|  |  |
| --- | --- |
| **Département d'informatique**  420-201-RE - Introduction à la programmation | **Note de cours 201A22 - notes de cours - ra - V01.docx** |

Table des matières

[Glossaire des termes utilisés dans le cours 5](#_Toc111987013)

[Introduction 6](#_Toc111987014)

[Qu’est-ce qu’un programme informatique? 6](#_Toc111987015)

[Comment passer d’un programme source à un programme exécutable par la machine? 8](#_Toc111987016)

[Environnements de développement 8](#_Toc111987017)

[Bases de programmation en Java 9](#_Toc111987018)

[Une instruction Java 9](#_Toc111987019)

[Les lignes blanches 9](#_Toc111987020)

[Les commentaires 9](#_Toc111987021)

[Commentaires de fin de ligne 9](#_Toc111987022)

[Commentaires sur plusieurs lignes 10](#_Toc111987023)

[Instruction d’affichage à la console 10](#_Toc111987024)

[Opérations simples sur des variables numériques (**+**, **-**, **\***, **/**) 11](#_Toc111987025)

[Patron de programme pour les premières semaines 12](#_Toc111987026)

[Les variables 13](#_Toc111987027)

[Conventions et règles de nommage des variables 14](#_Toc111987028)

[Modifier le contenu d'une variable ou « Affectation » 15](#_Toc111987029)

[Types primitifs de variables 15](#_Toc111987030)

[Opérateurs 16](#_Toc111987031)

[Opérateurs arithmétiques 16](#_Toc111987032)

[Opérateur modulo **%** 16](#_Toc111987033)

[Opérateurs d’assignation 17](#_Toc111987034)

[Opérateurs d’incrémentation et de décrémentation 18](#_Toc111987035)

[Opérateurs relationnels 19](#_Toc111987036)

[Priorité des opérateurs 19](#_Toc111987037)

[Introduction au type de variable **String** 20](#_Toc111987038)

[Création d’une variable **String** 20](#_Toc111987039)

[Concaténation d’une chaine de caractères 20](#_Toc111987040)

[Afficher le contenu d'une variable 21](#_Toc111987041)

[Lire une donnée au clavier fournie par l’usager (**Scanner**) 22](#_Toc111987042)

[Lire une donnée au clavier pour une variable de type **String** 22](#_Toc111987043)

[Lire une donnée au clavier pour une variable de type Entier (**int**) 23](#_Toc111987044)

[Structures conditionnelles 24](#_Toc111987045)

[Instruction **if** 24](#_Toc111987046)

[Instruction **if-else if** 26](#_Toc111987047)

[Condition et opérateurs logiques 27](#_Toc111987048)

[Instruction **switch** 29](#_Toc111987049)

[Introduction à la portée des variables 33](#_Toc111987050)

[Trace d’exécution 34](#_Toc111987051)

[Exercices – Traces 34](#_Toc111987052)

[Classe **String** 36](#_Toc111987053)

[Caractères spéciaux (**\n**, **\t**) 36](#_Toc111987054)

[Obtenir la longueur d’une chaîne (**length**) 36](#_Toc111987055)

[Concaténer deux chaînes (**concat**) 37](#_Toc111987056)

[Obtenir un caractère dans une chaîne (**charAt**) 37](#_Toc111987057)

[Obtenir la position d’un caractère dans une chaîne (**indexOf**) 37](#_Toc111987058)

[Remplacer un caractère dans une chaîne par un autre (**replace**) 38](#_Toc111987059)

[Obtenir une partie d’une chaîne (**substring**) 38](#_Toc111987060)

[Transformer une chaîne en minuscules (**toLowerCase**) 39](#_Toc111987061)

[Transformer une chaîne en majuscules (**toUpperCase**) 39](#_Toc111987062)

[Comparer deux chaînes (**compareTo**) 39](#_Toc111987063)

[Vérifier si deux chaînes sont égales (**equals**) 40](#_Toc111987064)

[Convertir un tableau de caractères en String (**valueOf**) 40](#_Toc111987065)

[Convertir une chaîne en vecteur de caractères (**toCharArray**) 40](#_Toc111987066)

[Classe **Math** 41](#_Toc111987067)

[Comment générer un nombre aléatoire (**Math.random)** : 41](#_Toc111987068)

[Classe **Integer** 42](#_Toc111987069)

[Comment convertir un **String** en **int** : 42](#_Toc111987070)

[Constantes 42](#_Toc111987071)

[Conversion de types 42](#_Toc111987072)

[Conversion sans perte d'information 42](#_Toc111987073)

[Promotion arithmétique 44](#_Toc111987074)

[Conversions explicites (ou Cast) 45](#_Toc111987075)

[Exercices 46](#_Toc111987076)

[Itérative – boucles 48](#_Toc111987077)

[**for** 48](#_Toc111987078)

[Instruction **while** 50](#_Toc111987079)

[Instruction **do – while** 52](#_Toc111987080)

[Boucles avancées 53](#_Toc111987081)

[Les boucles avec plusieurs conditions 53](#_Toc111987082)

[Les boucles imbriquées 53](#_Toc111987083)

[Introduction à la programmation orientée objet 54](#_Toc111987084)

[Lexique 54](#_Toc111987085)

[Classes et objets 55](#_Toc111987086)

[Définir les attributs (ou les propriétés) 55](#_Toc111987087)

[Classes et instances 56](#_Toc111987088)

[Les méthodes 57](#_Toc111987089)

[Appel de méthodes 58](#_Toc111987090)

[Renvoi d'une valeur par une méthode 58](#_Toc111987091)

[Le constructeur 59](#_Toc111987092)

[Définition d'une méthode 59](#_Toc111987093)

[Appel de méthodes 60](#_Toc111987094)

[Les arguments d'une méthode 60](#_Toc111987095)

[Renvoi d'une valeur par une méthode 60](#_Toc111987096)

[Comment instancier ou créer un objet? 61](#_Toc111987097)

[Accesseurs et mutateurs 61](#_Toc111987098)

[La méthode **toString()** 63](#_Toc111987099)

[Exemple complet 1 : Classes **Personne** et **PersonneApplication** 63](#_Toc111987100)

[**static** vs non **static** 67](#_Toc111987101)

[Exemple complet 2 : Classes **Heure** et **Intervalle** 67](#_Toc111987102)

[Tableaux statiques 74](#_Toc111987103)

[Déclaration du tableau 74](#_Toc111987104)

[Création du tableau 74](#_Toc111987105)

[Initialisation à la déclaration 75](#_Toc111987106)

[Accès à un tableau 76](#_Toc111987107)

[Taille d’un tableau (**length**) 76](#_Toc111987108)

[Tableau d’objets 77](#_Toc111987109)

[Parcourir un tableau à l’aide d’une boucle 77](#_Toc111987110)

[Passer un tableau en arguments d’une méthode 78](#_Toc111987111)

[Tableau comme valeur de retour d’une méthode 80](#_Toc111987112)

[Packages 81](#_Toc111987113)

[API Java 83](#_Toc111987114)

[Documentation 84](#_Toc111987115)

[Convention des noms dans le code Java 84](#_Toc111987116)

[Règles pour les méthodes 85](#_Toc111987117)

[L’espacement entre les opérateurs 85](#_Toc111987118)

[L’indentation ou formatage du code 86](#_Toc111987119)

[Les commentaires 87](#_Toc111987120)

[Commentaire sur une ligne 87](#_Toc111987121)

[Commentaire sur plusieurs lignes 87](#_Toc111987122)

[La Javadoc 88](#_Toc111987123)

[Commentaires Javadoc 89](#_Toc111987124)

[Balises Javadoc 90](#_Toc111987125)

[**Javadoc** d’une classe 90](#_Toc111987126)

[**Javadoc** d’une méthode 91](#_Toc111987127)

[**Javadoc** d’une constante et d’une variable d’instance (attribut) 91](#_Toc111987128)

[**Générer la JavaDoc** 92](#_Toc111987129)

[Générer la JavaDoc avec **Eclipse** 93](#_Toc111987130)

[Tableaux dynamiques 96](#_Toc111987131)

[Déclaration d’une **ArrayList** 96](#_Toc111987132)

[Création d’une **ArrayList** 96](#_Toc111987133)

[Ajouter des éléments ***arrayList*.add(…)** 97](#_Toc111987134)

[Obtenir un élément ***arrayList*.get** **(int index)** 98](#_Toc111987135)

[Modifier un élément ***arrayList*.set** **(int index, Object element)** 98](#_Toc111987136)

[Supprimer un élément ***arrayList*.remove(int index)** 98](#_Toc111987137)

[Obtenir le nombre d’élément dans une une **ArrayList** (**size()**) 99](#_Toc111987138)

[Parcourir les éléments d’une **ArrayList** 99](#_Toc111987139)

[Boucle classique 99](#_Toc111987140)

[Boucle **for-each** 99](#_Toc111987141)

[Transformer un tableau dynamique en tableau statique ***arrayList*.toArray()** 99](#_Toc111987142)

[Énumération (**enum**) 100](#_Toc111987143)

[**enum.values()** (notions avancées) 101](#_Toc111987144)

[Exemples de code Java 104](#_Toc111987145)

[JUnit - Validation et correction de programmes 105](#_Toc111987146)

[Tests unitaires 105](#_Toc111987147)

[**JUnit** 105](#_Toc111987148)

[Ajouter à un projet Eclipse la librairie JUnit 106](#_Toc111987149)

[Références 107](#_Toc111987150)

# Glossaire des termes utilisés dans le cours

|  |  |  |
| --- | --- | --- |
| Terme | | Définition |
| Accesseur | Méthode dont le rôle est de donner à l’appelant une valeur contenue dans les attributs de l’appelé | |
| Adresse mémoire | Nombre correspondant à l’emplacement de l’élément dans le tableau de la mémoire de l’application. | |
| Affectation | Instruction qui donne une nouvelle valeur à une variable. L’affectation écrase la valeur déjà présente. | |
| Appel de méthode | Instruction qui lance l'exécution d'une autre méthode (et possiblement à un autre objet qu’on nomme alors l’appelant). Un appel peut transmettre des valeurs (sous forme de paramètres) à la méthode appelée ou non. La méthode appelée peut ou non retourner une valeur à son appelant. | |
| Attribut | Variable appartenant à un objet. L’attribut existe en mémoire aussi longtemps que l’objet qui le possède. (aussi appelé cariable d’instance) | |
| Classe | Une classe est une définition d’un type pour des objets de ce type. Une classe comprend des variables et des méthodes. Chaque objet de ce type possède toutes les variables d'instance et peut utiliser toutes les méthodes déclarées dans la classe. | |
| Code binaire | Code composé de 0 et 1 prêt à être exécuté par un processeur donné. | |
| Code source | Fichier texte qui contient des lignes de code compréhensible par le programmeur respectant les standards d’un langage donné. | |
| Compilateur | Outil servant à transformer le code source en code binaire. Le compilateur transforme tout le code source en un programme exécutable d’un seul coup. | |
| Constructeur | Méthode servant à donner une valeur initiale correcte à l’ensemble des attributs d’un objet en création. Elle a exactement le même nom que la classe. | |
| Débogueur | Outil servant à exécuter le code ligne par ligne et à afficher le contenu des variables au fur et à mesure. Le débogueur est une sorte d’interpréteur. | |
| Déclaration | Instruction qui informe le compilateur de l’existence d’une nouvelle variable. La déclaration comprend obligatoirement un nom et un type. | |
| Erreur d’exécution | Le programme compile et le compilateur ne perçoit aucune erreur. Cependant lors de l’exécution, le programme se termine de manière anormale et brutale. Souvent causé par une opération illégale (ex. Division par zéro). | |
| Erreur de compilation | Erreur dans la syntaxe du code source qui empêche le compilateur de produire le code binaire. | |
| Erreur de logique | Ce sont les erreurs les plus difficiles à détecter et à corriger. Le programme compile et s’exécute sans problème, mais le résultat obtenu n’est pas celui escompté! (Exemple : un calcul qui ne donnerait pas le bon résultat). Pour ce type d’erreur, le débogueur est l’outil le plus intéressant. | |
| Initialisation | Première instruction qui affecte une valeur à une variable. | |
| Instance | Une instance est une réalisation en mémoire d’une classe. Toutes les instances d’une même classe partagent une définition commune, mais chacune possède ses propres valeurs. | |
| Instruction | Élément minimal du code source. En java les instructions sont séparées par des « ; ». | |
| Interpréteur | Outil servant à exécuter du code source. L’interpréteur traduit le code source ligne par ligne. Le code est exécuté au fur et à mesure que l'interpréteur traduit le code source. Le débogueur Java est une sorte d’interpréteur. | |
| Machine virtuelle | Programme qui sert à exécuter d’autres programmes. | |
| Méthode | Procédure associée à un objet | |
| Mutateur | Méthode dont le rôle est de modifier une valeur de l’appelé | |
| Objet | Un objet est une réalisation en mémoire d’une classe. (Synonyme d’instance) | |
| Opérateur | Instruction de base qui sert à traiter l’information en mémoire :   * modifier * tester * recommencer | |
| Paramètre | Variable reçu par une méthode et déclaré entre parenthèse à côté du nom de la méthode. Argument est un synonyme de paramètre. | |
| Plateforme | Combinaison d’éléments (système d’exploitation, processeur, mémoire, machine virtuelle…) qui détermine le cadre d’exécution d’un programme. | |
| Procédure | Groupes d’instructions qui réalisent un objectif commun. La procédure débute lorsqu'elle est appelée et se termine en retournant à son point d’appel. | |
| Référence | Adresse mémoire emmagasinée dans une variable servant à accéder à un objet. La référence n’est pas l’objet, elle permet simplement d’accéder à l’objet. Une même référence peut changer d’objet en cours d’exécution. | |
| Retour | Valeur retournée par une méthode. L’instruction return en java généralement placé à la fin d’une méthode indique la valeur retournée. | |
| Signature | Nom d’une méthode. | |
| Type | Élément du langage qui définit les propriétés principales d’une variable :   * le nombre d’octets attribué pour stocker l’information * les valeurs que peuvent prendre la variable * les méthodes/opérateurs qui peuvent y avoir recours. | |
| Variable | Espace mémoire réservé (lors d’une déclaration de variable) pour contenir une information qui peut être changée lors de l’exécution du programme. Une variable a toujours un nom, un type (qui définit ce qu’elle peut contenir) et une valeur. | |
| Variable d’instance | Variable appartenant à un objet. La variable d’instance existe en mémoire aussi longtemps que l’objet qui le possède. (aussi appelé attribut) | |

# Introduction

## Qu’est-ce qu’un programme informatique?

* C’est un ensemble d'instructions destinées à être exécutées par un ordinateur. Ces instructions ont une syntaxe assez stricte pour que l’ordinateur soit capable de les comprendre.
* On distingue 2 sortes de programme :
  + Un **programme source** est un code écrit par **un informaticien** dans un langage de programmation. Il peut être compilé vers une forme binaire. Exemple d’un **programme source Java** :

**public** **class** Exemple1 {

**public** **static** **void** main (String[] args)

{

System.***out***.print ("Ce cours se donne au cegep Limoilou");

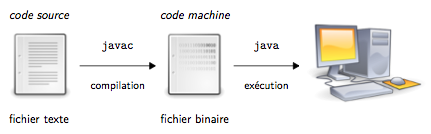
}

}

* + Un **programme binaire** décrit les instructions à exécuter par **un ordinateur** sous forme numérique. Ces instructions définissent un langage machine.



## Logo.Comment passer d’un programme source à un programme exécutable par la machine?

* Un **compilateur** permet de passer d’un **programme source** à un **code machine** exécutable par la machine.
* En java, le compilateur s’appelle **javac.**
* Opérations pour exécuter un programme :
  1. Le programmeur écrit un fichier texte qui contient un programme dans un certain langage (par exemple Java)
  2. Il le compile avec **javac.exe**. Ce qui produit un code machine dans un **fichier binaire**.
  3. Ce **code machine** est interprété et exécuter par le programme **java.exe** que l’on appelle la **machine virtuelle**. L’exécution, si le code source n’a pas d’erreur, devrait correspondre au résultat attendu par le programmeur.
* Les **environnements de développement** offrent au programmeur un outil puissant pour le supporter dans l’écriture des programmes informatiques.
* **IDE** = **Integrated Development Environment** (Environnement de développement intégré)

## Environnements de développement

* Un **environnement de développement** est un programme qui regroupe un ensemble d'outils pour augmenter la productivité des programmeurs qui développent des logiciels.
* Minimalement, il comporte :
  + un **éditeur de texte** destiné à la programmation
  + un **bouton** qui de démarre le compilateur puis qui exécute le programme
  + un **débogueur** en ligne, qui permet d'exécuter ligne par ligne le programme en cours d’exécution
* Il y a plusieurs environnements de programmation pour développer en java :
  + **IntelliJ IDEA** (<https://www.jetbrains.com/idea/download/#section=windows>)
  + **Eclipse** (<https://eclipse.org/downloads/> )
  + **NetBeans** (<https://netbeans.org/downloads/8.0.2/>)
* Dans le cadre de ce cours, nous utiliserons **IntelliJ IDEA**.

# Bases de programmation en Java

## Une instruction Java

* Une instruction est une opération simple que doit exécuter le programme.
* En java les instructions sont séparées par des « ; ».
* Les blocs de code en Java sont encadrés par des accolades {}.
* En général on ne met qu’une instruction par ligne de code, mais le langage permet d’en mettre autant qu’on le souhaite (c’est juste une mauvaise pratique).
* Certaines lignes du code source ne sont pas des instructions : les **lignes blanches** et les **commentaires**.

## Les lignes blanches

* Une ligne blanche n’est pas une instruction.
* Ces lignes sont ignorées lors de la compilation.
* Le Java permet de mettre autant de ligne vide que l’on veut.
* Elles permettent de séparer des blocs de code.

## Les commentaires

* Un commentaire n’est pas une instruction.
* Un commentaire permet d'insérer du texte qui ne sera pas compilé.
* Il donne la possibilité au programmeur d’ajouter du texte au code source.
* Nous verrons plus loin que les commentaires permettront au programmeur de documenter son code.
* Il donne la possibilité au programmeur de **neutraliser** une ligne de code
* En java, il y a 2 types de commentaire :
  + Les **commentaires de fin** de ligne (le //)
  + Les **commentaires sur plusieurs** lignes (les /\*et \*/)

### Commentaires de fin de ligne

* Un **commentaire de fin de ligne** commence par un double slash (//) et se termine au retour à la ligne.
* **Exemple**

*// Un commentaire pour donner l'exemple***int** nbArticle = 10; *// 10 articles  
  
// Ici, on veut simplement ignorer temporairement une instruction.  
// Celle-ci ne sera pas exécutée.  
  
