, mais effectuent des opérations différentes. Ce comportement s’appelle Surcharge, c’est une forme de polymorphisme. Pour distinguer les 2 méthodes, il est important que les paramètres entre les 2 méthodes soient différents en type et/ou en quantité.   
-On appelle signature une expression qui identifie le nom de la méthode, le nombre de paramètres ainsi que le type des paramètres, dans une structure nomDeMethode(type1, type2…). À noter que la sifnature n’inclue ni le type du retour, ni le nom des paramètres, ni le bloc de code de la méthode.   
Ex : retirer(int, int) //// retirer(string, string)  
-Pour permettre la surcharge, les méthodes doivent porter le même nom, avoir une signature différente et retourner le même type.   
  
-Pour utiliser un float en java, il est important de mettre une ‘f’ après notre nombre.   
  
**Constructeur**  
-NomDeMaClasse() est une méthode constructeur. Ce constructeur doit porter le même nom que la classe. Peut avoir ou non des paramètres. Ne peut pas avoir de valeur de retour. Cette méthode est défini à l’intérieur de ladite classe.   
-Jusqu’à maintenant, n’a pas défini de constructeurs dans nos classes, mais on créé quand même des objets avec le mot clé new. Si on ne définit pas de constructeur, le compilateur en ajouter/créé un automatiquement. Ce constructeur, définit par le système, n’a pas de paramètres et s’appelle constructeur par défaut.   
-On utilise le constructeur pour accomplir des tâches dès la création d’un objet. Par exemple, donner une valeur à un champs dès la création de l’objet.   
-En tant que méthode, un constructeur peur recevoir des paramètres. Ces paramètres s’utilisent comme avec les autres méthodes. On peut aussi faire plein d’opération dans notre constructeur, si c’est logique de le faire à la création de chaque objet du même type que notre classe.   
-Utilisé une méthode constructeur, nous permet de sauver du code. Par exemple, on n’a pas besoin de faire de setter à la création de l’objet avec ce type de méthode.   
-Les constructeurs supportent la surcharge. Les mêmes règles que pour les autres méthodes s’appliquent.   
-Pour utiliser la surcharge de constructeur et utilisez le constructeur par défaut sans paramètres, le constructeur par défaut doit être défini explicitement.   
-La meilleur pratique est de toujours définir explicitement au moins un constructeur par classe.   
  
**Expression Régulière (Méthodes pour les expressions régulières : import java.util.regex.\*)**- Est une séquence de caractères qui décrit un motif/modèle de caractères à rechercher dans une chaine ou comparer avec une chaine.   
-[] des caractères entre crochets signifient n’importe quel des caractères mentionnés. Les crochets représentent des options. Ex : unChaine.equals(‘’saut[ea]’’); La comparaison sera vraie pour saute et sauta.   
-‘’ ‘’ Si on utilise une expression régulière dans un string, il va falloir que le string qui est comparé respecte le pattern exact du string, il doit respecter le modèle.   
-[abc] a, b ou c  
-[^abc] Tous les caractères sauf a,b ou c (négation)  
-[a-zA-Z] a à z ou A à Z (ensemble)  
-[a-d[m-p]] a à d, ou m à p, est équivalent à [a-dm-p] (union)  
-[a-z&&[def]] d,e ou f (intersection)  
-[a-z&&[^bc] a à z, sauf b et c, est équivalent à [ad-z] (soustraction)  
-[a-z&&[^m-p]] a à z, mais pas m à p, est équivalent à [a-lq-z] (soustraction)  
-. Tout caractère  
-\d Un chiffre [0-9]  
-\D Tout sauf un chiffre [^0-9]  
-\s Un caractère blanc [\t\n\x0B\f\r]  
-\S Tout sauf un caractère blanc [^\s]  
-\w Un caractère, lettre ou chiffre [a-zA-Z\_0-9] (quand on dit caractère ici, on ne comprend que les \_)  
-\W Tout sauf un caractère, lettre ou chiffre [^\w]  
-X? Où X est un caractère qui apparait 0 ou 1 fois seulement  
-X\* Où X est un caractère qui apparait 0 ou plusieurs fois  
-X+ Où X est un caractère qui apparait 1 ou plusieurs fois  
-X{n} Où X est un caractère qui apparait exactement n fois  
-X{n,} Où X est un caractère qui apparait au moins n fois  
-X{n,m} Où X est un caractère qui apparait au moins n fois et au plus m fois.   
-{n} Où n est le nombre d’instance exact du token précédent qui devront être présent dans le string (si on utilise l’expression \\d{4} on devra retrouvé exactement 4 chiffres et aucun autre type de caractère)  
-() permettent d’associer une expression à un paramètre, tel que le nombre d’apparition (ex : (.\\d+)?)  
-Il est possible de faire un if sans utiliser d’expression conditionnel dans un regex. Pour se faire la syntaxe est la suivante : ((expression1)(expression2 à faire si expression1 est présente)|(expression3 à faire si expression 1 est absente)  
**\*\*\*attention en Java, \ est un caractère d’échappement. Pour utiliser les modèles de caractères, on doit donc utiliser \\.  
  