// System.out.print ("Au revoir!");*

### Commentaires sur plusieurs lignes

* Un **commentaire sur plusieurs lignes** est encadré par slash + étoile, et étoile + slash..
* Exemple 2 (suite) :

*/\**

*Ceci est un commentaire*

*sur plusieurs lignes.*

*\*/*

*/\**

*Code de débogage désactivé :*

*int a = 10;*

*System.out.println("Valeur de a" + a);*

*\*/*

## Instruction d’affichage à la console

* On peut utiliser les méthodes print et println de l'objet System.out pour afficher le contenu une chaîne et le résultat d’un calcul mathématique.

La différence entre le System.***out***.println et le System.***out***.print ç’est que le premier sort un saut de ligne après l’affichage de la ligne.

**Exemple**

System.***out***.println (**"Voici un programme Java écrit en 2020."**);  
System.***out***.println (**"Au mois de septembre."**);  
System.***out***.println (**"Le Titanic a coulé en 2012."**);  
System.***out***.print (**"Ça fait "**);  
System.***out***.print (2020 - 1912);   
System.***out***.println (**" ans."**);

À l’exécution ce code affichera :

Voici un programme Java écrit en 2022.

Au mois de septembre

Le Titanic a coulé en 2012.

Ça fait 108 ans.

## Opérations simples sur des variables numériques (**+**, **-**, **\***, **/**)

* Soit l’expression qui représente l’addition de 2 nombres entiers :

1 + 1

* + Cette opération est une addition : « **la valeur de l’expression est** » le résultat de l’addition des **2 valeurs 1** et l’expression vaut **2**.
  + De la même manière, on peut utiliser toutes les opérations arithmétiques simples : **+ - \* /**.

**Exemple**

4 + 1 *//L'expression vaut 5*6 \* 3 *//L'expression vaut 18*9 / 3 *//L'expression vaut 3*

* + Vous remarquerez que les instructions précédentes sont inutiles parce qu’elle n’utilise pas le résultat. **Une** façon d’utiliser serait d’en faire l’affichage.

**Exemple**

System.***out***.println (1 + 1); *//Affiche 2*System.***out***.println (4 + 1); *//Affiche 5*System.***out***.println (6 \* 3); *//Affiche 18*System.***out***.println (9 / 3); *//Affiche 3*

* + On peut également utiliser les parenthèses pour forcer l’ordre des opérations.

System.***out***.println (9 / 3 - 2); *//Affiche 1*System.***out***.println (9 / (3 - 2)); *//Affiche 9*

# Patron de programme pour les premières semaines

On placera dans le **src** les classes Java :



|  |
| --- |
| **Semaine01.java** |
| **public class** Semaine01 {  *// méthode Constructeur* **public** Semaine01() {  System.out.println(**"Constructeur de la classe Semaine01"**);  *// afficherBonjourLeMonde();  // afficherBonjourLeMondeV2();  //afficher1Plus1();  //afficherAgePereALucV1();* afficherAjouter1AuNombreRecuEnArgument(771);  }   *//méthode* **public void** afficherBonjourLeMonde() {  System.out.println(**"Bonjour le monde!"**);  }   **public void** afficherBonjourLeMondeV2() {  System.out.print(**"Bonjour"**);  System.out.print(**" le"**);  System.out.print(**" monde!!!!"**);  }   **public void** afficher1Plus1() {  System.out.println(**""**);  System.out.print(**"1 + 1 = "**);  System.out.println(1 + 1);  }   **public void** afficherAgePereALucV1() {  System.out.print(**"Le père à Luc a "**);  System.out.print(15 \* 3);  System.out.println(**" ans."**);  }   **public void** afficherPereALuc() {  System.out.print(**"Le père à Luc a "**);  }   **public void** afficherAgePereALucV2() {  afficherPereALuc();  System.out.print(15 \* 3);  System.out.println(**" ans."**);  }   **public void** afficherAjouter1AuNombreRecuEnArgument(**int** nb) {  System.out.print(nb);  System.out.print(**" + 1 = "**);  System.out.println(nb + 1);  }  **public static void** main(String[] args) {  System.out.println(**"demo126G1"**);  *//instanciation de la classe Semaine01* **new** Semaine01();  } } |

# Les variables

* Les variables sont des symboles qui associent un nom (un identifiant) à une valeur. La valeur peut être d'un type de donnée ou l'autre.
* On peut voir une variable comme **une boîte** dans laquelle on peut placer **une donnée**.

**Exemple**

int variable;

variable = 17;

Variable

* Cette boîte, on va lui donner **un nom** afin de l'identifier pour **accéder** et **modifier** son contenu.
* Dans l’exemple, on place **la valeur** 17 dans cette boîte. Comme cette valeur est **un entier**, on indique que le format de la boîte est fait pour contenir un entier. Ainsi, on **déclare** que cette variable est un **int**.
* Un nom de variable ne peut comporter que des lettres, des chiffres (les caractères \_ et $ peuvent être utilisés mais ne devrait pas l'être pour des variables)
* Un nom de variable ne peut pas commencer par un chiffre et ne doit pas comporter d'espace
* Les noms de variables ne peuvent pas être les noms réservés du langage (abstract, assert, boolean, break, byte, case, catch, char, class, const, continue, default, do, double, else, enum, extends, false, final, finally, float, for, goto, if, implements, import, instanceof, int, interface, long, native, new, null, package, private, protected, public, return, short, static, sctrictfp, super, switch, synchronized, this, throw, throws, transient, true, try, void, volatile, while)
* En Java, toutes les variables utilisées dans un programme doivent être déclarées.
* Dans tous les langages, la bonne pratique est de déclarer les variables dans le haut des méthodes.
* Le langage Java est sensible à la casse, ce qui signifie que des lettres majuscules et minuscules ne constituent pas la même variable. Exemple :

La variable I, est différente de i.

La variable Nombre est différente de nombre.

* Leur contenu est modifié avec le signe **=**. Exemple :

a = 2

* Les variables doivent commencer par une **lettre** (ou un "**\_**") et peuvent contenir **lettre**, **chiffres** et le caractère "**\_**" (barre de soulignement). L’espace n’est pas autorisé.

**Exemples**

|  |  |
| --- | --- |
| Nom de variable | Est valide ou est refusé? |
| \_ | Est valide  ? |
| \_a | Est valide |
| nombreDeChat | Est valide |
| Nombre de chat | Refusé parce qu’il y a des **espaces** |
| Jean@mail-city.com | Refusé parce qu’il y a des caractères spéciaux (**@**, **-**, **.**) |
| 11a | Refusé parce qu’elle **commence par un nombre** |
| true | Refusé parce que c’est un **mot réservé** |

**Exemple**

Voici un exemple simple de code source Java ou chaque ligne est une instruction:

// Déclaration des variables

**int** premierNombre;

**int** secondNombre;

**int** resultat;

// Affectation des variables

premierNombre = 5;

secondNombre = 8;

// Opération sur des variables

resultat = premierNombre + secondNombre;

## Conventions et règles de nommage des variables

* Le **nom d'une variable** est un **identificateur unique** pour une portion de code. Il faudra choisir un nom qui décrit bien son contenu. **Exemple** d’une portion de code :

|  |  |
| --- | --- |
| **int** a = 100;  **int** b = 50;  **int** c;  c = a / b; | int distance = 100;  Que fait ce code?  int vitesse = 50;  int tempsDeplacement;  tempsDeplacement = distance / vitesse |

* Par convention :
  + les noms de variables commencent par une minuscule
  + contiennent des majuscules uniquement à chaque changement de mot
  + on n’utilise pas d’accent. On préfère strEleve à strÉlève
* Il convient de choisir des noms de variable qui soient le plus concis et explicites possibles. Un nom de variable comme maSuperVariableQuiContientMonAge est tout à fait à éviter et on préférera monAge ou même simplement age. La longueur maximale suggérée est de 32 caractères
* Exemples de bon de nom de variable :

|  |  |
| --- | --- |
| **Explication du contenu de la variable** | **Bon nom de variable** |
| Vitesse d’une voiture | vitesseVoiture  ? |
| Nombre de réussites | nbReussite |
| Somme des factures | sommeFacture |
| Nom de l’employé | nomEmploye |
| Vrai si c’est un succès | estUnSucces |
| Vrai si c’est réussi | estReussi |

## Modifier le contenu d'une variable ou « Affectation »

* On utilise l’opérateur « = » pour modifier le contenu d’une variable.

premierNombre = 5;

* Avec cette instruction, on demande de placer la valeur 5 dans la variable **premierNombre**.
* À gauche il ne peut y avoir qu’une variable et à droite il peut y avoir toute forme d’expression qui fournit un entier.

# Types primitifs de variables

Les types primitifs de variables sont utilisés pour placer en mémoire les données de base les plus courantes. Ces types spécifient quelle donnée pourra être conservée dans l’espace réservé. Chacun de ses types est caractérisé par un nombre d'octets prédéfinis. Les types élémentaires commencent tous par une minuscule.

Voici un tableau répertoriant les huit types primitifs du langage Java :

|  |  |  |
| --- | --- | --- |
| **Type de donnée** | **Type primitif** | **Description** |
| **Entiers** | byte | 1 octet -128 à +127 |
| short | 2 octets -32768 à +32767 |
| int | 4 octets -2147483648 à +2147483647 |
| long | 8 octets -9233372036854775808 à 9233372036854775807 |
| **Flottants** | float | 4 octets -1.4 x 10-45 à +3.4 x 1038  ou 2-1074 à 21024  Par défaut, les variables sont créées en double. Pour forcer **float**, il faut faire suivre la constante de la lettre **f** ou **F**.  **Exemple**  float x;  x = 12.5; //erreur ici 12.5 est de type double  x = 12.5f; //ok |
| double | 8 octets 4.9 x 10-324 à +1.7 x 10308  ou 2-1074 à 21024 |
| **Caractère** | char | 2 octets 'c' (unicode)  **Exemple**  char c = 'e'; |
| **Booléen** | boolean | 1 octet **true** et **false**  **Exemple**  boolean estValide = true; |

Notez que le type **String** n'est pas un type de base, mais plutôt une classe qui contient des **char**. Comme toute classe, le type **String** s'écrit avec une lettre majuscule contrairement aux types de base. Les classes et les **String** seront vus plus loin dans le cours…

# Opérateurs

* Les opérateurs sont des symboles qui effectuent une action spécifique sur une ou deux opérandes et qui retournent un résultat. On dira que le résultat « **est la valeur de l’expression** ».

**Exemple 1**

Dans :

**1 + 6**

«**1**», «**6**» sont des **opérandes**

«**+**» est l’opérateur

**7 « est la valeur de cette expression ».**

**Exemple 2**

Dans :

**++a**

«**a**» est l’**opérande**

«**++**» est l’opérateur

**« La valeur de cette expression »** est (« la valeur de a » + 1).

* En Java, il existe plusieurs types d’opérateurs, pour le moment, nous allons nous intéresser aux catégories suivantes :
  + arithmétiques
  + d’assignation
  + d’incrémentation et de décrémentation
  + relationnels

## Opérateurs arithmétiques

|  |  |  |  |
| --- | --- | --- | --- |
| **opérateur** | **signification** | **expression** | **valeur** |
| **+** | addition | 2 + 2 | 4 |
| **-** | soustraction | 3 – 2 | 1 |
| **\*** | multiplication | 3 \* 2 | 6 |
| **/** | division | 4 / 2 | 2 |
| **%** | Modulo (reste de la division) | 12 % 5 | 2 |

### Opérateur modulo **%**

**Exemples**

|  |  |
| --- | --- |
| **expression** | **valeur** |
| **6 % 5** | 1 |
| **6 % 0** | Erreur division par 0 |
| **6 % 4** | 2 |
| **1 % 2** | 1 |
| **2 % 3** | 2 |
| **125 % 128** | 125 |
| **5.5 % 2** | 1.5 |

## Opérateurs d’assignation

* Ces opérateurs permettent d’affecter une valeur à une variable.
* Nous avons vu l’affectation simple avec le symbole **=**, il en existe d’autres qui permettent de simplifier des opérations arithmétiques et d’affectation.

|  |  |  |  |
| --- | --- | --- | --- |
| **Opérateurs d’assignation** | | **Exemple** | **Valeur de a** |
| **=** | affectation usuelle | a = 2 + 2 ; | 4 |
| **+=** | additionne deux valeurs et stocke le résultat dans la variable (à gauche) | a = 4 ; a += 3 ;  identique à  a = a + 3 ; | 7 |
| **-=** | soustrait deux valeurs et stocke le résultat dans la variable | a = 4 ; a -= 3 ;  identique à a = a – 3 ; | 1 |
| **\*=** | multiplie deux valeurs et stocke le résultat dans la variable | a = 3; a \*= 2;  identique à  a = a \* 2; | 6 |
| **/=** | divise deux valeurs et stocke le quotient dans la variable | a = 4 a /= 2; identique à  a = a / 2; | 2 |
| **%=** | divise deux valeurs et stocke le reste de la division dans la variable | a = 4 a %= 3 identique à  a = a % 3; | 1 |

## Opérateurs d’incrémentation et de décrémentation

* L’incrémentation permet l’ajout d’une unité à une variable. Par exemple si une variable qui vaut 4 aura la valeur 5 après l’avoir incrémenté. À l’inverse, la décrémentation diminue une d’une unité une variable.
* En Java, nous disposons de deux opérateurs ++ et -- pour respectivement incrémenter ou décrémenter.
* i++ ou ++i pour l'incrémentation
* i-- ou --i pour la décrémentation
* Employé sans affectation, il n'y a pas de différence entre la notation :
* préfixe (++i, --i) et la notation postfixe (i++ i--).
* incrémenter/décrémenter une variable et en même temps affecter sa valeur à une autre variable. Dans ce cas, nous devons choisir entre la notation préfixe et postfixe:
* x = i++ passe d'abord la valeur de i à x et incrémente après
* x = i-- passe d'abord la valeur de i à x et décrémente après
* x = ++i incrémente d'abord et passe la valeur incrémentée à x
* x = --i décrémente d'abord et passe la valeur décrémentée à x
* Exemples :

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **opérateur** | **signification** | **exemple** | **résultat** | **Valeur de a** | **Valeur de b** |
| **++** | incrément | a = 2;  b = ++a; | Incrémente d'abord **a** et passe la valeur incrémentée à **b** | 3 | 3 |
| a = 2;  b = a++; | Passe d'abord la valeur de a à b et incrémente après | 3 | 2 |
| **--** | décrément | a = 2;  b = --a; | Décrémente d'abord a et passe la valeur décrémentée à b | 1 | 1 |
| a = 2;  b = a--; | Passe d'abord la valeur de a à b et décréement après | 1 | 2 |

## Opérateurs relationnels

* Java propose une série d'opérateurs d'égalité et de comparaison qui renvoient comme résultat un booléen (true ou false).
* On les utilise notamment pour prendre des décisions dans un programme comme on verra à la leçon suivante.

|  |  |  |  |
| --- | --- | --- | --- |
| **opérateur** | **signification** | **exemple** | **Valeur de l’expression** |
| == | égal | 4 == 4 | true |
| != | différent | 4 != 4 | false |
| > | plus grand | 4 > 4 | false |
| >= | plus grand ou égal | 4 >= 4 | true |
| < | plus petit | 4 < 4 | false |
| <= | plus petit ou égal | 4 <= 4 | true |

## Priorité des opérateurs

Dans tous les langages de programmation, il existe un ordre d’exécution des calculs spécifiques. Les calculs sont analysés et réalisés selon la **priorité des opérations** et ensuite ils sont évalués **de la gauche vers la droite**.

Le tableau présente sous une forme simplifiée l’ordre des priorités des opérations les plus courantes.

|  |  |
| --- | --- |
| **Priorité** | **Opérateur** |
| **1** | () [] |
| **2** | ++ -- ! |
| **2** | le – unaire (changement de signe exemple : –a) |
| **3** | \* / % |
| **4** | + - |
| 5 | < > <= >= |
| 6 | == != |
| 7 | && |
| 8 | || |
| **9** | = += -= \*= /= %= &= ^= |= <<= >>= |
| 10 | , |

**Exemple**

Pour vous aider à comprendre, observons les opérations suivantes :

**int** a, b, c, d;

a = 5;

b = 10;

c = 2;

d = a + b \* c;

// Le compilateur fera (b \* c) + a = 25

d = c \* b / 2;

// le compilateur fera c \* b / 2 = 10

d = (a + b) \* c;

// Le compilateur fera (a + b) \* c = 30

d = c \* (b / 2);

// Le compilateur fera (b / 3) \* c = 10

Pour la clarté et afin de lever les ambiguïtés, **les parenthèses demeurent le meilleur moyen de composer vos expressions mathématiques**.

# Introduction au type de variable String

* Nous avons vu que pour utiliser des chaînes de caractères en Java, il suffit simplement d'écrire la séquence de caractères entre guillemets doubles. Par exemple, pour afficher le texte Hello World ! à l'écran, on doit écrire l'instruction suivante :

System.***out***.println("Hello World !")

## Création d’une variable String

* Pour créer une nouvelle variable chaîne de caractères, on fera comme tous les types de base en Java :

String s = " Hello World !";

## Concaténation d’une chaine de caractères

La concaténation permet de placer plusieurs chaînes de caractères ensemble tout comme placer plusieurs mots ensemble permettent de former une phrase. Ainsi, concaténer deux chaînes de caractères A et B donnera une nouvelle chaîne qui contiendra les informations des chaînes A et B. Cette opération se fait en utilisant l'opérateur de concaténation +.

**Exemple**

String prenom = "Denis";

String nom = "Fortin";

String prenomEtNom = prenom + " " + nom;

System.***out***.println (prenomEtNom);

System.***out***.println ("J'ai " + 18 + " ans.");

* À l’exécution de cette portion de code, voici ce qui sera affiché :

Denis Fortin

J'ai 18 ans.

## Afficher le contenu d'une variable

**Exemple**

*/\*\*  
 \* Affiche le nombre d'année depuis que le Titanic a échoué en fonction   
 \* de l'année reçue en argument.  
 \** ***@param annee*** *valeur sur laquel le calcule du nombre d'année se est fait.  
 \*/*

**public void** afficherTitanic(**int** annee) {  
 **int** anneeTitanic = annee - 1912;  
 System.***out***.println (**"Le Titanic a coulé en 2012."**);  
 System.***out***.println (**"Si nous sommes en "** + annee + **"."**);  
 System.***out***.println (**"Ça fait "** + anneeTitanic + **" ans qu'il a coulé."**);  
}

|  |  |
| --- | --- |
| Si on appelle cette méthode avec… | Le résultat est… |
| afficherTitanic(2019); | Le Titanic a coulé en 1912.  Si nous sommes en 2019.  Ça fait 107 ans qu'il a coulé. |
| afficherTitanic(2012); | Le Titanic a coulé en 1912.  Si nous sommes en 2012.  Ça fait 100 ans qu'il a coulé. |

* Attention, si vous tentez d'afficher la valeur d'une variable non-initialisée, cela provoquera une erreur de compilation.