Pattern et Matcher**Les classes Pattern et Matcher servent à manipuler des expressions régulières. Il faut importer le package javax.util.regex pour utiliser ces classes. matches de la classe String imite en tous points celle de la classe Pattern.   
Pattern représente une version compilée d’une expression régulière.Elle n’a pas de constructeur; on crée un objet Pattern à l’aide de la méthode statique compile : Pattern p = Pattern.compile(‘’\\d{5}’’);  
Matcher permet d’obtenir un objet qui interprétera le Pattern. Comme Pattern, on ne peut pas créer un objet Matcher avec un constructeur, on doit utiliser la méthode matcher de la classe Pattern :  
Matcher m = p.matcher(‘’12345666633’’);  
m.matches() 🡪 faux car 12345666633 ne correspond pas à 5 chiffres.  
m.find() 🡪 vrai car tu peux trouver 5 chiffres dans 12345666633  
m.find() 🡪 vrai car tu peux trouver 5 chiffres à la suite de l’autre 5 chiffres  
m.find() 🡪 faux car il ne reste qu’un chiffre  
m.reset() 🡪 Retourne au début de la chaine de caractère  
  
Les classes StringBuffer et StringBuilder offrent une série de méthodes conçues spécifiquement pour construire des chaines de caractères.   
  
**Scanner**La classe Scanner qui est présent dans le package java.util. Permet de scanner tout un fichier plutôt que seulement une String. Permet de retourner tout type prédéfini plutôt que seulement des Strings. Les délimiteurs par défaut sont les caractères blanc (espace, \r, \n, etc…) mais on peut utiliser une expression régulière à la place.  
Scanner scanner = new Scanner(System.in) nous permet de créer un objet scanner qui va nous permettre d’attendre un input de l’usager avant de continuer le code. scanner.nextLine() nous retourne sous forme de String l’input de notre usager.   
  
**Objets et new**  
Quand on créé un objet grâce à new, cet objet existe alors en mémoire vive à une adresse mémoire donnée. La syntaxe Message objMessage = new Message() va alors crée un objet de la classe Message, stocké en RAM à une adresse mémoire donnée, alors que la variable objMessage contient l’adresse mémoire pour accéder à l’objet. On est pas obligé de stocker en mémoire l’adresse pour appeler les méthodes d’une classe ainsi que new Message().afficherMsg(); va appeler la méthode afficherMsg de la classe Message mais on aura plus accès à cette objet par la suite.   
Il nous faut distinguer si on parle de deux objets ou de deux références à un objet; des références étant des adresses indiquant où se trouvent les variables et les méthodes d’un objet donnée. Donc : Point p1, p2; p1 = new Point (100, 99); p2 = p1; p2 est une référence à la même adresse que p1, donc si on modifie p2, on modifie aussi p1. Si on fait p1 == p2, alors on aura true, car il possède la même référence. Alors que Point p3 = new Point (15, 15); Point p4 = new Point (15, 15); p3==p4 va nous retourner false car il ne sont pas des références à la même adresse en mémoire.   
.equals Nous permet de comparer deux objets entre eux, vérifiant que toutes les variables d’instance sont égales. Si le type des objets à comparer ne définit pas la méthode equals, on doit la redéfinir (la coder) nous-mêmes. Dans le fond si on crée nos propres classes, il nous définir nous-même equals. Un objet va appeler la méthode qui va prendre en paramètre un objet du même type et va ensuite comparer les variables d’instance des deux objets.  
String est particulier, car il est mi-prédéfini, mi-objet. Le choix d’utiliser == ou equals pour les comparer réside dans la définition des Strings. Si on a deux chaines littérales, soit String a = ‘’allo’’ et String b = ‘’allo’’, alors on peut faire ==. Si on a une chaine littérales et un objet String b = new String(‘’allo’’);, alors il nous faudrait utiliser equals tel que a.equals(b) nous retournerait vrai. On peut transformer un objet en chaine littéral grâce à .intern(). Si on utilise String b = champTexte.getText(); on aura créé un objet String et non une chaine de texte littérale. À savoir que les méthodes des différents types crée généralement des objets tel que charAt().   
  