String annee;

System.***out***.print ("Voici un programme Java écrit en " + annee);

À l’exécution on aura l’erreur suivante qui sera affichée :

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

The local variable annee may not have been initialized

# Lire une donnée au clavier fournie par l’usager (Scanner)

* Pour lire une donnée fournie par l’usager, le code devra obligatoirement utiliser une **variable** pour être en mesure de l’utiliser sa valeur par la suite.

## Lire une donnée au clavier pour une variable de type String

**Exemple**

**import** java.util.Scanner; *//librairie nécessaire pour lire des caractères* *//...*

*..../\*\***....\* Lit au clavier un nom et un prénom et produit un affichage qui retourne   
....\* les valeurs saisies au clavier  
....\*/*

**public void** lireEtAfficherNomPrenom() {  
 Scanner sc; *//Instance de l'utilitaire de lecture* String nom;  
 String prenom;  
  
 sc = **new** Scanner(System.***in***); *//Instanciation de l'utilitaire de lecture* System.***out***.println(**"Entrez votre nom : "**);

*//Appel d’une méthode qui récupère une chaine de caractères saisis au clavier.* nom = sc.nextLine();

System.***out***.println(**"Entrez votre prénom : "**);  
 prenom = sc.nextLine(); *//lecture du prenom* System.***out***.println(**"Bonjour Monsieur ou Madame "** + nom);  
 System.***out***.print(**"Je sais que votre prénom est "** + prenom);  
  
}

À l’exécution de cette portion de code, voici ce qui sera affiché :

Entrez votre nom :

Ça dépend ce que l’usager entre au clavier

Fortin

Entrez votre prénom :

Denis

Ça dépend ce que l’usager entre au clavier

Bonjour Monsieur ou Madame Fortin

Je sais que votre prénom est Denis

## Lire une donnée au clavier pour une variable de type Entier (int)

**Exemple**

*/\*\*  
 \* Saisit le nombre de billes que possèdent Denis et Louise  
 \* et affiche le total de leurs billles.  
 \*/***public void** lireEtAfficherDesNombresDeBilles() {  
 Scanner sc; *//Instance de l'utilitaire de lecture* **int** nbBillesDenis;  
 **int** nbBillesLouise;  
 **int** totalBilles;  
  
 sc = **new** Scanner(System.***in***); *//Instanciation de l'utilitaire de lecture* System.***out***.println(**"Entrez le nombre de billes de Denis : "**);  
 nbBillesDenis = sc.nextInt();  
  
 System.***out***.println(**"Entrez le nombre de billes de Louise : "**);  
 nbBillesLouise = sc.nextInt();  
  
totalBilles = nbBillesDenis + nbBillesLouise;  
  
 System.***out***.println(**"Denis et Louise ont "** + totalBilles + **" billes ensembles."**);  
}

À l’exécution de cette portion de code, voici ce qui sera affiché :

Ça dépend de ce que l’usager entre au clavier

Entrez le nombre de billes de Denis :

10

Entrez le nombre de billes de Louise :

24

Ça dépend de ce que l’usager entre au clavier

Denis et Louise ont 34 billes ensembles.

# Structures conditionnelles

## Instruction if

* L'instruction if est une structure de base et on la retrouve dans tous les langages. Elle permet d'exécuter une série d'instructions lorsqu'une **condition** est réalisée. Une **condition** est une expression booléenne et elle est vérifiée si sa valeur est vraie (true).
* Voici la syntaxe de l’instruction if :

If-else instruction syntax

if (condition) {

//liste d'instructions si la condition est vraie

}

else {

//liste d'instructions si la condition est fausse

}

* Remarquez que la condition doit être entre parenthèses. Comme la condition doit retourner un résultat booléen, elle est composée avec des opérateurs relationnels (==, !=, >, >=, <, <=).

**Exemple**

Par exemple, voici une méthode qui indique si le nombre reçu en paramètre est négatif ou positif.

**public void** afficherSiNombreNegatifPositif(**int** nombre) {  
 String str;  
  
 str = **"Le nombre "** + nombre + **" est "**;  
 **if** (nombre < 0)  
 str += **"négatif."**;  
 **else** str += **"positif."**;  
 System.***out***.println(str);  
}

Avec les appels suivants :

afficherSiNombreNegatifPositif (-2);  
afficherSiNombreNegatifPositif (4);  
afficherSiNombreNegatifPositif (0);

Voici le résultat obtenu :

Le nombre -2 est négatif.

Le nombre 4 est positif.

Le nombre 0 est positif.

* Augmentons la complexité de notre instruction, nous voudrions vérifier si notre nombre est **négatif**, **positif** ou **nul** (valeur 0) et retourner une chaîne correspondante. Voici une solution possible :

**public void** afficherSiNombreNegatifPositifOuNulV1(**int** nombre) {  
 String str = nombre + **" est un nombre "**;  
  
 str = **"Le nombre "** + nombre + **" est "**;  
 **if** (nombre == 0)  
 str += **"nul."**;  
 **else** {  
 **if** (nombre < 0)  
 str += **"négatif."**;  
 **else** str += **"positif."**;  
 }  
 System.***out***.println(str);  
}

Avec les appels suivants :

afficherSiNombreNegatifPositifOuNulV1(-2);  
afficherSiNombreNegatifPositifOuNulV1(4);  
afficherSiNombreNegatifPositifOuNulV1(0);

Voici le résultat obtenu :

Le nombre -2 est négatif.

Le nombre 4 est positif.

Le nombre 0 est nul.

## Instruction if-else if

* Dans l’exemple précédent, on voit qu’il y a une instruction if imbriqué dans un else. Cette suite d’instructions est très courante et peut être contractée en une seule instruction qui elseif. Son utilisation est hautement recommandée puisqu’elle clarifie la lecture du programme en évitant de faire une indentation qui part vers la droite.
* Par exemple on voudrait élaborer une méthode qui convertit les un résultat scolaire en grade (lettre A, B, C, etc.) selon le barème suivant :

|  |  |
| --- | --- |
| 90 + | A |
| 80-89 | B |
| 70-79 | C |
| 60-69 | D |
| 50-59 | E |
| <50 | F |

* Voici 2 solutions :

|  |  |
| --- | --- |
| **Solution 1** | **Solution 2** |
| **public char** calculerNote1(**double** note) {   **char** grade;   **if** (note >= 90) {  grade = **'A'**;  } **else** {  **if** (note >= 80) {  grade = **'B'**;  } **else** {  **if** (note >= 70) {  grade = **'C'**;  } **else** {  **if** (note >= 60) {  grade = **'D'**;  } **else** {  **if** (note >= 50) {  grade = **'E'**;  } **else** {  grade = **'F'**;  }  }  }  }  }  **return** grade; } | **public char** calculerNote2(**double** note) {   **char** grade;   **if** (note >= 90) {  grade = **'A'**;  } **else if** (note >= 80) {  grade = **'B'**;  } **else if** (note >= 70) {  grade = **'C'**;  } **else if** (note >= 60) {  grade = **'D'**;  } **else if** (note >= 50) {  grade = **'E'**;  } **else** {  grade = **'F'**;  }  **return** grade; } |

Évidemment, nous utiliserons la **solution 2** qui est beaucoup plus simple à lire.

## Condition et opérateurs logiques

* Lorsque nous avons présenté les opérateurs, nous n’avons pas présenté les opérateurs logiques. Voici le moment de le faire.

|  |  |  |
| --- | --- | --- |
| **Opérateurs logiques** | | **Explication** |
| ! | négation | !(4 == 2 + 2) (false)  Retourne *true* si la variable vaut *false*, et *false* si elle vaut *true* |
| && | et | (4 == 2 + 2) && (3 == 2 + 2) (false)  Retourne *true* si les deux conditions valent *true*, sinon retourne *false* |
| || | ou inclusif | (4 == 2 + 2) || (3 == 2 + 2) (true)  Retourne true si au moins une des deux conditions vaut *true,* sinon retourne *false* |

* Avec les conditionnelles, il est fréquent d’avoir à composer des expressions complexes qui vérifient l’état ou la valeur de deux ou plusieurs variables.
* Voici la syntaxe de ces conditions :

if(!condition)…

if ((condition1)&&(condition2))…

if ((condition1)||(condition2))…

Voyons maintenant un exemple de chacun des cas en utilisant l’exemple calculerNotes écrit d’une façon différente.

**public char** calculerNote3(**double** note) {  
 **char** grade = **'G'**;  
  
 **if** (!(note > 0 && note < 100)) {  
 System.***out***.println(**"La note reçue n'est pas une valeur entre 0 et 100"**);  
 } **else if** (note >= 90) {  
 grade = **'A'**;  
 } **else if** (note >= 80) {  
 grade = **'B'**;  
 } **else if** (note >= 70) {  
 grade = **'C'**;  
 } **else if** (note >= 60) {  
 grade = **'D'**;  
 } **else if** (note >= 50) {  
 grade = **'E'**;  
 } **else** {  
 grade = **'F'**;  
 }  
 **if** (grade == **'A'** || grade == **'B'**) {  
 System.***out***.println(**"C'est une excellente note"**);  
 }  
 **return** grade;  
}

Voici le résultat avec différentes valeurs :

System.***out***.println(calculerNote3(90));

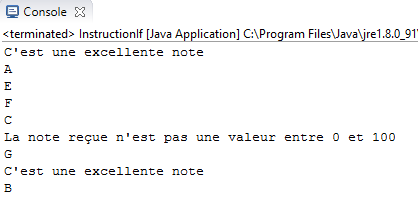
System.***out***.println(calculerNote3(59));

System.***out***.println(calculerNote3(35));

System.***out***.println(calculerNote3(71));

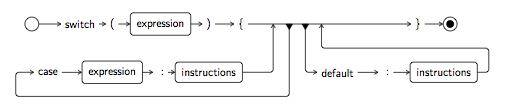
System.***out***.println(calculerNote3(-10));

System.***out***.println(calculerNote3(81));



## Instruction switch

* Cette instruction représente un aiguillage multiple : le choix est fait en fonction de la valeur d'une variable ; chaque chemin possible correspond à une valeur différente pour la variable. L’instruction switch ne fonctionne que sur les types de variables byte, short, char, int, enum et les sur les objets String (depuis le JDK7)
* Voici la syntaxe de l’instruction switch :



switch (variable) {

case valeur1 :

//Liste d'instructions

break**;**

case valeur2 :

//Liste d'instructions

break**;**

case valeurN:

//Liste d'instructions

break**;**

default**:**

//Liste d'instructions

}

Les parenthèses qui suivent le mot clé **switch** indiquent une expression dont la valeur est vérifiée successivement par chacune des clauses case. Lorsque l'expression est égale à une des valeurs suivant un case, la liste d'instructions qui suit celui-ci est exécuté. Le mot clé break indique la sortie de la structure conditionnelle, s'il n'est pas mis alors toutes les instructions suivantes seront également exécutées. Le mot clé default est facultatif et on le met à la fin, pour le cas où l'expression n'est jamais égale aux valeurs précédentes (c’est le sinon du switch)

* Exemple : l'instruction switch suivante permet d'afficher une phrase à l'écran en fonction de la valeur de la variable sexe :

**public** **void** afficherSexe(**char** sexe) {

**switch** (sexe) {

**case** 'M':

System.***out***.println("Vous êtes un homme.");

**break**;

**case** 'F':

System.***out***.println("Vous êtes une femme.");

**break**;

**default**:

System.***out***.println("Vous êtes de sexe indéterminé.");

}

}

La valeur de l'expression du switch est tout d'abord évaluée. Dans notre exemple, si la valeur de sexe est 'F', le programme se rend directement au case correspondant à cette valeur. L'instruction System.out.println ("Vous êtes une femme."); est donc exécutée et puis c'est au tour de l'instruction break qui permet de directement quitter le switch, puis la méthode.

* L'instruction switch n'est ni plus ni moins qu'un raccourci d'écriture pour l'instruction if-else if. Ce genre de raccourci est appelé **sucre syntaxique** ; il s'agit de constructions du langage dont on pourrait se passer et qui n'apportent rien de plus si ce n'est des facilités au niveau de l'écriture du code et de sa lisibilité (il y a quand même une petite différence puisqu'avec switch, la valeur de l'expression est évaluée une seule fois, mais avec l'équivalent if-else if, elle l'est à chaque fois).

**public** **void** afficherSexe2(**char** sexe) {

**if** (sexe == 'M') {

System.***out***.println("Vous êtes un homme.");

} **else** **if** (sexe == 'F') {

System.***out***.println("Vous êtes une femme.");

} **else** {

System.***out***.println("Vous êtes de sexe indéterminé.");

}

}

* Voyons un autre exemple de l’instruction switch.

**public** String quelJour(**int** jour) {

String jourSemaine;

**switch** (jour) {

**case** 1:

jourSemaine = "Dimanche";

**break**;

**case** 2:

jourSemaine = "Lundi";

**break**;

**case** 3:

jourSemaine = "Mardi";

**break**;

**case** 4:

jourSemaine = "Mercredi";

**break**;

**case** 5:

jourSemaine = "Jeudi";

**break**;

**case** 6:

jourSemaine = "Vendredi";

**break**;

**case** 7:

jourSemaine = "Samedi";

**break**;

**default**:

jourSemaine = "Invalide";

}

**return** jourSemaine;

}

* Avec un type enum la switch prend la forme suivante :

**public** **enum** JourSemaine {***LUNDI***, ***MARDI***, ***MERCREDI***, ***JEUDI***, ***VENDREDI***, ***SAMEDI***, ***DIMANCHE***};

**public** String quelJour(JourSemaine jour) {

String typeTravail = "";

**switch** (jour) {

**case** ***LUNDI***:

**case** ***MARDI***:

**case** ***MERCREDI***:

**case** ***JEUDI***:

**case** ***VENDREDI***:

typeTravail = "jour de travail";

**break**;

**case** ***SAMEDI***:

**case** ***DIMANCHE***:

typeTravail = "jour de congé";

**break**;

}

**return** typeTravail;

}

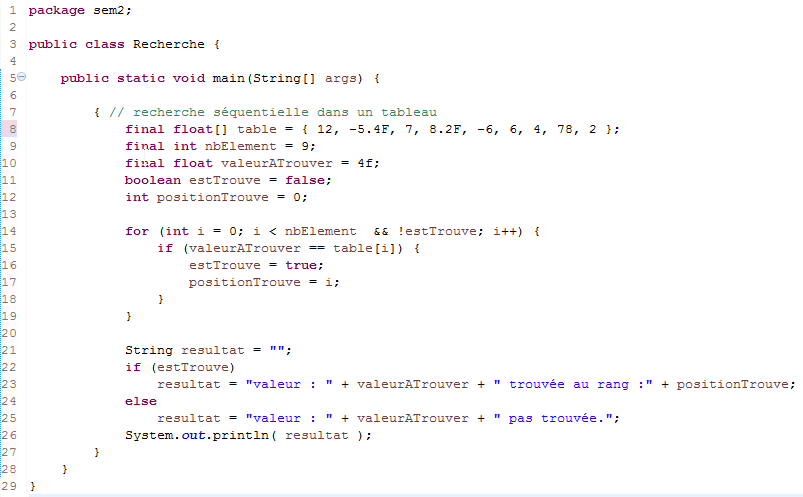
Voici le résultat avec différentes valeurs :

|  |  |
| --- | --- |
| **test** | **Résultat** |
| InstructionIf iIf = **new** InstructionIf();  System.***out***.println(iIf.quelJour(1));  System.***out***.println(iIf.quelJour(3));  System.***out***.println(iIf.quelJour(7));  System.***out***.println(iIf.quelJour(8));  System.***out***.println(iIf.quelJour(-10)); |  |

# Introduction à la portée des variables

* Selon l'endroit où l’on déclare une variable, celle-ci pourra être accessible (visible) de partout dans le code ou bien uniquement dans une portion de celui-ci, on parle alors de portée d'une variable.
* En java, toute variable est déclarée à l'intérieur d'un groupe d'accolades; sa portée se restreint donc à l'intérieur de ce bloc. Le groupe d'accolades peut être une classe, une méthode ou d'autres structures que nous verrons bientôt.
* Aussi, une variable est disponible pour utilisation uniquement après qu’elle a été déclarée.
* Il est interdit d'avoir deux variables de même nom si elles ont une portée commune.

**Exemple**

****

|  |  |  |  |
| --- | --- | --- | --- |
| **Variables ou constantes** | **Type** | **Variables ou constantes** | **Porté** Exemple : 5 à 22 |
| table | tableau de float\* | constante | 8 à 27 |
| nbElement | int | constante | 9 à 27 |
| valeurATrouver | float | constante | 10 à 27 |
| estTrouve | boolean | variable | 11 à 27 |
| positionTrouve | int | variable | 12 à 27 |
| i | int | variable | 14 à 19 |
| resultat | string | variable | 21 à 27 |

\*Ce type sera vu plus loin

# Trace d’exécution

* On appelle une trace, un tableau qui donne l’historique des valeurs des différentes variables durant l’exécution d’un programme. Cet outil est très intéressant pour vérifier la validité d’un algorithme.
* **Algorithme** : suite finie et non ambiguë d’opérations ou d'instructions permettant de résoudre un problème.
* Soit le code suivant :

**int** var1;

**int** var2;

**int** var3;

var1 = 5; // ligne 1

var2 = var1 + 4; // ligne 2

var3 = var1 \* var2 - 12; // ligne 3

var1 = var1 + var1; // ligne 4

var2 = var3 - var1 \* 2; // ligne 5

* Faisons la trace des 5 lignes identifiées dans le code. Pour cela on doit d’abord faire un tableau qui aura pour colonnes chacune des variables impliquées dans la section de code et pour ranger les 5 instructions concernées:

|  |  |  |  |
| --- | --- | --- | --- |
| étape | var1 (int) | var2 (int) | var3 (int) |
| Ligne 1 | 5 | -  ? | - |
| Ligne 2 | 5 | 9 | - |
| Ligne 3 | 5 | 9 | 33 |
| Ligne 4 | 10 | 9 | 33 |
| Ligne 5 | 10 | 13 | 33 |

## Exercices – Traces

?

|  |  |  |
| --- | --- | --- |
| Nom->  Type-> | a  int | b  int |
| int a = -2; | -2 | - |
| int b; | -2 | - |
| b = 5; | -2 | 5 |
| a = -a; | 2 | 5 |
| b = a; | 2 | 2 |
| b = b + a; | 2 | 4 |
| a = -a; | -2 | 2 |
| b = a; | -2 | -2 |
| b = b + a; | -2 | -4 |

|  |  |  |  |
| --- | --- | --- | --- |
| Nom->  Type-> | x  int | y  int | z  int |
| int x; | - | - | - |
| int y; | - | - | - |
| int z; | - | - | - |
| x = 0; | 0 | 0 | 0 |
| y = 10; | 0 | 10 | 0 |
| z = y; | 0 | 10 | 10 |
| y = y + 1; | 0 | 11 | 10 |
| z = z \* y; | 0 | 11 | 110 |
| x = x – 1; | -1 | 11 | 110 |
| X = x \* -2 + 3; | 5 | 11 | 110 |
| X = 3 + x \* 2; | 13 | 11 | 110 |

?