**Création/initialisation d’un tableau de 3 façons :**  
Lorsqu’on initialise un tableau sans donner de valeurs aux éléments, des valeurs par défaut sont automatiquement assignées. Ces valeurs dépendent du type des éléments : boolean est false, int est 0, double est 0.0, String est null, Objet ou type défini par l’utilisateur est null.  
On doit déclarer et initialiser pour qu’on puisse y assigner des valeurs.   
Un tableau a une grandeur fixe, on ne peut pas dépasser la capacité indiquée au départ.  
1. int[] tableau = new int[12]; Déclaration et initialisation en 1 ligne.  
2. int[] tableau; Déclaration d’une variable d’instance.  
 tableau = new int[12]; Initialisation du tableau. Serait fait dans un constructeur par exemple. L’initialisation d’un tableau à une variable déjà déclarée doit utiliser le mot new.   
3. int[] tableau = {3, 4, 5, 6, 2, 0, 0, 9}; Déclaration, initialisation et assignation de valeurs.   
  
**Tableaux d’objets**  
On doit créer le tableau et les objets faisant partie du tableau. Il faut initialiser en premier lieu le tableau, et ensuite chaque élément qui sera dans le tableau.   
LocationFilm[] tab = new LocationFilm[3];  
tab[0] = new LocationFilm(‘’Forrest Gump’’);  
tab[1] = new LocationFilm(‘’End Game’’);  
tab[2] = new LocationFilm(‘’Alvin et les chimpmunks’’);  
///////////////////////////////////////////////////////////  
Paiement tableau5 [] = new Paiement [3]  
tableau5[0] = new Paiement(1,4,0); tableau5[1] = new Paiement(0,3,0);tableau[2] = new Paiement(1,0,0);  
tableau5[2].getNbreDeux() fonctionne car à cette emplacement dans le tableau il y a un objet de type paiement qui possède la méthode getNbreDeux().  
  
**Tableau Multidimensionnel**  
int[][] tableau = new int[2][4];  
int tableau4[][] = {{2,3},{3,4},{3,9}};  
tableau4[2][1] = 3; tableau4.length = 3; tableau4[1].length = 2  
  
**La valeur null**  
Le mot clé null indique l’absence de valeur. On ne peut pas assigner null à une variable de type primitif. On peut assigner null à une référence ou un pointer, sa signification est que cette variable n’indique aucune adresse mémoire. Parce qu’une variable d’objet ou de tableau est une référence, java autorise l’assignation de null à ces variables. On ne peut pas passer une référence null en paramètre à une méthode. Parce que la valeur null peut causer des exceptions dans plusieurs situations, on l’utilise seulement lorsque d’autres moyens ne sont pas disponibles.  
  
**Déclaration, définition et initialisations d’objets**  
-La déclaration consiste à indiquer qu’une variable existe (je déclare qu’il y aura une variable X de type Y)  
- Définir une variable implique de donner une valeur à ladite variable X  
- Initialiser une variable consiste à la définir pour la première fois, lui donner une valeur de départ. Lors de la modélisation d’une classe, la convention est de déclarer les champs sans les définir, puis de les définir dans le constructeur ou dans une méthode. C’est le rôle du constructeur de créer l’objet et toutes les initialisations de valeurs se retrouvent alors à un endroit.  
  