# Classe String

* La classe String fournie par l’API de Java permet de créer et manipuler des chaînes de caractères. Une instance de la classe String peut s’exprimer de différentes façons :
* Pour construire une chaîne vide :

String chaine1 = "";

//ou

String chaine2 = **new** String();

* Pour construire une chaîne avec du texte :

String chaine = "Une chaîne de caractères quelconque !";

//ou

String chaine = **new** String("Une chaine de caractères quelconque !");

## Caractères spéciaux (\n, \t)

Dans une chaine de caractère, vous pouvez insérer des caractères spéciaux :

|  |  |
| --- | --- |
| Séquence | Caractère |
| \n | nouvelle ligne |
| \r | retour chariot |
| \t | Tabulation |
| \b | Backspace |
| \f | form feed |
| \XXX | le caractère dont la valeur en octal est XXX |
| \uXXXX | le caractère dont la valeur en hexadécimal est XXXX |

**Exemple :**

String s = "Hello\nWorld !";

System.***out***.println (s);

Résultat :

Hello

World !

## Obtenir la longueur d’une chaîne (length)

Cette méthode retourne la longueur de la chaîne de caractères de l’objet en cours.

public int [length](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html#length())();

**Exemple :**

String s = "Paul";

System.out.println(s.length()); //affiche 4

s = "";

System.out.println(s.length()); //affiche 0

## Concaténer deux chaînes (concat)

Concatène la chaîne spécifiée à la fin de la chaîne courante. L’opérateur « + » fait le même travail.

public [String](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html) [concat](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html#concat(java.lang.String))([String](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html) str);

**Exemple**

*//Première façon de faire une concaténation* String strBienvenue = **"Cher étudiant,\n"**;  
 strBienvenue = strBienvenue.concat(**"Bienvenue au "**);  
 strBienvenue = strBienvenue.concat(**"cégep Limoilou"**);  
 System.***out***.println(strBienvenue);

*//Deuxième façon de faire une concaténation* String strBienvenue = **"Cher étudiant,\n"**;  
 strBienvenue += **"Bienvenue au "**;  
 strBienvenue += **"cégep Limoilou"**;  
 System.***out***.println(strBienvenue);

Résultat :

Cher étudiant,

Bienvenue au cégep Limoilou

## Obtenir un caractère dans une chaîne (charAt)

Cette méthode retourne le caractère de la chaîne courante qui se trouve à la position donnée en paramètre. Le premier caractère d’une chaîne se trouve à la position 0 et le dernier à la position length() – 1.

public char charAt(int index);

**Exemple :**

s = "Paul";

System.*out*.println(s.charAt(0)); //affiche P

System.*out*.println(s.charAt(s.length()-1)); //affiche l

//La ligne suivante génère une exception StringIndexOutOfBounds

//car cette position n'existe pas dans la chaine

System.*out*.println(s.charAt(s.length()));

## Obtenir la position d’un caractère dans une chaîne (indexOf)

Il s’agit de l’inverse de la méthode précédente. La méthode retourne l'index de la première occurrence du caractère mentionné. Plusieurs surcharges de cette méthode sont offertes, pour rechercher une chaîne ou rechercher à partir d’une position en particulier. Consultez l’API de Java pour en savoir plus.

public int [indexOf](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html#indexOf(int))(int ch);

**Exemple :**

String chaine = "unechaineencore";

System.out.println(chaine.indexOf('e')); //affiche 2

## Remplacer un caractère dans une chaîne par un autre (replace)

Remplace toutes les occurrences du premier caractère reçu en paramètre par celui reçu comme deuxième paramètre et retourne la chaîne résultante. Plusieurs surcharges de cette méthode sont offertes, consultez l’API de Java pour en savoir plus.

public [String](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html) [replace](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html#replace(char, char))(char oldChar, char newChar);

**Exemple :**

String mot = "equilibre";

mot.replace('e', 'é');

//affiche equilibre car nous n'avons pas touché à la variable mot

System.out.println(mot);

mot = mot.replace('e','é');

//affiche équilibré car nous avons remplacé le contenu de la variable mot par le //résultat du remplacement

System.out.println(mot);

## Obtenir une partie d’une chaîne (substring)

Retourne une nouvelle chaîne qui est une sous-chaîne de l'originale à partir de la position mentionnée en paramètre jusqu’à la fin. Il existe aussi une version qui précise la position de début et celle de la fin (exclus) de la sous-chaîne à extraire.

public [String](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html) [substring](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html#substring(int))(int beginIndex);

public [String](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html) [substring](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html#substring(int))(int beginIndex, int endIndex);

**Exemple :**

String s;

s = "Bonne journée";

//affiche journée

System.out.println(s.substring(6));

//affiche jour

System.out.println(s.substring(6,10));

//la chaîne originale est intacte Bonne journée

System.out.println(s);

## Transformer une chaîne en minuscules (toLowerCase)

Cette méthode convertit la chaîne en cours en minuscules et retourne la chaîne résultante. L’original demeure intact.

public [String](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html) [toLowerCase](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html#toLowerCase())();

**Exemple :**

String s;

s = "BRavo";

//affiche bravo

System.out.println(s.toLowerCase());

//affiche BRavo, la chaine originale n'est pas affectée

System.out.println(s);

## Transformer une chaîne en majuscules (toUpperCase)

Cette méthode convertit la chaîne en cours en majuscules et retourne la chaîne résultante. L’original demeure intact.

public [String](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html) [toUpperCase](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html#toLowerCase())();

**Exemple :**

String s;

s = "BRavo";

//affiche BRAVO

System.out.println(s.toUpperCase());

//affiche BRavo, la chaine originale n'est pas affectée

System.out.println(s);

## Comparer deux chaînes (compareTo)

Sans doute, l’une des méthodes les plus utilisées de cette classe. Elle permet de comparer deux chaînes et de retourner si la première est égale, plus petite ou plus grande à la seconde chaîne. Elle existe en deux saveurs, la première vérifie les chaînes telles que transmises, la seconde ignore les majuscules et minuscules lors de la comparaison. Les valeurs retournées sont 0 si les chaînes sont égales, une valeur < 0 si la chaîne en cours est plus petite que celle donnée en paramètre et une valeur >0 si la chaîne en cours est plus grande que celle donnée en paramètre.

public int [compareTo](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html#compareTo(java.lang.String))([String](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html) chaine);

public int [compareTo](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html#compareTo(java.lang.String))IgnoreCase([String](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html) chaine);

**Exemple :**

String chaine1 = "Julie";

String chaine2 = "jolie";

String chaine3 = "Paul";

/\* retourne une valeur négative (-32) Julie est plus petit que jolie

le code ASCII de J est 74 et j est 106\*/

System.out.println(chaine1.compareTo(chaine2));

// retourne une valeur positive(6) Julie est plus grand que jolie

// les majuscules sont ignorées le u (117) est plus grand que le o (111)

System.out.println(chaine1.compareToIgnoreCase(chaine2));

// retourne une valeur négative (-6) Julie (j = 106) est plus petit que Paul (p= 112)

System.out.println(chaine1.compareToIgnoreCase(chaine3));

// retourne 0 les deux chaînes sont égales

System.out.println(chaine1.compareToIgnoreCase(chaine1));

## Vérifier si deux chaînes sont égales (equals)

Compare la chaîne spécifiée à la chaîne courante.

public [boolean](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html) [equals](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html#concat(java.lang.String))([Object](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html) unObjet);

public [boolean](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html) [equals](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html#concat(java.lang.String))IgnoreCase([String](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html) uneString);

**Exemple :**

String s1 = "bon";

String s2 = "Bon";

System.*out*.println(s1.equals(s2)); //affiche false

System.*out*.println(s1.equalsIgnoreCase(s2)); //affiche true

## Convertir un tableau de caractères en String (valueOf)

Retourne la String correspondant au vecteur de caractères spécifié.

public static [String](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html) [valueOf](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html#concat(java.lang.String))(char[] unTab);

**Exemple :**

char[] vec = {'s', 'a', 'l', 'u', 't'};

System.*out*.println(String.valueOf(vec); //affiche salut

## Convertir une chaîne en vecteur de caractères (toCharArray)

Convertit la chaîne courante en tableau de char.

public [char[]](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html) [toCharArray](file:///C:/Users/julie.freve/Desktop/Documents%20and%20Settings/HP_Administrator/My%20Documents/ProgrammesJavadocsapijavalangString.html#concat(java.lang.String))();

**Exemple :**

String s1 = "bonjour";

char[] tab = s1.toCharArray();

System.*out*.println(String.valueOf(tab)); // affiche bonjour

Voilà, nous avons fait le tour de quelques méthodes très intéressantes de la classe String. Pour connaître l’ensemble des méthodes offertes par la classe String, consultez **l'API de Java**.

# Classe Math

## Comment générer un nombre aléatoire (Math.random) :

La méthode random de la classe Math génère un nombre réel aléatoire entre 0 et 1 (Attention : 0 inclus, 1 exclus donc entre 0 et 0.99999999…) de type double. Voici la signature de cette méthode :

public static double random()

* Pour obtenir un nombre dans un intervalle différent, il faut faire des manipulations mathématiques sur ce nombre pour obtenir une valeur dans l'intervalle voulue.

**Exemple**

Pour obtenir un nombre aléatoire dans l’intervalle **[0, 10[**:

int nb = Math.random() \* 10;

Après compilation, vous obtiendrez un message d’erreur : Can't convert double to int, explicit cast needed.

Cela est dû au fait que Math.random renvoie un double et non un int, il faut donc modifier le type de données en faisant un "casting".

nb = **(int)** **(**Math.random() \* 10**)**;

Attention, les parenthèses sont importantes :

nb = **(int)** Math.random() \* 75; donnerait 0 parce qu’ici le cast s’applique au random et non au résultat de la multiplication.

point de départ

**Exemple**

Pour obtenir un nombre aléatoire dans l’intervalle **[50, 55[**:

**int nb = (int) (Math.random() \* 5) + 50;**

intervalle

# Classe Integer

## Comment convertir un **String** en **int** :

1. L’instruction **Integer.parseInt()** convertit l’élément **String** qui lui est passé en paramètre en un entier **int**.

**Exemple**

À l’exécution de ce code…

String strValeur = **"2019"**;  
**int** intValeur;  
System.***out***.println(**"Integer.parseInt est une méthode qui convertit une chaine en variable int:"**);  
intValeur = Integer.*parseInt*( strValeur );  
System.***out***.println(**"La chaine est "** + strValeur);  
System.***out***.println(**"L'entier est "** + intValeur );  
System.***out***.println(**"Si on ajoute un à l'entier : "** + (intValeur + 1));  
System.***out***.println(**"Si on ajoute un à la string : "** + (strValeur + 1));

Le résultat est le suivant :

Integer.parseInt est une méthode qui convertit une chaine en variable int:

La chaine est 2019

L'entier est 2019

Si on ajoute un à l'entier : 2020

Si on ajoute un à la string : 20191

# Constantes

* Une constante est une donnée dont la valeur est inchangeable lors de l'exécution d'un programme. Elle est initialisée lors de sa création et sa valeur ne changera jamais tout au long du programme.
* Les constantes sont soumises aux mêmes règles que les variables en ce qui concerne leur portée.
* Voici comment déclarer et initialiser une constante locale :

final double TAXE = 0.075;

* Par convention, et pour les distinguer des variables, les constantes sont écrites entièrement en majuscules.
* (À voir plus tard lorsque la théorie sur les classes sera assimilée) Si on définit une constante au niveau d’une classe, elle sera statique. Comme nous l’avons mentionné plus tôt, une variable statique n’existe qu’en un seul exemplaire pour l’ensemble des objets créés par la classe. Il est inutile que chaque objet produit, par exemple, ait un exemplaire de la variable TAXE. Cette variable constante peut être partagée par l’ensemble des produits.

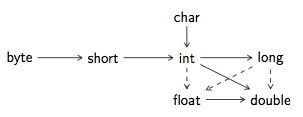
public static final double TAUX\_TAXE = 0.075;

# Conversion de types

* On ne peut faire du traitement arithmétique que sur des variables de mêmes types sous peine de perdre de la précision lors du calcul.
* On doit éviter à diviser un **int** par un **float**. Ceci est valable pour tous les opérateurs arithmétiques et pour tous les types de variables numériques.
* En Java, il existe des **conversions implicites**, celles effectuées par le compilateur lors d’une opération et les **conversions explicites** demandées par le programmeur au moment d’une opération.

## Conversion sans perte d'information

* La figure suivante résume les conversions sans perte d'information :



* Pour qu'une conversion sans perte d'information soit possible entre deux types, il faut juste que vous puissiez les rejoindre en suivant les flèches. Les flèches pleines représentent les conversions **sans perte de précision** et les flèches pointillées celles où une **perte de précision** peut avoir lieu.
* Si une perte de précision se produit, le programme continue à s'exécuter sans problème, aucune erreur d'exécution ne se produit; il faut donc être bien conscient que cela peut se produire.
* Il n'y a pas de perte d'information lorsque la grandeur du nombre converti est conservée. Voici un tableau représentant les conversions sans perte de données. Mais comme énoncé plus haut, les conversions vers un type primitif décimal peuvent mener à une **perte de précision.** Affiché en *italique* dans ce tableau.

|  |  |  |
| --- | --- | --- |
| **De** | **Taille en octets** | **Vers** |
| byte | 1 | short, int, long, float, double |
| short | 2 | int, long, float, double |
| char | 2 | int, long, float, double |
| int | 4 | long, *float*, double |
| long | 8 | *float*, *double* |
| float | 4 | double |

## Promotion arithmétique

* La **promotion arithmétique** intervient **automatiquement** lorsque certains opérateurs mathématiques doivent modifier leurs opérandes pour pouvoir exécuter l'opération.
* Par exemple, lorsque vous tentez de diviser un **float** par un **int**, la valeur du dénominateur sera promue en un type **float** avant la division, afin que le résultat de la division soit un **float**.
* Soient les instructions suivantes :

float num = 10.0F;

int denom = 2;

System.out.println (num / denom);

L'exécution du programme affiche 5.0 à l'écran puisqu'un des deux opérandes de l'opérateur **/** est de type **float**, ce qui implique que le résultat sera également un **float**. La valeur de la variable **denom** est convertie en un **float** avant la division.

* Soient les instructions suivantes :

float f = 25.5;

int i = 500;

double sum = f + i;

Dans cet exemple, on utilise l'opérateur **+** sur des opérandes de type **float** et **int**. L'opérande de droite est donc converti depuis le type **int** vers le type **float**. L'addition est ensuite effectuée en **float** et son résultat, qui est un **float**, est affecté à une variable de type **double**. Il y a donc deux conversions implicites lors de l'exécution de la troisième instruction : une promotion arithmétique suivie d'une conversion par affectation.

## Conversions explicites (ou Cast)

* Les conversions qu'on vient de voir se font de manière implicite, automatiquement. Mais on ne peut faire que des conversions sans perte d'information. Il est néanmoins parfois utile de réaliser des conversions avec perte d'information ou de convertir un **byte** en un **char**. Pour cela, on doit faire un cast, aussi appelé conversion explicite.
* On fait un cast grâce à l'opérateur de cast. Il s'agit d'un opérateur unaire. Grâce à cet opérateur, on va pouvoir convertir une donnée dans un type spécifié, que la conversion soit avec ou sans perte d'information.
* L'opérateur est constitué de deux parenthèses entre lesquelles se trouve un type de donnée, précisément celui vers lequel la conversion doit avoir lieu. Ainsi, pour convertir une donnée en **int**, la forme prise par l'opérateur est **(int).**
* L'exemple suivant montre plusieurs conversions utilisant l'opérateur de cast. Il y a une conversion de **int** vers **float** (sans perte d'information), de **int** vers **byte** (avec perte d'information) et enfin de **byte** vers **char**.

int x = 292;

float f = (float) x; // Conversion sans perte d'information

byte b = (byte) x; // Conversion avec perte d'information

char c = (char) b; // Conversion d'un byte vers un char

System.out.println("f = " + f);

System.out.println("b = " + b);

System.out.println("c = " + c);

System.out.println("c = " + (byte) c);

Voici ce qui est affiché à l'écran après exécution de ce programme :

f = 292.0

b = 36

c = $

c = 36

On voit effectivement que la première conversion n'a pas fait perdre d'information et l'entier 292 est devenu le flottant 292.0. L'opérateur de cast n'est pas nécessaire puisque cette conversion aurait pu avoir lieu de manière implicite (conversion par affectation). On peut néanmoins l'utiliser pour mettre en évidence le changement de type, le rendre explicite.

La deuxième conversion provoque une perte d'information. En effet, comme vous pouvez l'observer sur le résultat produit, la valeur n'est plus la même puisqu'on passe de 292 à 36. Pour comprendre pourquoi on obtient la valeur 36, il faut comprendre comment les entiers sont représentés dans l'ordinateur; Pour l’instant, tout ce que l’on a à comprendre c’est qu’il y a perte d’information et qu’une telle conversion n’est pas souhaitable.

Enfin la troisième conversion va du type **byte** vers le type **char**. Aucune information n'est perdue, mais cette conversion ne peut se faire que de manière explicite. En effet, pour rappel, le **byte** est d'abord converti en un **int** qui est ensuite converti en **char**. Cette dernière conversion étant classée parmi les conversions avec perte d'information, il faut utiliser l'opérateur de cast.

## Exercices

**public** **void** precision1() {

**int** nb; // entier

nb = 1 / 3;

System.***out***.println(" 1 / 3 : " + nb);

// resultat = 1 / 3 : 0

?

nb = 1 / 2;

System.***out***.println(" 1 / 2 : " + nb);

?

// resultat = 1 / 2 : 0

nb = 2 / 3;

System.***out***.println(" 2 / 3 : " + nb);

// resultat = 2 / 3 : 0

?

}

**public** **void** precision2() {

**float** nb; // réel

nb = 1 / 3;

System.***out***.println(" 1 / 3 : " + nb);

// resultat = 1 / 3 : 0.0

?

nb = 1 / 2;

System.***out***.println(" 1 / 2 : " + nb);

// resultat = 1 / 2 : 0.0

?

nb = 2 / 3;

System.***out***.println(" 2 / 3 : " + nb);

?

// resultat = 2 / 3 : 0.0

}

**public** **void** precision3() {

**float** nb; // réel

nb = (**float**) 1 / 3;

System.***out***.println(" 1 / 3 : " + nb);

// resultat = 1 / 3 : 0.33333334

?

nb = (**float**) 1 / 2;

System.***out***.println(" 1 / 2 : " + nb);

// resultat = 1 / 2 : 0.5

?

nb = (**float**) 2 / 3;

System.***out***.println(" 2 / 3 : " + nb);

?

// resultat = 2 / 3 : 0.6666667

}

**public** **class** Variable {

**public** **void** nomVariable () {

/\*

Exercice : Variables valides et respectant les normes.

Est-ce que les déclarations de variables suivantes sont

valides et selon les normes fixées ?