**Traitement de dates et heure**  
Les API’s Date et Time sont des classes disponibles dans le package java.time.\*  
On travaille avec 6 classes principales :  
-LocalDate nous retourne la date sans l’heure – selon le fuseau horaire local (local zone)  
-LocalTime heure sans la date – fuseau horaire local (local zone)  
-LocalDateTime date et heure – fuseau horaire local (local zone)  
-ZoneDateTime date et heure – manipulation d’autres fuseaux horaires (zones)  
-Period intervalle de temps basé sur de composants de dates (années, mois, jour)  
-Duration intervalle de temps basé sur des unités d’heures, minutes, seconds, nanosecondes…  
Exemples :  
-LocalDateTime uneDate = LocalDateTime.of(2006, 03, 22, 21, 15, 36); Est une méthode static. Le 22 mars 2006 à 9h15 minutes et 36 secondes PM  
-LocalDateTime uneDate = LocalDateTime.parse(‘’2006-03-22T21:15:36’’); Est une méthode staitc. Parse peut être utilisé au lieu de of si la date est disponible en format texte, et si cette chaine respecte le format date/heure demandé  
-LocalDateTime uneDate = LocalDateTime.now(); Est une méthode static. La méthode .now créé une variable avec la date et l’heure actuelle  
-LocalDateTime uneDate = LocalDateTime.now(); println(uneDate.plusDays(3)); Affichera maintenant + 3 jours. uneDate n’a pas changé, un LocalDateTime est immuable. Si on veut stocker en mémoire, il nous faut réassigner notre variable à la nouvelle valeur, ou mettre la nouvelle valeur dans une autre variable.   
-LocalDateTime uneDate = LocalDateTime.now(); println(uneDate.minusDays(3)); Affichera maintenant – 3 jours.  
-Aussi disponible : .plusNanos() .plusSeconds() .plusMinutes() .plusHours() .plusDays() .plusWeeks()   
.plusMonths() .plusYears() et toutes les méthodes ci-dessus en version minus. Ces méthodes gèrent automatiquement le parcours sur plusieurs mois, années, gère les années bissextiles, etc…  
-Des méthodes d’accès (getters) sont disponibles pour extraire un des composants d’un Date/Heure :  
int jourDuMois = uneDate.getDayOfMonth(); Retourne un entier qui est le jour du mois de la variable LocalDateTim. D’autres méthodes sont disponibles pour tous les composants de dates incluant même le jour de la semaine, ils retournent parfois autre chose que des int… à vérifier selon la méthode getter  
-On utilise ZoneDateTime lorsque la notion de fuseau horaire est importante dans une application :  
ZoneDateTime now = ZoneDateTime.now(ZoneId.of(‘’GMT+5 :30’’)); Il y a plusieurs façons de traiter les ZoneId. La classe ZoneDateTime a plusieurs méthodes particulières.   
-La classe **Period** permet de gérer un intervalle de temps en années/mois/semaines/jours :  
Period unePeriode = LocalDateTime.of(4,1,2); Une période de 4 ans, 1 mois et 2 jours. Pas le 2 janvier de l’an 0004!  
unePeriod.getDays(); Retourne le composant jour de unePeriode. Dans l’exemple, c’est 2. Il y a des méthodes .getYears(), .getMonths() etc…  
.plusDays(x), .plusMonths(x), .plusYears(), ces méthodes ajoutent à l’intervalles une quantité x correspondante. Si x est négatif, c’est l’équivalent de faire une soustraction.   
-La classe **Duration** permet de gérer un intervalle de tmpes en heures/minutes/secondes/nanosecondes :  
.between va créer un intervalle de temps entre 2 date/heures.   
LocalDateTime maintenant = LocalDateTime.now(); LocalDateTime dans3Jours = maintenant.plusDays(3);  
Duration intervalle = Duration.between(maintenant, dans3Jours);  
println(intervalle.toDays()); va retourner l’intervalle de temps entre maintenant et dans3Jours en jours  
println(intervalle.toHours()); va retourner l’intervalle de temps entre maintenant et dans3Jours en heures  
println(intervalle.toMinutes()); va retourner l’intervalle de temps entre maintenant et dans3Jours en minutes.  
-La classe **ChronoUnit** possède une méthode static permettant de mesurer le nombre de jours/mois/années/siècle etc entre deux LocalDate ou même LocalDateTime (retourne un long):  
long nbJours = ChronoUnit.DAYS.between(date1, date2);  
  
  
  
**DateTimeFormatter**  
Est une classe disponible à partir de la librairie java.text.\*  
Le champ principal d’un objet DateTimeFormatter est une chaine de caractère qui représente le format d’affichage lui-même (un modèle, comme un regex mais pour les dates)  
Exemple :  
DateTimeFormatter dtf = DateTimeFormatter.ofPattern(‘’yyyy-MM-dd’’);  
On utilise ensuite la méthode .format de l’objet DateTimeFormatter pour créer une chaine de caractère dans le format spécifié, en lui passant l’objet de date/heure désiré :  
LocalDate noel2001 = LocalDate.of(2001,12,25);  
System.out.println(dtf.format(noel2001)); Fera affiché la date selon le format indiqué   
Voici une liste des symboles approuvés pour créer un modèle (documentation : <https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html>) :

|  |  |  |  |
| --- | --- | --- | --- |
| **Symbole** | **Explication** | **Présentation** | **Examples** |
| y | Année de calendrier | Année | 2004, 04 |
| Y | Année de la semaine | Année | 2004, 04 |
| M | Mois de l’année | Nombre/texte | 7, 07, Jul, July, J |
| w | Semaine de l’année | Nombre | 27 |
| W | Semaine du mois | Nombre | 3 |
| D | Jour de l’année | Nombre | 189 |
| d | Jour du mois | Nombre | 10 |
| H | Heure de la journée (0-23) | Nombre | 0 |
| m | Minute de l’heure | Nombre | 30 |
| s | Secondes de la minute | Nombre | 5 |
| e | Jour de la semaine | Nombre/texte | 2; 02; Tue; Tuesday; T |

LocalDateTime uneDate = LocalDateTime.of(2016, 2, 14, 16, 1, 1);

|  |  |
| --- | --- |
| **Pour une valeur de modele de:** | **L’affichage sera:** |
| "yyyy-M-dd" | 2016-2-14 |
| "yy-MM-dd" | 16-02-14 |
| "yyyy/MMM/dd" | 2016/févr./14 |
| "Date: yyyy-MMMM-dd" | Exception... Unknown pattern letter: t |
| "'Date: 'yyyy-MMMM-dd" | Certains caractères comme – et / sont acceptés, mais si on veut écrire du texte, il vaut encadrer avec des apostrophes. |
| "'Date: 'dd MMMM yyyy 'Heure: 'HH:mm" | Date: 14 février 2016 Heure: 16:01 |