\*/

**int** 4chiens; //une variable ne peut pas commencer un chiffre

?

Short deuxChiens; //ok

bit nom3Ours; //bit type inexistant

**byte** couleur4x4; //ok

**byte** couleur4\*4; //opérateur \* interdit

String NomdePersonne; //ok

**int** besoin?; //? interdit

**int** a-b; //- interdit

**double** x; //ok

String nomdecommerce; //ok

String nobPerPremAcc; //ok

**boolean** **enum**; //mot réservé interdit

**boolean** \_possible; //ok

}

**public** **void** affectation () {

/\*

Exercice 1.3 Est-ce que les initialisations de variables suivantes

sont valides ? Si elles ne sont pas valides dites pourquoi ?

\*/

**int** nombre10 = 34.5; //on ne peut pas placer un réel (double) dans un  
 // entier

?

**int** nombre20 = (**int**)34.5; //ok conversion explicite

**int** nombre30 = 66; //ok

**float** nombre40 = 33.45 ; //on ne peut pas placer un double dans un float

**float** nombre45 = (**float**)33.45 ; //ok

**double** nombre50 = (**double**) 55,456665;//c'est le point en java qui sépare la partie  
 // décimal

**float** nombre60 = 45665.54f; //ok

**byte** nombre70 = 600; //dépassement de capacité

**short** nombre80 = 32767; //ok

**short** nombre85 = 32768; //dépassement de capacité

**char** char1 = 255; //ok

**char** char2 = "a"; //pas de chaîne dans un char

String string1 = "chaîne"; //ok

String chaine2 = 'asd'; //les apostrophes son réservé aux caractères

**boolean** estIci10 = 0; //ok le 0 représente le faux

**boolean** estIci20 = True; //ok

}

}

# Itérative – boucles

Les instructions répétitives sont des structures qui permettent d'exécuter plusieurs fois la même série d'instructions jusqu'à ce qu'une condition ne soit plus vraie.

Nous présentons trois structures répétitives équivalentes qu’on appelle aussi boucle :

* for
* while
* do – while

## for

L'instruction for permet de répéter la même série d’actions un nombre de fois défini. Pour se réaliser, l’instruction a besoin d’une variable qui compte le nombre de fois qu’une action est répétée, on appelle cette variable l’incrément. On précise une valeur de départ pour cette variable, une valeur de fin et une valeur d’incrément (ou décrément).

La syntaxe de cette instruction est la suivante :

for (compteur; condition; modification du compteur) {

liste d'instructions

}

La séquence d’exécution est la suivante :

1. Le compteur est initialisé.
2. La condition est vérifiée : On passe à « 3 » seulement si la condition est vraie. Sinon, on sort de la boucle;
3. On exécute la liste d’instruction
4. On modifie le compteur puis on retourne à « 2 » sans passer par « 1 »

**Exemple 1**

Afficher **5 fois** un message.

for(int i = 1; i <= 5; i++){

System.out.println("Voici la ligne " + i);

}

Remarquez la déclaration et l’initialisation de la variable i. L’instruction se lit comme suit : pour i variant de 1 à 5 (inclusivement grâce au symbole <=), on affiche le message puis on **incrémente** la variable i de 1 (grâce à i++).

Résultat :

Voici la ligne 1

Voici la ligne 2

Voici la ligne 3

Voici la ligne 4

Voici la ligne 5

**Exemple 2**

Affichez les nombres de 5 à 1

**public** **void** afficher5F1() {

**for** (**int** i = 5; i >= 1; i--) {

System.*out*.print (i + "; ");

}

}

Remarquez l’utilisation de l’opérateur pour décrémenter (i--).

Résultat :

5; 4; 3; 2; 1;

**Exemple 3**

**int** colonne = 0;

**for** (**int** i = 9; i >= 1; i--) {

**if** (colonne < 3) {

colonne++;

}

**else** {

colonne = 1;

//saut de ligne

System.*out*.println();

}

// imprimer la valeur de i et un espace

System.*out*.print(i + " ");

}

Résultat :

9 8 7

6 5 4

3 2 1

## Instruction while

La boucle while représente un autre moyen d'exécuter plusieurs fois la même série d'instructions.

La syntaxe est la suivante :

while (condition) {

//liste d'instructions

}

Cette instruction exécute la liste d'instructions **tant que** la **condition** est **vraie**.

**Exemple 1**

Voici un exemple qui permet d’afficher **5 fois** un message.

int fois = 1;

while (fois <= 5) {

System.*out*.println("Voici la ligne " + fois);

fois++;

}

Résultat :

Voici la ligne 1

Voici la ligne 2

Voici la ligne 3

Voici la ligne 4

Voici la ligne 5

Remarquez l’instruction fois++ à l’intérieur de la boucle while, elle est très importante. Il arrive souvent au début que l’on oublie d’incrémenter ou de décrémenter la variable qui sert de condition d’arrêt à la boucle, lorsque cela arrive, le programme est pris dans une **boucle sans fin**. Remarquez également la déclaration et l’initialisation de la variable fois avant de débuter la boucle.

**Exemple 2**

**public** **void** afficher5W1() {

**int** i = 5;

**while** (i >= 1) {

System.*out*.println(i);

i--;

}

}

Résultat :

5; 4; 3; 2; 1;

**Exemple 3**

**public** **void** afficherPar3() {

**int** colonne = 0;

**int** i = 9;

**while** (i >= 1) {

**if** (colonne < 3) {

colonne++;

} **else** {

colonne = 1;

// saut de ligne

System.*out*.println();

}

// imprimer la valeur de i et un espace

System.*out*.print(i + " ");

i--;

}

}

Résultat :

9 8 7

6 5 4

3 2 1

## Instruction do – while

La boucle do while est une variante de la boucle while, où la condition d'arrêt est testée après que les instructions aient été exécutées au lieu d’au début. Ainsi, **la boucle est exécutée au moins une fois**, même si l'expression est fausse la première fois. Dans une boucle while, si l'expression conditionnelle est fausse la première fois, l'instruction n'est jamais exécutée.

La syntaxe est la suivante :

do {

liste d'instructions

} while (condition réalisée);

Voici le même exemple que la boucle while.

**int** fois = 1000;

**do** {

System.*out*.println("Voici la ligne " + fois);

fois++;

}

**while** (fois<=**10**);

Ici, même si la valeur fois est à 1000 en partant, le message s’affichera une fois avant de sortir.

Résultat :

Voici la ligne 1000

# Boucles avancées

## Les boucles avec plusieurs conditions

Une boucle peut vérifier des conditions complexes mettant en relation les opérateurs logiques.

while (fois >= 0 && fois<= 10) {

System.*out*.println("Voici la ligne " + fois);

fois++;

}

On s’assure ici que la valeur fois est comprise entre 0 et 10, autrement on ne réalise pas les instructions.

while (fois >10 || fois == 0) {

System.out.println("Voici la ligne " + fois);

fois++;

}

Cette boucle sera exécutée tant que la variable est plus grande que 10, ou égale à 0.

Dans une boucle for, on peut aussi vérifier plusieurs conditions ainsi :

for (int i = 0 ; i < 10 && fin == false ; i++)…

Les opérateurs logiques **AND** (&&), **OR** (||) et **NOT** (!) sont fréquemment utilisés dans les boucles pour composer des conditions.

## Les boucles imbriquées

Une **boucle imbriquée** est une boucle dans une boucle, c’est-à-dire une boucle à l'intérieur du corps d'une autre boucle. Une boucle imbriquée peut s’appliquer à l’instruction for, while ou do – while.

**public** **static** **void** rectangle5Lx3C() {

**for** (**int** ligne = 0; ligne < 5; ligne++) {

**for** (**int** colonne = 0; colonne < 3; colonne++) {

System.***out***.print('\*');

}

System.***out***.println();

}

}

Nous aurions comme résultat cinq lignes de trois étoiles comme ceci :

\*\*\*

\*\*\*

\*\*\*

\*\*\*

\*\*\*

Pour chaque itération de la première boucle (celle en jaune) que l’on nomme **boucle externe**, la deuxième que l’on **boucle interne** (celle en vert) sera exécutée en entier. L'imbrication de boucle est très courante.

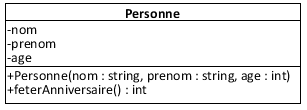
# Introduction à la programmation orientée objet

## Lexique



## Classes et objets

* Une **classe** est la définition d’une structure de données. Par exemple, nous pourrons dire que la classe **Personne** est une structure de données pour laquelle nous aurons des informations et des comportements. Pour celle-ci, on pourrait avoir **nom**, le **prénom** et **l’âge**. Comme comportement, on pourrait avoir « **fêter anniversaire** » qui ferait en sorte d’augmenter l’âge de la personne :



Et l’objet? C’est parfait d’avoir une structure, mais ça prend aussi des **objets** (ou des **instances**). L’objet c’est la matérialisation de la classe. Par exemple, nous voudrons avoir des personnes comme Pierre Carrier qui a 22 ans, Denis Fortin qui à 33 ans, Louise Boulanger qui a 55 ans… Ceux-ci sont les objets de ma classe **Personne**… Ainsi, s’ils fêtent leur anniversaire la même journée, on appliquera la méthode « **fêter anniversaire** » sur chaque objet et cela aura pour effet d’augmenter leur âge. Ainsi, maintenant, Pierre Carrier aura 23 ans, Denis Fortin, 34 ans et Louise Boulanger, 55 ans…

* Exercice :

|  |  |
| --- | --- |
| Nom de la classe | Objet ou instance |
| Voiture(Marque, modèle, couleur, année) | (Audi, Quatro, jaune, 1997)  (Lamborgini, Galarno, rouge, 2007) |
| Maison(SuperficieHabitable, année, prix) | (21245; 1996; 754102)  (12459; 1994; 254000) |

## Définir les attributs (ou les propriétés)

* En Java, nous définissons les caractéristiques de notre classe en définissant des **variables d'instance** (qu’on nomme aussi **attributs**), des variables déclarées immédiatement à l’intérieur de la classe. Le rôle de ces variables est d’emmagasiner l’information propre à notre objet.
* Dans notre cas, nous voulons conserver, le nom, le prénom et l’âge. Voici à quoi ressemble notre classe **Personne** à présent :

**public** **class** Personne {

**private** String nom;

**private** String prenom;

**private** **int** age;

}

* Observez le mot « class » devant Personne. Il signifie que nous sommes à définir une classe qui portera le nom Personne.
* Par convention, le nom d’une classe est un nom commun **débutant** par une lettre en **majuscules** et **sans accent** et qui représente bien ce que nous voulons représenter.
* Le mot « **private**» devant les attributs signifie que les données de la classe sont protégées, elles sont comme confidentielles et cachées pour l’extérieur. Pour les obtenir ou les modifier, il faudra utiliser des méthodes. La plupart du temps, les caractéristiques d’un objet sont privées.
* Immédiatement après le mot **private**, nous rencontrons le **type de données** que l’on veut emmagasiner.
* Finalement, le mot qui suit le type de données constitue le **nom de notre attribut**.

## Classes et instances

* **Instancier** une **classe** consiste à créer un **objet** sur son modèle. C’est pourquoi on dit une **instance** (ou objet) d’une classe.
* On appelle un **objet** qui est produit à partir d’une classe une instance de classe ou simplement une instance. Il faut bien distinguer les 2 notions, on retient :
  + Une classe est un plan ou un modèle qui indique au compilateur comment créer des objets, les instances. (Exemple : **Personne**)
  + Un objet créé à partir d’une classe est une instance. Chaque instance est une entité en mémoire qui contient toute l’information définie par la classe. On peut créer un très grand nombre d’instances d’une même classe (selon la mémoire disponible). Chaque instance possède ses propres données, mais elles ont toutes les mêmes variables. (Exemple : Denis Fortin)

## Les méthodes

* Une méthode est toujours déclarée à l’intérieur d’une classe et se définit de la façon suivante :

**<accessibilité> <type de retour> <nomDeLaMethode>(<type de paramètre> <paramètre>,…)**

public int maMéthode()

{

//instructions…

return 0; // La méthode doit retourner un entier

}

public void maSecondeMéthode(int monPremierParam, String monSecondParam)

{

//instructions qui utilisent monPremierParam et monSecondParam

}

**accessibilité**

D’abord on trouve l’**accessibilité** (ou **visibilité**). On verra plus loin les différentes visibilités, mais pour l’instant on utilise la visibilité **public** qui indique que la méthode peut être utilisée par n’importe qui, depuis n’importe où. Pour l’instant, on vous demande de toujours mettre la visibilité **public**.

**type de retour**

Après la visibilité on retrouve toujours un **type de retour**. Ce type représente comment l’information est fournie par la méthode. Ici **maMéthode()** retourne un entier (int) alors que **maSecondeMethode(…)** ne retourne rien (void). En java, il est obligatoire de mettre le mot clef voidsi la méthode ne retourne rien. La seule méthode qui ne doit pas avoir de type de retour est le constructeur (que nous verrons plus bas).

**nomDeLaMethode**

Après le type de retour, on retrouve le **nom de la méthode**. Le nom d’une méthode est toujours accompagné de ().

Le nom de la méthode suit les mêmes règles que les noms de variables :

* + - il commence généralement par un verbe actif.
    - le nom doit être écrit en minuscule, sauf pour les initiales de chaque nouveau mot
    - un nom de méthode peut comporter des chiffres, mais pas pour le premier caractère
    - les caractères spéciaux \_ et $ peuvent être utilisés, mais ne devraient pas l'être
    - le nom de la méthode est sensible à la casse (différence entre les minuscules et majuscules)

Après le type de retour, on retrouve le **nom** de la méthode.

**(<type de paramètre> <paramètre>,…)**

Le nom d’une méthode est toujours accompagné de (). À l’intérieur de ces parenthèses, on place des variables qui servent à fournir des données à la méthode. On les appelle des paramètres. Les paramètres sont ordonnés. Le langage fait une correspondance entre les variables passées lors de l'appel et les paramètres attendus grâce à l'ordre défini par la méthode.

Le nombre et le type d'arguments dans la déclaration, le prototype et dans **l'appel** doit correspondre, au risque, sinon, de générer une erreur lors de la compilation...

**instructions**

On retrouve finalement une suite d’instructions entre des accolades, c’est le code qu’exécutera la méthode.

### Appel de méthodes

* Pour exécuter une méthode, il suffit de faire appel à elle en écrivant l'objet auquel elle s'applique (celui qui contient les données), le nom de la méthode (en respectant la casse), suivie de ses arguments entre parenthèses:

objet.nomDeLaMethode(argument1,argument2);

* Remarques :
  + Si vous exécutez une méthode sur l'objet courant (this), c'est-à-dire que vous utilisez dans une classe, une méthode de la même classe, alors il est inutile de préciser que this est l'objet auquel s'applique la méthode.
  + Si jamais vous avez défini des arguments dans la déclaration de la méthode, il faudra veiller à les inclure lors de l'appel de la méthode (le même nombre d'arguments séparés par des virgules), mais sans avoir à préciser leur type.
  + Un argument peut être :
    - une constante
    - une variable
    - une expression
    - une autre méthode retournant une valeur

### Renvoi d'une valeur par une méthode

* La méthode peut renvoyer une valeur (et donc se terminer) grâce au mot-clé return. Lorsque l'instruction return est rencontrée, la méthode évalue la valeur qui la suit, puis la renvoie au programme appelant (classe à partir de laquelle la méthode a été appelée).
* Une méthode peut contenir plusieurs instructions return, ce sera toutefois la première instruction return rencontrée qui provoquera la fin de l'exécution de la méthode et le renvoi de la valeur qui la suit.
* La syntaxe de l'instruction return est simple :

return valeurDeRetour;

### Le constructeur

* Le premier comportement que l’on veut programmer c’est de donner à l’objet des valeurs intéressantes dès sa création. Par exemple, on aimerait pouvoir donner des valeurs propres à chacune des 4 personnes. Pour cela, il existe une méthode particulière que l’on nomme un **constructeur**. Le constructeur, c’est la méthode qui donne vie à un objet. Les constructeurs sont des méthodes qui permettent d'initialiser les objets. Pour initialiser notre objet, on doit lui fournir toute l’information nécessaire à la méthode constructeur.
* Par exemple, pour notre classe Personne, le constructeur pourrait se présenter ainsi :

public Personne (String pNom, String pPrenom, int pAge)

Encore une fois, certains mots peuvent sembler inconnus. Voici quelques explications : si les caractéristiques (variables d’instances ou attributs) vues précédemment sont habituellement privées, les méthodes sont, pour leur part, la plupart du temps publiques, car c’est à partir des méthodes que nous pouvons agir sur l’objet. C’est ce qui explique le mot « public » au début de notre méthode spéciale du constructeur.

* Pour le constructeur le **nom** est **toujours le même que le nom de la classe** (et ce en majuscules, c’est la seule exception aux noms de méthode).
* Voyons, maintenant comment écrire notre constructeur. Il faut pour cela inscrire l’information reçue en paramètres dans les attributs de notre classe (affectation des variables). Ce qui nous donne ceci :

**public** Personne (String pNom, String pPrenom, **int** pAge) {

System.***out***.println("création d'une instance");

nom = pNom;

prenom = pPrenom;

age = pAge;

}

Remarquez que nous devons fournir le type du paramètre comme pour les variables d’instances. Évidemment, pour inscrire une donnée dans notre attribut **age**, il faut que notre paramètre soit de même type (ici un entier **int**) autrement nous aurons une erreur. On met un « **p** » devant le nom du paramètre pour pouvoir le différentier dans le code avec l’attribut correspondant.

### Définition d'une méthode

* La déclaration d'une méthode se fait selon la syntaxe suivante :

TypeDeRetour nomDeLaMethode(Type1 argument1, Type2 argument2) {

liste d'instructions

}

* Remarques :
  + TypeDeRetour représente le type de valeur que la méthode va retourner, cela peut-être un type primitif, une classe, ou alors le mot-clé void si la méthode ne retourne aucune valeur.
  + le nom de la méthode suit les mêmes règles que les noms de variables :
    - le nom doit être écrit en minuscule, sauf pour les initiales de chaque nouveau mot
    - un nom de méthode peut comporter des chiffres, mais pas pour le premier caractère
    - les caractères spéciaux \_ et $ peuvent être utilisés, mais ne devraient pas l'être
    - le nom de la méthode est sensible à la casse (différence entre les minuscules et majuscules)
  + Les arguments sont facultatifs, mais s'il n'y a pas d'arguments, les parenthèses doivent rester présentes
  + Il ne faut pas oublier de refermer les parenthèses et les accolades
  + Le nombre d'accolades ouvertes (méthode, boucles et autres structures) doit être égal au nombre d'accolades fermées !
  + La même chose s'applique pour les parenthèses, les crochets ou les guillemets !