**Tests Logiciels**  
-Ils permettent de s’assurer que l’application adopte le comportement attendu, de détecter des comportements indésirables et ils doivent être fait avant la publication de l’application.   
-Un test unitaire consiste à tester une petite portion du code, une seule fonctionnalité ou partie d’une fonctionnalité. En java, c’est souvent la vérification d’une seule méthode. On peut faire des tests de façon manuelle, donc on essaie les méthodes avec différentes valeurs et on vérifie si les résultats correspondent aux attentes. Ou bien de façon automatisée, donc on va utiliser des outils spécialisés qui effectueront des tests automatiquement.   
-JUnit est un framework conçu spécifiquement pour faire des tests unitaires automatisés sur du code Java. Les étapes suivantes permettent de mettre en place des tests JUnit :  
1. Ajouter les librairies (dépendances) JUnit au projet (File 🡪 Project Structure 🡪 Librairies 🡪 + 🡪 From Maven 🡪 org.junit.jupiter:junit-jupiter-params:5.8.1 🡪 OK-OK-OK)  
2. Créer un répertoire racine de tests qui doit être au même niveau que le src. (Faire un clique droit sur le fichier du projet, créer un nouveau dossier. Faire un clique droit sur le nouveau dossier et sélectionner Mark Directory as et sélectionner Test Sources Root)   
3. Créer une classe de test dans le dossier créé précédemment. La classe devrait avoir le nom classATesterTest, donc blankTest.   
4. Importer les classes requises de la librairie JUnit (import org.junit.jupiter.api.Test; et import static org.junit.jupiter.api.Assertions.\*; et importer les classes que l’on veut tester.)  
5. Ajouter des méthodes de tests (On met @Test à la ligne avant chaque test, notre méthode de test doit toujours être public void et son nom doit commencer par le mot test. On crée un objet de la classe que l’on test, ensuite on utilise la méthode de junit assertEquals() qui prend en premier paramètre le résultat que l’on veut obtenir, et en deuxième paramètre la méthode que l’on veut tester pour s’assurer de son résultat. On peut faire plusieurs tests dans la même méthode, le problème est qu’on ne sera pas lequel des tests a échoué.)  
6. Exécuter et interpréter les résultats  
-Pour tester une méthode qui ne retourne rien, on doit valider le résultat à l’aide d’une autre information. Par exemple, si la méthode modifie un champ, on peut vérifier que la bonne valeur est assignée à ce champ.   
-Pour tester des champs ou méthodes private, on ne peut pas directement. Il n’est pas recommandé de changer le modificateur d’un champ pour les tests, on désire plutôt tester les méthodes qui utilisent ladite méthode private et vérifier que les résultats sont ceux attendus. En dernier recours, s’il n’y en a pas déjà une, on peut ajouter une méthode d’accès publique (getter) avec des commentaires « Pour tests seulement » et s’assurer que ces méthodes ne seront pas publiées avec le code final.   
-On peut ajouter du code qui sera exécuté avant ou après les tests. Ce code doit aussi être dans une méthode. @BeforeAll – code exécuté avant tous les tests. @BeforeEach – code exécuté avant chaque test. @AfterAll – code exécuté après que tous les tests soient terminés. @AfterEach – code exécuté après chaque test. Le @BeforeAll et @AfterAll sont utilisés pour préparer l’environnement à exécuter les tests. Une utilisation typique est la connexion à une base de données et la fermeture de cette connexion à la fin. D’autres exemples incluent l’ouverture/fermeture de fichiers, réservation de mémoire etc… Les méthodes associées au @BeforeAll et @AfterAll doivent être déclaré comme static, ce qui limite l’utilisation de variables déclarées dans son code. On peut contourner ce comportement en ajoutant, au début du fichier de classe : @TestInstance(Lifecycle.PER\_CLASS). Une utilisation typique du @BeforeEach est la création d’objet, au lieu d’inclure celui-ci dans la méthode de test. Pour plus d’information sur JUnit : <https://junit.org>  
-Quelles valeurs doit-on tester dans une suite de tests?: Dépend de la fonctionnalité de la méthode. On devrait tester des valeurs qui fonctionne et vérifier qu’on obtient le résultat désiré. On devrait tester des valeurs qui ne sont pas autorisées et vérifier que le programme gère ces valeurs tel qu’attendu. On devrait tester les valeurs limites, de très grandes valeurs ainsi que de très petites valeurs. On devrait tester des valeurs absentes (null, 0, String vide, etc…). On devrait tester le comportement en absence de données de base (fichiers, bases de données non disponibles). Etc…  
Ps : Voir le projet D03A pour avoir un exemple codé.  
  
**Méthode de Junit**  
-assertEquals() prend en premier paramètre le résultat que l’on veut obtenir, et en deuxième paramètre la méthode que l’on veut tester pour s’assurer de son résultat.  
-assertTrue() prend en paramètre la méthode que l’on veut tester pour s’assurer que son résultat est true.  
-assertFalse() prend en paramètre la méthode que l’on veut tester pour s’assurer que son réfultat est false.  
  
**Super-classe (Parent) et Sous-classe (Enfant)**Une sous-classe hérite des champs et méthodes de la super-classe en y ajoutant d’autres membres (champs et méthodes). Les points communs à toutes les sous-classes seront présents dans la super-classe. L’avantage principal est la réutilisation du code.   
Le principe fondamental d’héritage est ce qui permet à la sous-classe de recevoir les champs et méthodes de sa super-classe. Les méthodes et champs peuvent être héritées.   
Une superclasse est déclarée comme une classe normale, c’est la présence de sous-classe qui en fait une super-classe.   
Les champs private d’une superclasse ne sont pas modifiable directement par la classe enfant, si on en a, il faut utiliser des getters et des setters pour avoir accès à l’information. Il est plus commun d’utilisé protected dans la superclasse.  
Une méthode qui prend en paramètre un objet de la classe parent dans sa définition acceptera aussi un objet de la classe enfant lorsque mis en paramètre lors de l’appel de ladite méthode. Elle acceptera aussi toujours l’objet de la classe parent tant que ladite classe n’est pas abstraite ou une interface. Si une méthode a été redéfini dans la classe enfant, la méthode utilisera la méthode redéfinie.   
  