### Appel de méthodes

* Pour exécuter une méthode, il suffit de faire appel à elle en écrivant l'objet auquel elle s'applique (celui qui contient les données), le nom de la méthode ( en respectant la casse), suivie de ses arguments entre parenthèses:

objet.nomDeLaMethode(argument1,argument2);

* Remarques :
  + si vous exécutez une méthode sur l'objet courant (this), c'est-à-dire que vous utilisez dans une classe, une méthode de la même classe, alors il est inutile de préciser que this est l'objet auquel s'applique la méthode.
  + le point virgule signifie la fin d'une instruction pour distinguer les différents blocs d'instructions
  + si jamais vous avez défini des arguments dans la déclaration de la méthode, il faudra veiller à les inclure lors de l'appel de la méthode (le même nombre d'arguments séparés par des virgules !), mais sans avoir à préciser leur type.
  + On considère que chaque valeur passée en argument est associée dans l’ordre à la variable

### Les arguments d'une méthode

* Il est possible de passer des arguments (appelés aussi paramètres) à une méthode, c'est-à-dire lui fournir une valeur ou le nom d'une variable afin que la méthode puisse effectuer des opérations sur ces arguments ou bien grâce à ces arguments.
* Le passage d'arguments à une méthode se fait au moyen d'une liste d'arguments (séparés par des virgules) entre parenthèses suivant immédiatement le nom de la méthode.
* Le nombre et le type d'arguments dans la déclaration, le prototype et dans l'appel doit correspondre, au risque, sinon, de générer une erreur lors de la compilation...
* Un argument peut être :
  + une constante
  + une variable
  + une expression
  + une autre méthode retournant une valeur

### Renvoi d'une valeur par une méthode

* La méthode peut renvoyer une valeur (et donc se terminer) grâce au mot-clé return. Lorsque l'instruction return est rencontrée, la méthode évalue la valeur qui la suit, puis la renvoie au programme appelant (classe à partir de laquelle la méthode a été appelée).
* Une méthode peut contenir plusieurs instructions return, ce sera toutefois la première instruction return rencontrée qui provoquera la fin de l'exécution de la méthode et le renvoi de la valeur qui la suit.
* La syntaxe de l'instruction return est simple :

return valeurDeRetour;

## Comment instancier ou créer un objet?

* L’opérateur new crée un objet de la classe.
* Dans notre exemple, la classe Personne permet de déterminer ce que les objets de type Personne contiendront comme information. À ce stade, il n’y a aucun objet en mémoire! Nous utiliserons une nouvelle classe **PersonneApplication** et cette dernière utilisera la classe Personne.
* Une application Java démarre toujours dans une méthode spéciale qu’on nomme **main**. Elle doit être inscrite exactement ainsi :

public static void main(String[] args)

* Petit truc : Dans l’éditeur d’**IntelliJ**, tapez **main** et ensuite appuyez sur la touche **TAB**. L’éditeur inscrira la méthode **main** automatiquement.
* Voici la classe PersonneApplication qui permet d’instancier la classe Personne :

**public** **class** PersonneApplication {

**public** **static** **void** main(String[] args) {

Personne Pierre = **new** Personne("Carrier", "Pierre", 22);

Personne Denis = **new** Personne("Fortin", "Denis", 34);

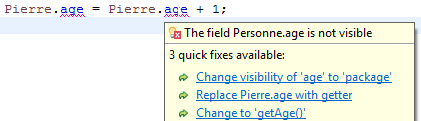
Personne Louise = **new** Personne("Boulanger", "Louise", 55);

}

}

## Accesseurs et mutateurs

* Il est évidant que nous voulions modifier ou consulter la valeur d’un attribut de notre classe à partir d’une Application qui l’utilise. Dans notre exemple, la façon intuitive sera d’utiliser le code suivant pour modifier l’âge de Pierre à partir de PersonneApplication**:**



Cependant, cela cause une erreur d’accessibilité…

Mais c’est normal, nous avons défini l’accessibilité de l’attribut age d’une personne avec une accessibilité **Private**. On rappelle que cette accessibilité autorise l’accès complet de l’attribut **uniquement** à l’intérieur de la classe. Et c’est exactement ce que nous voulions… Ici, nous sommes dans la classe **PersonneApplication** qui est à l’extérieur de la classe **Personne.**

* Pour corriger la situation, nous devrons définir des accesseurs et **mutateurs** pour pouvoir respectivement **consulter** ou **modifier** la valeur d’un attribut à l’extérieur de la classe. Cela se fait en définissant pour chaque attribut private les méthodes set et des méthodes get.
* Ainsi, on ajoutera le code suivant à la classe Personne :

**public** String getNom() {

**return** nom;

}

**public** **void** setNom(String pNom) {

nom = pNom;

}

**public** String getPrenom() {

**return** prenom;

}

**public** **void** setPrenom(String prenom) {

**this**.prenom = prenom;

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

* Ainsi, le code suivant permettra l’accès en consultation et en modification de l’attribut age  :

Pierre.setAge(Pierre.getAge() + 1);

## La méthode toString()

* Dans notre exemple, soit le code suivant :

**public** **class** PersonneApplication {

**public** **static** **void** main(String[] args) {

Personne Pierre = **new** Personne("Carrier", "Pierre", 22);

System.***out***.println(Pierre);

}

}

Qui affiche à la console le résultat incompréhensible suivant :

sem3.Personne@15db9742

* Il serait plus intéressant de voir apparaître de l’information pertinente sur l’objet. Pour cela on doit ajouter la méthode spécialetoString()à la classePersonne. Cette méthode est responsable de retourner une chaîne de caractères qui représente bien l’objet. Elle est donc formée à partir des valeurs des différents attributs de l’objet. Par exemple, pour la classePersonne la méthodetoString() pourrait être :

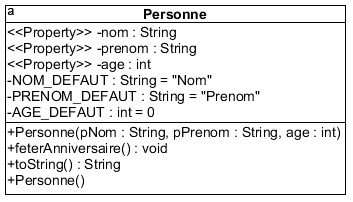
**public** String toString(){

**return** prenom +" "+ nom + " a " + age + " ans" ;

}

## Exemple complet 1 : Classes **Personne** et **PersonneApplication**

* Le Diagramme de Classe :



* Le code des 2 classes :

**public** **class** Personne {

//constantes de classe

**final** **private** String NOM\_DEFAUT = "Nom";

**final** **private** String PRENOM\_DEFAUT = "Prenom";

**final** **private** **int** AGE\_DEFAUT = 0;

//variables d'instance ou attribut

**private** String nom;

**private** String prenom;

**private** **int** age;

**public** Personne() { //Constructeur par défaut

System.***out***.println("création d'une instance par défaut");

nom = NOM\_DEFAUT;

prenom = PRENOM\_DEFAUT;

age = AGE\_DEFAUT;

}

**public** Personne(String pNom, String prenom, **int** age) {

System.***out***.println("création d'une instance");

nom = pNom;

**this**.prenom = prenom;

**this**.age = age; // this.age c'est l'attribut et age, c'est le paramètre.

}

**public** String getNom() {

**return** nom;

}

**public** **void** setNom(String nom) {

**this**.nom = nom;

}

**public** String getPrenom() {

**return** prenom;

}

**public** **void** setPrenom(String prenom) {

**this**.prenom = prenom;

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

**public** **void** feterAnniversaire() {

**this**.age = age + 1;

}

**public** String toString() {

**return** prenom + " " + nom + " a " + age + " ans";

}

}

**public** **class** PersonneApplication {

**public** **static** **void** main(String[] args) {

Personne Pierre = **new** Personne("Carrier", "Pierre", 22);

Personne Denis = **new** Personne("Fortin", "Denis", 34);

Personne Louise = **new** Personne("Boulanger", "Louise", 55);

Personne defaut = **new** Personne();

System.***out***.println(defaut);

System.***out***.println(Pierre);

// Pierre.age = Pierre.age + 1;

Pierre.setAge(Pierre.getAge() + 1);

System.***out***.println(Pierre);

Pierre.feterAnniversaire();

System.***out***.println(Pierre);

}

}

* Sortie à la console :

création d'une instance

création d'une instance

création d'une instance

création d'une instance par défaut

Prenom Nom a 0 ans

Pierre Carrier a 22 ans

Pierre Carrier a 23 ans

Pierre Carrier a 24 ans

## static vs non static

* Le mot-clé static devant une variable (ou méthode) indique que celle-ci n'appartient pas à une instance particulière de la classe. Les variables ou méthodes statiques appartiennent à la classe elle-même. On peut ainsi les utiliser sans avoir une instance créée.
* Par exemple la classe java.lang.Math

System.out.println(Math.PI);

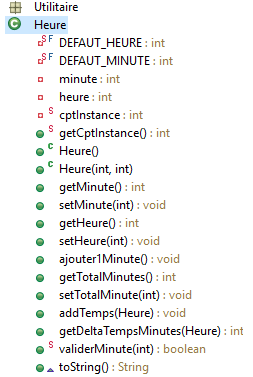
// Affiche la valeur de PI.

System.out.println(Math.abs(-1));

// Affiche la valeur absolue de -1.

* on peut aussi manipuler une variable ou méthode statique à partir d'une instance de la classe ;
* pour rendre des variables statiques comme des constantes, il faut combiner le mot-clé static avec le mot-clé final;
* **Important** : les méthodes statiques, étant indépendantes de toute instance, n'ont pas accès aux variables ou méthodes non statiques.

## Exemple complet 2 : Classes **Heure** et **Intervalle**



**public** **class** Heure

{

//constantes de classe

**private** **final** **static** **int** ***DEFAUT\_HEURE*** = 0;

**private** **final** **static** **int** ***DEFAUT\_MINUTE*** = 0;

//variables d'instance

**private** **int** minute;

**private** **int** heure;

//variables de classe statique

**private** **static** **int** *cptInstance* = 0;

**public** **static** **int** getCptInstance()

{

**return** *cptInstance*;

}

**public** Heure()

{//appelle du constructeur avec arguments (doit être placé en première ligne)

**this**(***DEFAUT\_HEURE***, ***DEFAUT\_MINUTE***);

}

**public** Heure(**int** heure, **int** minute)

{

**this**.setHeure(heure);

**this**.setMinute(minute);

++*cptInstance*;

}

**public** **int** getMinute()

{

**return** **this**.minute;

}

**public** **void** setMinute(**int** minute)

{

**if** (*validerMinute*(minute))

**this**.minute = minute;

**else**

**this**.minute = ***DEFAUT\_MINUTE***;

}

**public** **int** getHeure()

{

**return** heure;

}

**public** **void** setHeure(**int** heure)

{

**this**.heure = heure;

}

**public** **void** ajouter1Minute() {

++minute;

**if** (minute == 60)

{

minute = 0;

++heure;

}

}

**public** **int** getTotalMinutes()

{

**int** totMin = 60 \* heure + minute;

**return** totMin;

}

**public** **void** setTotalMinute(**int** totalMinute) {

**this**.heure = totalMinute / 60;

**this**.minute = totalMinute % 60;

}

**public** **void** addTemps(Heure pH) {

**int** pHTotalMinute;

**int** instanceTotalMinute;

**int** totalMinute;

pHTotalMinute = pH.getTotalMinutes();

instanceTotalMinute = **this**.getTotalMinutes();

totalMinute = pHTotalMinute + instanceTotalMinute;

setTotalMinute(totalMinute);

}

**public** **int** getDeltaTempsMinutes(Heure pH) {

**int** pHTotalMinute;

**int** instanceTotalMinute;

pHTotalMinute = pH.getTotalMinutes();

instanceTotalMinute = **this**.getTotalMinutes();

**return** pHTotalMinute - instanceTotalMinute;

}

//méthode de classe (static):

// Peut être appelé sans que l'on aille à instancier la classe

// Exemple: minuteEstValide = Heure.validerMinute(99);

// Remarque: Cette méthode ne fait appel à aucune variable d'instance

// (ce ne serait pas possible...)

**public** **static** **boolean** validerMinute(**int** pMinute)

{

**return** (0 <= pMinute && pMinute < 60);

}

**public** String toString() {

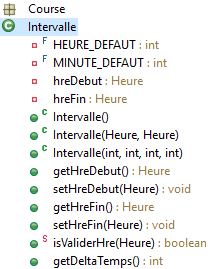
String s;

s = heure + ":" + String.*format*("%02d", minute);

**return** s;

}

}



**package** Course;

**import** Utilitaire.Heure;

**public** **class** Intervalle

{

// constantes de classe

**private** **final** **int** HEURE\_DEFAUT = 0;

**private** **final** **int** MINUTE\_DEFAUT = 0;

// variable l'instance

**private** Heure hreDebut;

**private** Heure hreFin;

**public** Intervalle()

{

**this**.hreDebut = **new** Heure();

**this**.hreFin = **new** Heure(HEURE\_DEFAUT, MINUTE\_DEFAUT);

}

**public** Intervalle(Heure heure1, Heure heure2)

{

**this**.hreDebut = heure1;

**this**.hreFin = heure2;

}

**public** Intervalle(**int** h1, **int** m1, **int** h2, **int** m2)

{

**this**.hreDebut = **new** Heure(h1, m1);

**this**.hreFin = **new** Heure(h2, m2);

}

**public** Heure getHreDebut()

{

**return** hreDebut;

}

**public** **void** setHreDebut(Heure hreDebut)

{

**if** (*isValiderHre*(hreDebut))

**this**.hreDebut = hreDebut;

**else**

**this**.hreDebut = **new** Heure();

}

**public** Heure getHreFin()

{

**return** hreFin;

}

**public** **void** setHreFin(Heure hreFin)

{

**if** (*isValiderHre*(hreFin))

**this**.hreFin = hreFin;

**else**

**this**.hreFin = **new** Heure();

}

**public** **static** **boolean** isValiderHre(Heure hre)

{

**return** hre != **null**;

}

**public** **int** getDeltaTemps()

{

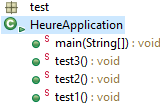
**int** deltaTempsMinutes;

deltaTempsMinutes = hreDebut.getDeltaTempsMinutes(hreFin);

**return** deltaTempsMinutes;

}

}

****

**package** test;

**import** java.util.Scanner;

**import** Course.Intervalle;

**import** Utilitaire.Heure;

**public** **class** HeureApplication

{

**public** **static** **void** main(String[] args)

{

*test3*();

}

**public** **static** **void** test3()

{

Heure h1 = **new** Heure(12, 35);

Heure h2 = **new** Heure(13, 45);

System.***out***.println("h1 : " + h1);

System.***out***.println("h2 : " + h2);

System.***out***.println("Delta temps : " + h1.getDeltaTempsMinutes(h2));

Intervalle i = **new** Intervalle(h1, h2);

System.***out***.println("Delta temps : " + i.getDeltaTemps());

}

|  |
| --- |
| Test 1  Donnez heure et minute :  13  45  Heure = 13:45  Test 2  nb d'instance : 1  nb d'instance : 2  nb d'instance : 3  12:35  13:45  Test 3  h1 : 12:35  h2 : 13:45  Delta temps : 70  Delta temps : 70 |

**public** **static** **void** test2()

{

System.***out***.println("nb d'instance : " + Heure.*getCptInstance*());

Heure h1 = **new** Heure(12, 35);

System.***out***.println("nb d'instance : " + Heure.*getCptInstance*());

Heure h2 = **new** Heure(13, 45);

System.***out***.println("nb d'instance : " + Heure.*getCptInstance*());

System.***out***.println(h1);

System.***out***.println(h2);

}

|  |
| --- |
| Test 2  nb d'instance : 1  nb d'instance : 2  nb d'instance : 3  12:35  13:45 |

**public** **static** **void** test1()

{

**int** heure, minute;

Heure h;

Scanner sc; // Instance de l'utilitaire de lecture

sc = **new** Scanner(System.***in***); // Instanciation de l'utilitaire de lecture

System.***out***.println("Donnez heure et minute :");

heure = sc.nextInt();

minute = sc.nextInt();

**if** (!Heure.*validerMinute*(minute))

{

System.***out***.println("Erreur de minute");

}

**else**

{

h = **new** Heure(heure, minute);

System.***out***.println("Heure = " + h);

}

}

}

|  |
| --- |
| Test 3  h1 : 12:35  h2 : 13:45  Delta temps : 70  Delta temps : 70 |

# Tableaux statiques

* Pour déclarer 100 d’objets de même type faut-il déclarer 100 variables? Par exemple, comment déclarer 100 notes d’étudiants ou encore 100 étudiants…
* Les tableaux statiques sont une structure de taille fixe qui permet de faire la déclaration de plusieurs données de même type en même temps. Qui par exemple nous permettrait de stocker dans un tableau, disons 100 notes ou encore 100 étudiants.
* ATTENTION ! Même si l’on emploie le mot statique pour décrire ce type tableau, il n’a rien à voir avec le mot clef «static» qui signifie une seule version partagée par tous les éléments de la classe. Cela signifie que les tableaux en Java sont des objets qui ont une taille fixée dès leur création, c’est pourquoi on dit qu’ils sont statiques.

## Déclaration du tableau

Le tableau lui-même est un objet et, comme tout objet, pour l’utiliser il faut d’abord déclarer une variable pour tenir une référence verse ce dernier. Pour déclarer une variable contenant un tableau il faut ajouter les crochets [] après le type souhaité pour la variable. En Java les crochets sont réservés exclusivement pour la gestion des tableaux. On déclare un tableau ainsi :

<type du tableau> [] <nom du tableau>;

**Exemples**

**int**[] tabNotes;

Etudiant[] tabEtudiants;

## Création du tableau

Le tableau lui-même est un objet et comme pour tout objet, il faut utiliser l’opérateur new pour allouer la mémoire nécessaire pour le tableau.

**Exemples**

//réserver de l’espace pour 100 entiers pour contenir des notes :

tabNotes = **new** **int**[100];

//réserver de l’espace pour 100 étudiants pour contenir des étudiants :

tabEtudiants = **new** Etudiant[100];

On peut aussi faire les deux instructions en même temps :

<type du tableau>[] <nom du tableau> = new <type du tableau> [taille];

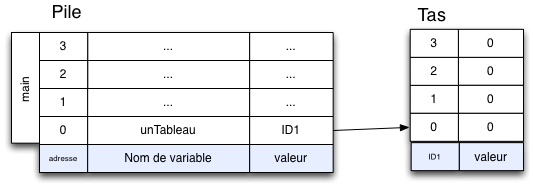
**Exemples**

**int**[] tabNotes = **new** **int**[100];

Etudiant[] tabEtudiants = **new** Etudiant[100];

**Exemples**

int [] unTableau = new int [4];



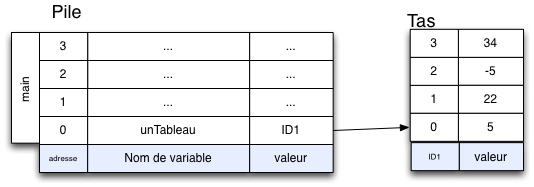
## Initialisation à la déclaration

Il est possible de donner des valeurs à un tableau **au moment de sa déclaration** et **uniquement à ce moment**:

<type du tableau> <nom du tableau> [] = { <contenu du tableau>};

Notez que le compilateur déduit automatiquement la dimension du tableau en fonction du nombre d’éléments présents dans le contenu du tableau.