-extends indique qu’une sous-classe hérite d’une super-classe, au moment de sa déclaration. Tel que   
public class nom\_sous-classe extends nom\_super-classe. Une sous classe ne peut hériter que d’une super-classe.   
-super vient remplacer le nom de la classe parent et est l’opposé de this. Indique clairement que cet élément est hérité. Est surtout utile si la classe parent et enfant ont des champs ou méthode portant le même nom. Optimalement on veut par contre éviter que deux méthodes ou champs d’une classe parent et une enfant portent le même nom, on préfère utiliser des noms uniques.   
Ex :  
public class Animaux{  
 protected String description;  
}  
public class Mammiferes extends Animaux{  
 private String description;  
  
 public void afficherInfo(){  
 System.out.println(‘’La description de la classe animaux : ‘’ + super.description)  
 System.out.println(‘’La description de la classe mammifère : ‘’ + this.description)  
 }  
}  
  
On peut faire de la surcharge de méthodes incluant des méthodes définies dans la superclasse ET dans la sous-classe. Les règles habituelles s’appliquent : les méthodes doivent porter le même nom et avoir une signature différente.   
  
Les constructeurs ne sont pas hérités d’une superclasse. C’est une méthode qui porte le nom de la classe, il ne peut donc pas être le même dans la sous-classe. Mais il peut être appelé à partir d’une sous-classe.   
Ex :   
  
public class Animaux{  
 public Animaux() {};  
}  
public class Mammiferes extends Animaux{  
 public Mammiferes(){  
 super();  
 }  
}  
  
Redéfinition de méthodes(Overriding)  
Comme la surcharge (overloading), la redéfinition (overriding) permet de créer des méthodes qui portent le même nom, mais qui ont une fonctionnalité différente. Contrairement à la surcharge, redéfinition ne requiert pas que les signatures soient différentes. Cependant, on ne peut pas utiliser la redéfinition à l’intérieur d’une même classe, c’est un mécanisme réservé à l’héritage (sous-classe, classe abstraite, interface).   
Une méthode redéfinie est une forme de polymorphisme. Elle utilise l’annotation @Override. Elle doit être redéfinie dans une sous-classe et doit avoir la même signature ainsi que le même type de retours que la classe parente (ou un sous-type). Elle doit avoir un modificateur d’accès avec le même accès ou plus d’accès, mais jamais moins que la méthode de la classe parent. On ne peut pas redéfinir une méthode parente private, static ou final. Ça ne fonctionne pas avec les constructeurs. Il est obligatoire de faire de la surcharge pour les méthodes abstraites et les interfaces. Si une méthode est utilisé dans le constructeur super et qu’elle est plus tard override par un enfant, si l’enfant fait appel au constructeur parent, ce dernier va utiliser la méthode override de l’enfant.   
Ex :  
public class Produit{  
 private String description;  
 protected double prix;  
 public double calculerPrix(){  
 return prix \* 1.14975;}  
}  
public class Portable extends Produit {  
 public static final double ECOFRAIS = 0.20;  
 @Override  
 public double calculerPrix(){  
 return (prix + ECOFRAIS) \* 1.14975;}  
}   
  
  
**Classes abstraites**C’est une classe parent qui ne peut pas être instanciée, on ne peut donc pas créer d’objets de cette classe. Tous ses membres sont destinés à l’héritage. Une classe abstraite peut avoir des méthodes abstraites et non abstraites (concrètes).  
Une méthode abstraite est une méthode qui n’a pas de bloc de code, elle est obligatoirement définie dans une sous-classe.   
Pour choisir si une classe parent sera abstraite, il faut en premier lieu déterminer si on devra créer des objets à partir de cette classe.   
  
Une classe abstraite utilise le mot abstract, elle peut avoir un constructeur, elle ne peut pas être instanciée, elle peut avoir des méthodes concrètes ou abstraites et elle peut utiliser les champs static non-static final et non final.   
  