**int** [] unTableau = {5, 22, -5, 34};



Le tableau sera équivalent à celui obtenu avec:

**int** [] tableau = **new** **int**[4];

Évidemment on peut faire des tableaux avec n’importe quel type, type de base ou type objet. Voici d’autres exemples avec d’autres types :

**double** tableauDouble[] = { 0.0, 1.0, 2.0, 3.0, 4.0 };

**char** tableauCaractere[] = { 'a', 'b', 'c', 'd', 'e', 'f', 'g' };

String tableauChaine[] = { "Line", "Marie", "Paul", "Louis" };

## Accès à un tableau

* On accède au tableau par un indice correspondant à la position d’un élément du tableau. Il est important de noter que le **premier indice est toujours 0**.
* Pour lire ou écrire les valeurs d'un élément du tableau, il faut ajouter, entre crochets ([ ]) à la suite du nom du tableau, l'indice correspondant à la position de l'élément.

**Exemple**

**int** [] unTableau = {5, 22, -5, 34};

unTableau[0] // contient la valeur 5

unTableau[1] // contient la valeur 22

unTableau[2] // contient la valeur -5

unTableau[3] // contient la valeur 34

**Exemple**

**int** [] unTableau = {5, 22, -5, 34};

// placer le contenu du deuxième espace du tableau (22) dans la variable var :

**int** var = unTableau[1];

// placer -10 dans le troisième espace du tableau :

unTableau[2] = -10;

après l’exécution de cette portion de code, on aura :

unTableau[0] // contient la valeur 5

unTableau[1] // contient la valeur 22

unTableau[2] // contient la valeur -10

unTableau[3] // contient la valeur 34

var // contient la valeur 22

## Taille d’un tableau (length)

Tous les objets tableaux ont un attribut length qui renvoie la taille du tableau. Remarquez que puisque la taille du tableau ne peut pas être changée.

// placer le contenu du dernier espace du tableau (34) dans la variable dernierNb

**int** dernierNb = unTableau[unTableau.length - 1];

Remarquez comment on accède au dernier élément de notre tableau qui a la valeur 34. On sait que pour obtenir la taille du tableau, on peut utiliser tableau.length, qui nous retourne la valeur 4 puisque nous avons 4 éléments. Comme le premier élément est logé à l’indice 0, le dernier élément est logé à tableau.length – 1 puisque dans notre exemple notre premier élément est logé à 0, le second à 1 et ainsi suite jusqu’au dernier qui est logé à l’indice 3.

## Tableau d’objets

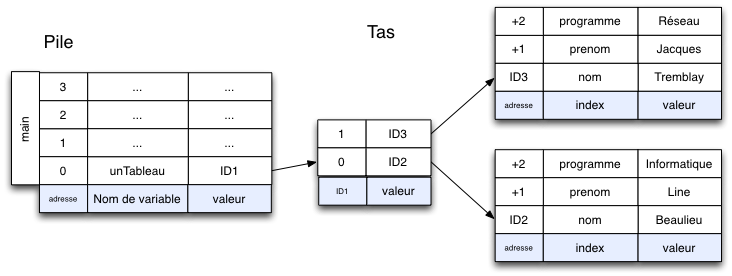
**Exemple**

Etudiant[] classe = **new** Etudiant[2];

// Ajouter les étudiants dans le tableau

classe[0] = **new** Etudiant("Line", "Beaulieu", "Informatique");

classe[1] = **new** Etudiant("Jacques", "Tremblay", "Réseau");



## Parcourir un tableau à l’aide d’une boucle

L’exemple suivant permet d’afficher tous les éléments du tableau unTableau :

**int**[] unTableau = { 5, 22, -5, 34 };

**for** (**int** i = 0; i < unTableau.length; i++) {

System.***out***.println(unTableau[i] + " ");

}

Résultat :

5, 22, -5, 34,

## Passer un tableau en arguments d’une méthode

Les tableaux, étant des types à part entière, peuvent être utilisés comme paramètre pour une méthode. On le spécifie ainsi :

public void methode(int [] tableau)

La méthode a accès ensuite au tableau et aux éléments de ceux-ci, par exemple :

**public** **static void** afficher(**int** [] tableau){

**for** (**int** i = 0; i < tableau.length ; i ++) {

System.***out***.print(tableau[i] + ", ");

}

}

Si on appelle cette méthode avec…

**int**[] unTableau = { 5, 22, -5, 34 };

afficher (unTableau);

Voici ce qui sera affiché :

5, 22, -5, 34,

**Exemple :**

**public static** **void** multiplierTableauParDeux(**int** [] tableau){

**for** (**int** i = 0; i < tableau.length ; i ++) {

tableau[i] \*= 2;

System.***out***.print(tableau[i] + ", ");

}

}

Si on appelle cette méthode avec…

**int**[] unTableau = { 5, 22, -5, 34 };

*multiplierTableauParDeux*(unTableau);

System.***out***.print("\n");

*afficher* (unTableau);

Voici ce qui sera affiché :

10, 44, -10, 68,

10, 44, -10, 68,

**Exemple**

Dans l’exemple, ci-dessous, on modifie un objet dans le tableau.

**public** **static** **void** changerProgrammeEtudiant(Etudiant [] cohorte){

**for** (**int** i = 0; i < cohorte.length ; i ++) {

cohorte[i].setProgramme("Informatique");

System.***out***.println(cohorte[i]);

}

}

Si on appelle cette méthode avec…

Etudiant[] classe = **new** Etudiant[2];

// Ajouter les étudiants dans le tableau

classe[0] = **new** Etudiant("Line", "Beaulieu", "Géomatique");

classe[1] = **new** Etudiant("Jacques", "Tremblay", "Réseau");

*changerProgrammeEtudiant*(classe);

Voici ce qui sera affiché :

Etudiant [nom=Beaulieu, prenom=Line, programme=Informatique]

Etudiant [nom=Tremblay, prenom=Jacques, programme=Informatique]

## Tableau comme valeur de retour d’une méthode

On spécifie une méthode avec valeur de retour de cette façon :

public int[] getTableau()

{ ... }

Dans l’exemple ci-dessous, on reçoit en paramètres 3 entiers et on veut retourner un tableau composé de ces trois valeurs :

**public** **static int**[] creerTableau(**int** n1, **int** n2, **int** n3) {

**int**[] tableau = { n1, n2, n3 };

**return** tableau;

}

Si on appelle cette méthode avec…

**int**[] unTableau = *creerTableau*(30, 40, 50);

*afficher*(unTableau);

Voici ce qui sera affiché :

30, 40, 50,

Dans cet autre exemple, on reçoit un tableau d’entiers et on veut retourner un tableau constitué uniquement des éléments pairs.

**public** **static int**[] creerTableauPair(**int**[] tableau) {

**int**[] tableauPair = **new** **int**[tableau.length];

**int** indice = 0;

**for** (**int** i = 0; i < tableau.length; i++) {

**if** (tableau[i] % 2 == 0) {

tableauPair[indice] = tableau[i];

indice++;

}

}

**return** tableauPair;

}

Si on appelle cette méthode avec…

**int**[] unTableau = { 5, 22, -5, 34 };

**int**[] unLesPairsDuTableau;

unLesPairsDuTableau = *creerTableauPair* (unTableau);

*afficher*(unLesPairsDuTableau);

Voici ce qui sera affiché :

22, 34, 0, 0,

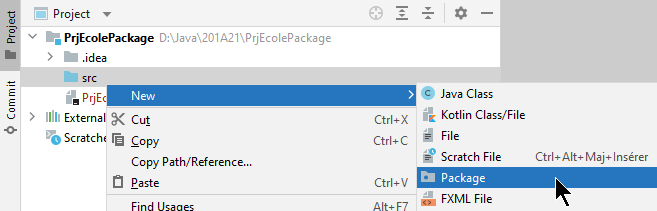
# Packages

Bien entendu on peut mettre tous les fichiers de classe au même endroit, mais, au fur et à mesure que le projet prend de l’ampleur, ça risque d’engendrer un beau fouillis. Pour éviter cela, le java nous offre un mécanisme de paquetage. Les packages fonctionnent exactement comme les dossiers dans l’explorateur de fichiers. On peut créer toute une arborescence de paquetages. Les classes java seront placées librement, mais intelligemment dans l’arborescence de dossiers.

Voici comment créer un package dans IntelliJ

*Remarquer que les noms de package commencent toujours par une lettre minuscule*

Dans l’explorateur de fichier vous obtiendrez alors :



Remarquez que si vous ne spécifiez pas de package lors de la création d’une classe, cette dernière se retrouvera automatiquement dans le package *(default package).* Oracle recommande de toujours spécifier un package et de ne jamais utiliser le package par défaut.

Notez également que notre projet comporte maintenant 2 classes Etudiant. Java le permet, car les 2 classes ne sont pas dans le même package. Comment le compilateur fait-il pour distinguer à quel type d’Etudiant a-t-on affaire?

* Lorsqu’une classe n’est pas dans le package defaut, sa première ligne de code doit être la déclaration de son package. Par exemple

package ecole;  
  
public class Etudiant {  
}

* Par défaut une classe peut utiliser directement (sans qu’on n’ait rien à faire de plus) toutes les classes faisant partie du même package qu’elle.
* Si une classe doit utiliser une classe qui fait partie d’un autre package il faut préciser au compilateur où trouver la classe. Pour cela, on utilise l’instruction suivante au début de la classe immédiatement après la déclaration du package:

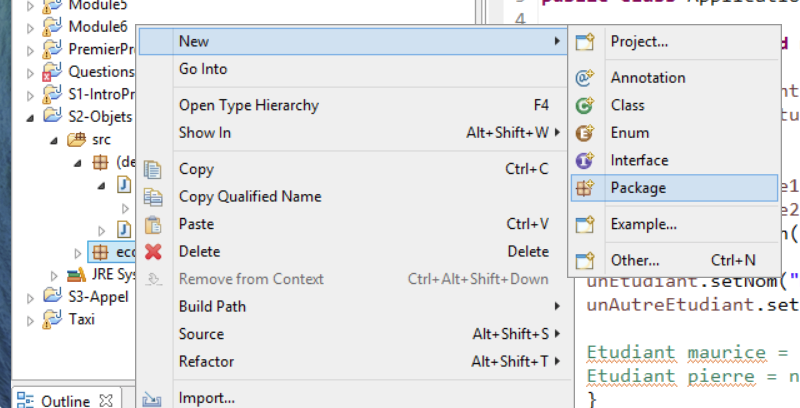
**package** lecon;

**import** ecole.Etudiant;

**public** **class** Application {…}

Ici, en plaçant cette ligne au début du fichier *Application.java,* on vient spécifier que la classe *Etudiant* utilisée dans le code de la classe *Application* est celle qui se trouve dans le package *ecole*. (ici Application se trouve dans le package *lecon*).

Pour créer un package directement dans Eclipse il suffit de cliquer avec le bouton droit de la souris (menu contextuel) directement à l’endroit où l’on veut mettre le nouveau package. Si l’on veut rajouter un package locaux dans le package ecole :



Par exemple, si nous créons une classe *Bureau* dans le nouveau package *locaux,* nous verrons dans l’explorateur de projet :

|  |  |
| --- | --- |
| présentation hiérarchique | présentation plate (flat) |

Pour changer le type de présentation cliquer sur le triangle dans le coin supérieur droit de l’explorateur de projet et changez le mode « package presentation ».

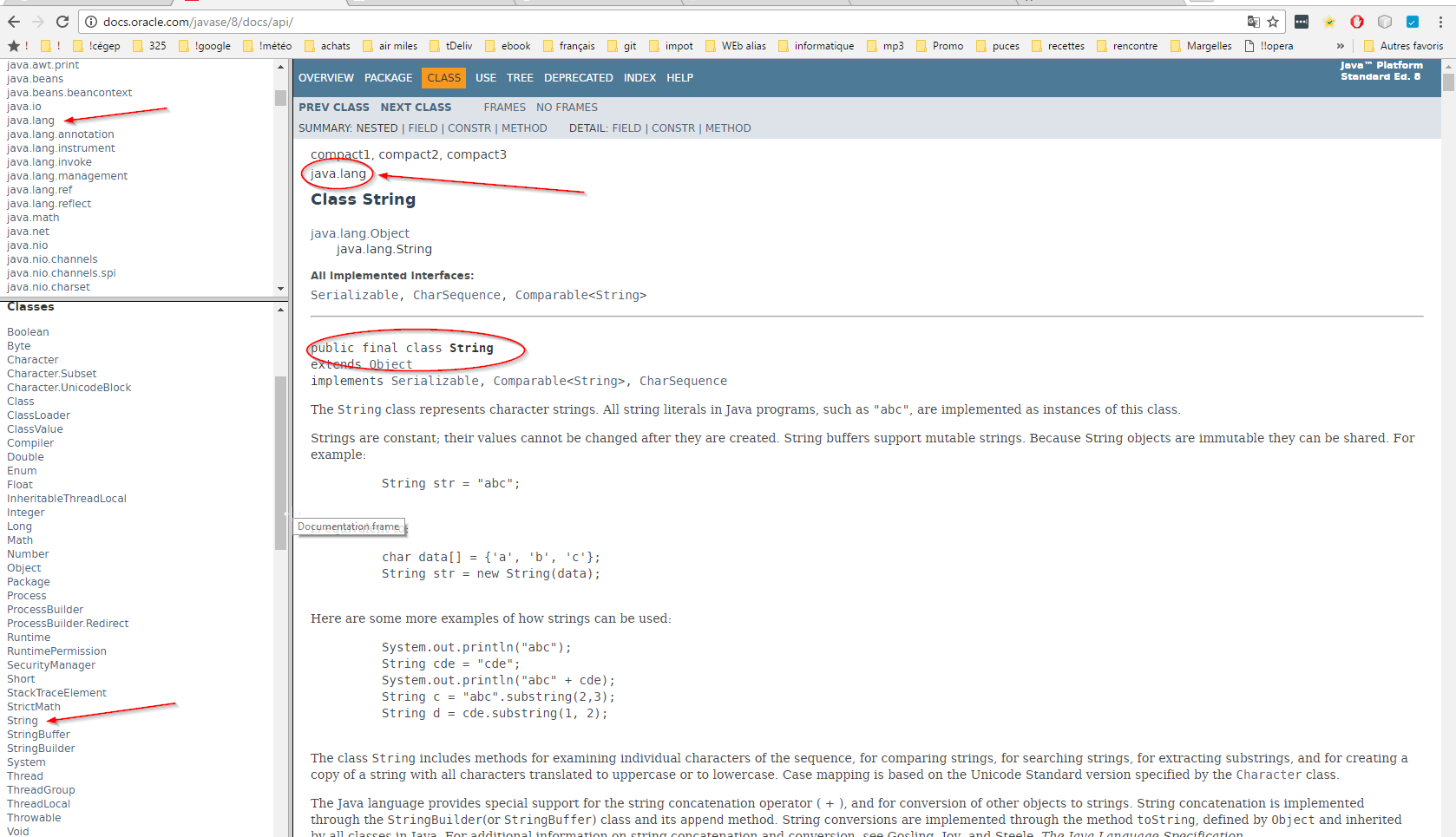
# API Java

* Une interface de programmation applicative (souvent désignée par le terme **API** pour **Application Programming Interface**) est un ensemble normalisé de classes, de méthodes ou de fonctions qui sert de façade par laquelle un logiciel offre des services à d'autres logiciels. Elle est offerte par une bibliothèque logicielle, le plus souvent accompagnée d'une description qui spécifie comment des programmes consommateurs peuvent se servir des fonctionnalités du programme fournisseur.
* L’API Java est disponible sur le WEB en accédant à :

<http://docs.oracle.com/javase/8/docs/api/>

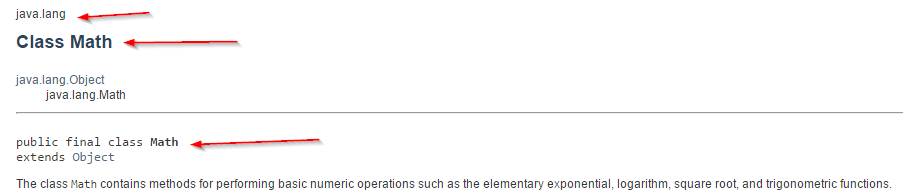
Exemple 1

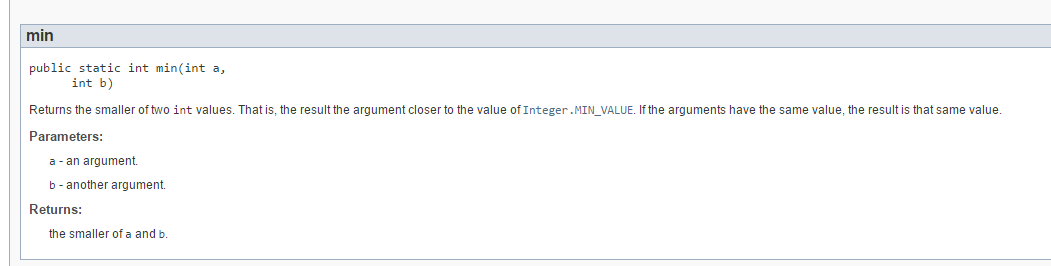
Voici la description de la classe String dans le package java.lang qui est un package accessible par défaut dans les classes. C’est-à-dire que vous n’avez pas à placer import java.lang au début de votre fichier pour avois accès à ses fonctionnalités :



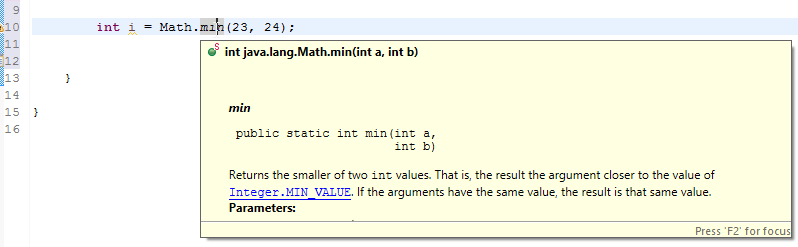
Exemple 2

Voici la description de la classe Math dans le package java.lang





Notez que c’est exactement la même description qui sera affichée si vous placez le curseur au-dessus du code dans Eclipse :



# Documentation

* Documenter vos programmes est essentiel pour faciliter la maintenance et l’évolution de votre code. Selon Robert C. Martin, auteur du livre « **Clean Code** » : « Apprendre à écrire du code clair est difficile. Cela demande plus que la connaissance de principes et de bonnes pratiques. Vous devez l’adopter, le pratiquer par vous-mêmes et voir vos propres échecs. Vous devez aussi observer d’autres pratiques, lire du code et constater ce qui distingue le beau code du code laid. ».