***//Un objet Produit ne peut pas être créé  
//La classe abstraite sert à fournir des membres communs aux sous-classes***

**public abstract class Produit{**

***//Une constante commune à toutes les sous-classes* static final double *taxeTPS* = 0.05;  
 *//Tous les produits ont un prix mais il n'est pas le même* protected double prix;  
 *//Tous les produits ont un nom mais il n'est pas le même* protected String nom;  
  
 *//Produit.getTauxTPS est accepté parce static => pas besoin d'instance* static double getTauxTPS(){  
 return *taxeTPS*;  
 }  
  
 *//Méthodes abstraite, redéfinie pour chaque produit  
 //Pas de bloc de code  
 //tous les produits ont des spécifications,  
 //mais elles ne sont pas les même pour chacun (écran a une résolution, disque dur a une taille)* abstract public void afficherSpecifications();  
}  
  
public class Ecran extends Produit{  
 *//Pas besoin de redéclarer prix ou nom, déjà déclaré dans la classe Produit* private double resolution; *//Existe pour les ecrans mais pas pour d'autres produits  
  
 //Constructeur pour cette sous-classe, inclus la résolution* public Produit(double pPrix, String pNom, double pResolution){  
 prix = pPrix;  
 nom = pNom;  
 resolution = pResolution;  
 }  
  
 *//Implémentation de la méthode déclarée dans la classe abstraite* public void afficherSpecifications(){  
 System.*out*.println("Prix: " + prix);  
 System.*out*.println("Nom: " + nom);  
 System.*out*.println("Resolution: " + resolution);  
  
 }  
  
}**  
  
Les méthodes abstraites n’ont pas de bloc de code, elle sont obligatoirement définie dans une sous-classe et ont pour objectif de s’assurer que les sous-classes définissent ces méthodes avec une signature uniforme. Elle requiert une classe abstraite, elle utilise le mot abstract, elle n’a pas de bloc de code et elle renvoie à des méthodes des sous-classes.   
On déclare une méthode abstraite dans la classe parent sans bloc de code et utilise ; et non {}  
On définit la méthode dans la classe enfant avec son bloc de code. Pour un exemple, aller voir la méthode afficherSpecifications() de l’exemple précédent.   
  
**Interface**Ce sont des classes qui ne peuvent pas être instanciée, on ne peut pas créer d’objets.   
On utilise des interfaces pour garantir qu’une méthode sera mise en place par toutes les sous-classes, avec un nom uniforme.  
On utilise aussi les interfaces pour exposer des champs et des méthodes à d’autres classes en gardant le contenu des sous-classes cachées.   
C’est un mécanisme utilisé entre autres lorsque des équipes de développeurs différentes travaillent sur des entités communes.   
On utilise aussi les interfaces pour définir des champs ou des méthodes pour des classes qui n’ont pas de lien logique entre elles. Par exemple, une classe Enregistrement pourrait avoir une méthode enregistrer() utilisée par des classe Musique, Vidéo, Conférence, etc… Ces classes ne sont pas des sous-catégories d’un ensemble Enregistrement. Cette façon de faire permet de créer des méthodes avec une syntaxe universelle pour des classes différentes.   
Une interface utilise les mots clés interface et implements, elle ne peut être instanciée, elle ne peut avoir de champs sauf des champs static et final, tous ses membres sont public par défaut, une sous-classe peut hériter de plusieurs interfaces ce qui n’est pas le cas des super-classes et classes abstraites, une sous-classe peut hériter d’une super-classe/classe abstraite et implémenter des interfaces, elle ne peut pas avoir de méthode concrète que des méthodes sans bloc de code.   
  
***// Une interface est une classe nécessairement public***

**public interface Taxable{**

***// Les méthodes d’une interface n’ont pas de code   
 // Notez l’absence de { } qui sont remplacées par un ;* public float *calculerTaxe*(float prix);  
  
}**

**class NoisettesAuChocolat extends NonPerissable implements Taxable,Pesable{  
 *//Mot clé implements pour les interfaces***

***//Plusieurs interfaces peuvent être implémentées par une classe***

***//Une classe peut hériter (extends) d’une classe ET implémenter une/des interface(s)***

***//Implémentation de la méthode de l’interface Taxable* public float calculerTaxe(float prix){  
 return prix \* 0.15975;  
 }**

**...  
  
}**