## Convention des noms dans le code Java

Important :

* Tenez-vous en à une des deux grandes règles de composition : majuscules à chaque nouveau mot de la composition (ageDuCapitaine) ou soulignés (age\_du\_capitaine).
* Ne pas mettre d’accent.
* Si ça simplifie sans compromettre la compréhension du code, l’anglais est accepté. Exemple : getNom()

Voici quelques règles pour le choix de vos noms :

|  |  |  |
| --- | --- | --- |
| **Type** | **Règles de nommage** | **Exemples** |
| **Variables** | Première lettre en minuscules  Les variables à une seule lettre sont permises uniquement pour un usage local (dans une boucle for, par exemple) par exemple i ou j. | int i; float largeur; String nomDuCapitaine; |
| **Méthodes** | Première lettre en minuscules  Les noms de méthodes (fonctions) doivent refléter une action | afficher(); getValue(); setValue(); |
| **Constante** | Les noms des constantes sont écrits en majuscules et on sépare les mots par le trait de soulignement « \_ » | TAUX\_DE\_TAXE |
| **Classes** | Première lettre en majuscule. | class Image;  class ClientPrivilegie |
| **packages** | Mots en minuscule | facture |

## Règles pour les méthodes

* Elles doivent être courtes. Ne pas dépasser 80 caractères de large et un maximum de 24 lignes.
* Faire une seule chose, et uniquement cette chose.
* Bien utiliser les modificateurs d’accès private et public.
* Lire le code du début vers la fin.
* Le nombre d’arguments des méthodes ne doit pas être trop élevé. Autrement, il faut penser créer un objet contenant toutes les informations nécessaires.
* Être structuré, une seule instruction de retour (return) par méthode.

**public** **double** calculerSalaire(Employe employe)

{

**double** salaire = 0;

**if** (employe.getType() == *COMMISSION*)

{

salaire = calculerSalaireCommission(employe);

}

**else** **if** (employe.getType() == *HORAIRE*)

{

salaire = calculerSalaireHoraire(employe);

}

**else** **if** (employe.getType() == *SALARIE*)

{

salaire = calculerSalaireSalarie(employe);

}

**return** salaire;

}

## L’espacement entre les opérateurs

On utilise les espaces pour mettre en évidence les éléments associés ou dissociés. Par exemple, pour bien voir l’affectation et les opérateurs.

int produit=valeurA\*valeurB; //mauvais

comparativement à…

int produit = valeurA \* valeurB; //bon

## L’indentation ou formatage du code

Les lignes doivent être indentées de manière à mettre les blocs de code en valeur. Observez la différence de lisibilité entre les deux méthodes ci-dessous :

**public** **double** calculerSalaire(Employe employe) {

**double** salaire = 0;

**if** (employe.getType() == *COMMISSION*) {

salaire = calculerSalaireCommission(employe);} **else** **if** (employe.getType() == *HORAIRE*) {

salaire = calculerSalaireHoraire(employe);

} **else** **if** (employe.getType() == *SALARIE*) { salaire = calculerSalaireSalarie(employe); }

**return** salaire;}

comparativement à…

**public** **double** calculerSalaire(Employe employe)

{

**double** salaire = 0;

**if** (employe.getType() == *COMMISSION*)

{

salaire = calculerSalaireCommission(employe);

}

**else** **if** (employe.getType() == *HORAIRE*)

{

salaire = calculerSalaireHoraire(employe);

}

**else** **if** (employe.getType() == *SALARIE*)

{

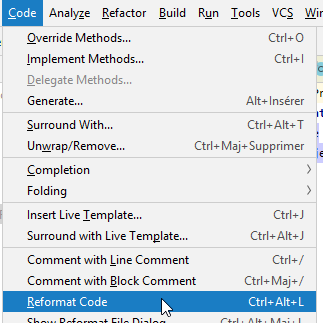
salaire = calculerSalaireSalarie(employe);

}

**return** salaire;

**}**

L’indentation est quelque chose qui est fréquemment déplacé au fur et à mesure qu’on ajoute ou qu’on retire du code. Tout comme pour la modification des noms, il n’est pas intéressant que le programmeur perde du temps continuellement à remettre en forme le code. **IntelliJ** offre une commande qui permet d’un seul clic de refaire l’indentation du code. Il suffit d’utiliser la combinaison de touche CTRL+ALT+L[[1]](#footnote-1).



## Les commentaires

Le langage Java possède trois styles de commentaires.

* Commentaire sur une ligne
* Commentaire sur plusieurs lignes
* Commentaire Javadoc

### Commentaire sur une ligne

Tout ce qui se trouve après la double barre oblique « // » constitue un commentaire et est ignoré par le compilateur. Pour éviter la profusion de ce type de commentaire qui peut alourdir considérablement le code, il suffit d’utiliser des noms de variables et de méthodes appropriés.

//calcule le prix total

double total = valeurA \* valeurB;

double totalPrix = prix \* quantite;

Que préférez-vous?

|  |  |
| --- | --- |
| **Ceci?** | **ou ceci?** |
| public class RapportConfiguration  {  private String propriete1; //orientation  private String propriete2; //largeur  private String propriete3; //longueur  } | public class RapportConfiguration  {  private String orientationPapier;  private String largeurPapier;  private String longueurPapier;  } |

### Commentaire sur plusieurs lignes

Tout le texte entre ces caractères /\* \*/ constitue un commentaire qui peut s’étendre sur plusieurs lignes.

/\*

\* Permet de tracer un triangle isocèle selon la hauteur du triangle

\* On affiche des étoiles pour dessiner le triangle

\*/

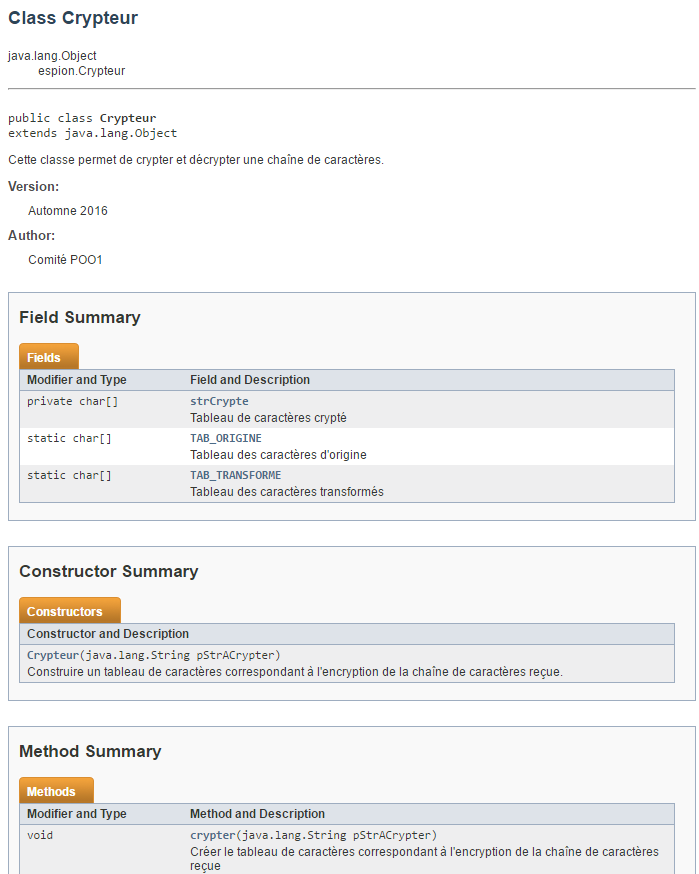
for (int i = 0; i <= h; i++)

{

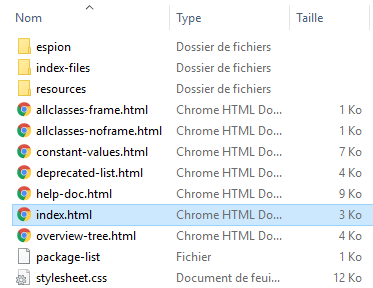
nbEspaces = …

# La Javadoc

Les commentaires présentés entre les caractères /\*\* \*/ sont spécialement traités par le programme javadoc et génèrent un fichier **HTML**. Ils servent à documenter une classe ou une méthode. Voici un exemple de commentaires générés par la javadoc :



La génération de la **Javadoc** produit un ensemble de fichiers plus ou moins long qui peut ressembler à celui-ci :



Voyons maintenant comment construire ces commentaires.

## Commentaires Javadoc

Les commentaires à la manière JavaDoc sont écrits en HTML et doivent précéder directement la **classe**, les **attributs** ou les **méthodes** auxquels ils s’adressent. Les commentaires se décomposent en quatre parties :

1. les caractères de début /\*\* qui indiquent le début des commentaires Javadoc;
2. la description qui commence immédiatement après les caractères de début et se termine à la première balise rencontrée;
3. une liste de balises débutant par le symbole @ et un mot réservé Javadoc;
4. les caractères \*/ qui indiquent la fin des commentaires Javadoc.

**Exemple**

/\*\*

\* Valide le déplacement d'une Tour dans un jeu d'échec.

\* **@param** origine : Point d'origine de la pièce à déplacer

\* **@param** fin : Point de destination de la pièce à déplacer

\* **@return** vrai(true) si le mouvement d'échec est valide ou

\* faux(false) sinon

\*/

**public** **boolean** deplacer(Point origine, Point fin)

{

**boolean** estDeplace = **true**;

// ...

**return** estDeplace;

}

## Balises Javadoc

Le tableau ci-dessous présente les balises **Javadoc** :

|  |  |  |  |
| --- | --- | --- | --- |
| Balise | Description | Remarque | Où |
| @param | documente les paramètres des méthodes. | si plus d’un paramètre mettre plusieurs balises dans l’ordre des paramètres | méthode |
| @return | documente ce que renvoie une méthode |  | méthode |
| @author | indique l’auteur (mettre plusieurs balises si plus d’un auteur). | pour les classes, seulement | classe |
| @version | précise la version du code source de la classe ou de la méthode. |  | classe |
| @exception | documente les exceptions lancées par la méthode. |  | méthode |
| @see | permet de créer une entrée dans la section « see also » de la documentation pour référer à des classes ou méthodes apparentées. |  | méthode et classe |

## **Javadoc** d’une classe

La documentation d’une classe comprend les éléments suivants :

* La présentation de la classe (son but ou son utilité)
* La balise @version pour indiquer la version du programme et la date.
* La balise @author pour mentionner le créateur de la classe.
* La balise @see pour faire des références à d’autres classes s’il y a lieu.

Exemple : Un début de classe

/\*\*

\* Classe TempsJ

\* Cette classe gère le temps au format 24 heures.

\*

\* **@see** java.lang.Object

\* **@author** Julie Frève

\* **@version** 1.3 : 22 mai 2002

\*

\*/

## **Javadoc** d’une méthode

La documentation d’une **méthode** comprend les éléments suivants :

* La présentation de la méthode (son rôle).
* La balise @param pour chaque paramètre d’entrée de la méthode s’il y a lieu.
* La balise @return pour indiquer ce que retourne la méthode s’il y a lieu.
* La balise @throws pour préciser les exceptions que la méthode lève s’il y a lieu. (Les exceptions seront vues plus tard dans le cours)
* La balise @see pour réaliser un lien hypertexte avec une méthode apparentée s’il y a lieu.

Exemple

/\*\*

\* Concatène une chaîne à la chaîne en cours.

\* Le résultat est une nouvelle chaîne.

\* **@param** chaine : la chaîne à concaténer

\* **@return** la chaîne résultante

\*/

**public** String concat(String chaine)

{

String strConcatene = "";

// ...

**return** strConcatene;

}

## **Javadoc** d’une constante et d’une variable d’instance (attribut)

Si on veut que les **constantes** et les **variables d’instance** se retrouvent dans la documentation html, on doit faire un commentaire **javadoc** pour chacun comme suit :

/\*\*

\* Nombre d'essais par défaut

\*/

**public** **static** **final** **int** ***NB\_ESSAIS\_DEFAUT*** = 0;

/\*\*

\* Nombre aléatoire à deviner

\*/

**private** **int** nbAleatoire;

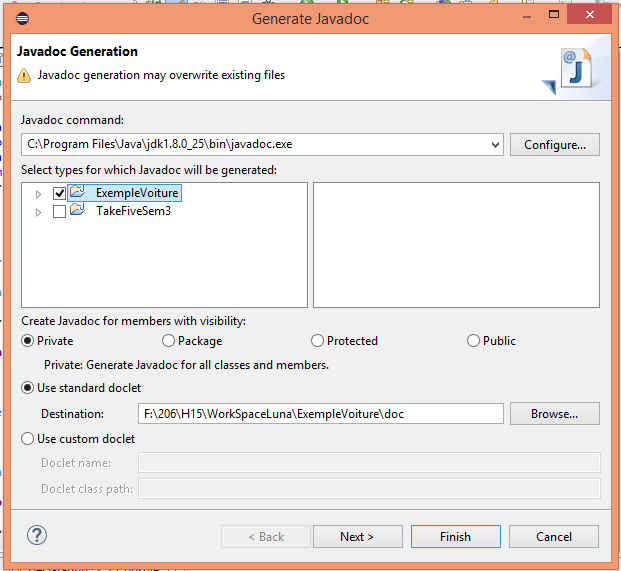
## **Générer la JavaDoc**

Une fois l'application terminée, la documentation est créée automatiquement grâce au programme exécutable javadoc.exe. La documentation HTML est générée dans le dossier doc du projet. Ce programme pourrait se trouver dans :

C:\Program Files\Java\jdk1.8.0\_20\bin\javadoc.exe

### Générer la JavaDoc avec **Eclipse**

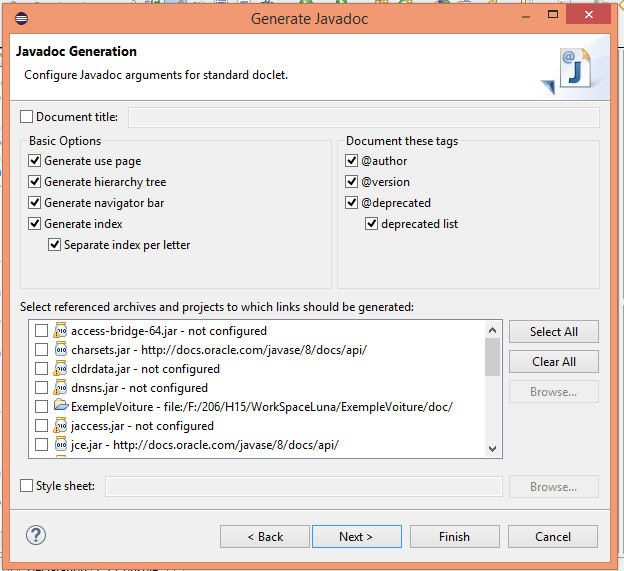
Avec Eclipse, la génération de la documentation est simple puisque vous n'avez pas à fournir la commande, mais à activer le menu « **Project/Generate JavaDoc...** ».



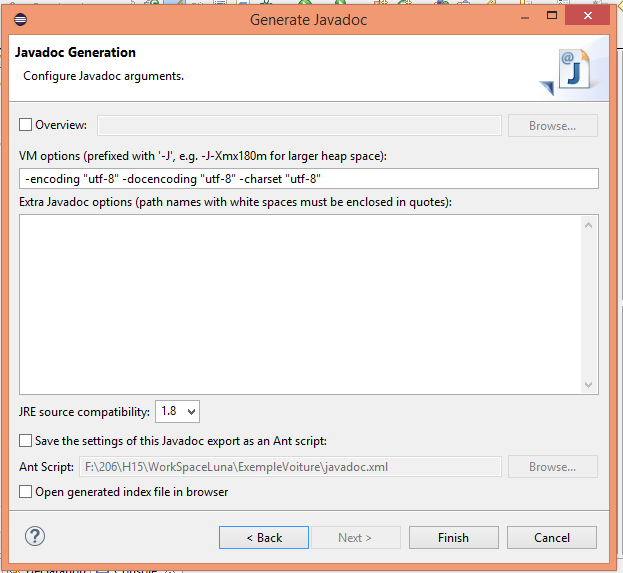
1. Vous devez remplir le champ « **Javadoc Command** » avec le chemin d’accès de l’exécutable **javadoc** de votre JDK. Ce chemin pourrait être…

C:\Program Files\Java\jdk1.8.0\_20\bin\javadoc.exe

1. Cocher le projet désiré
2. Choisir la visibilité (**public** ou **private**)
3. Choisir le dossier destination qui en principe devrait être dans votre projet.
4. Faire **Next**



1. Cocher tout dans la partie du haut
2. Ne rien cocher dans la partie du bas
3. Faire **Next**



1. Copier / Coller la ligne suivante la boîte texte VM options :

-encoding "utf-8" -docencoding "utf-8" -charset "utf-8"

1. Faire **Finish**

La documentation Java de toutes les classes du projet sera dans le dossier doc. En ouvrant index.html, on a la **javaDoc** entière.

# Tableaux dynamiques

Les tableaux dynamiques font partie de l’offre de Java pour gérer les collections d’objets. Ils ne peuvent pas accueillir de primitives comme les int, boolean, char ou double. Les données emmagasinées dans un tableau dynamique sont nécessairement des objets. La classe ArrayList permet d’insérer autant d’objets que souhaité dans le contenant, car elle est extensible, c’est la raison pour laquelle on appelle cette structure, un tableau dynamique. La gestion d’un tableau dynamique est différente de celle d’un tableau statique. En effet, puisque c’est une classe, les actions à poser sur les éléments de la classe se font généralement au moyen d’une méthode. Voyons comment travailler avec la classe ArrayList.

## Déclaration d’une ArrayList

Voici la syntaxe pour déclarer un tableau dynamique de type ArrayList :

ArrayList <<type>> <nomListe>;

Exemple :

ArrayList <String> maListe;

ArrayList <Etudiant> classe;

Même si le type de la liste (String ou Etudiant) n’est pas obligatoire, il est recommandé de le préciser, d’ailleurs Eclipse donne un avertissement si vous ne le faites pas. En précisant le type d'objet contenu dans le tableau, vous permettez au compilateur de détecter plus d'erreurs dans votre code. Ceci sera expliqué plus loin dans ce document.

Comme la classe ArrayList ne fait pas partie du **package standard de Java**, **Eclipse** vous le signale par une erreur avec un petit x rouge dans la marge gauche. Il suffit de cliquer sur le x rouge et d’accepter l’importation de la classe ArrayList. Vous pouvez ajouter vous-même l'instruction d'importation dans le début du fichier (entre l'instruction package et la déclaration de la classe).

import java.util.ArrayList;

## Création d’une ArrayList

Tout comme pour les tableaux statiques, la déclaration d’une ArrayList ne construit pas notre tableau et ne réserve pas l’espace nécessaire. Pour créer le vecteur dynamique, nous devons utiliser l’instruction new comme ceci :

maListe = new ArrayList<>();

Notez que nous ne mentionnons pas de taille. On pourrait le faire, mais comme le tableau est élastique, il s’agrandit automatiquement lorsqu’il n’y a plus de place. Par défaut, Java, génère de l’espace pour dix objets, lorsqu’il arrive au bout de l’espace et qu’on ajoute un objet, il ajoute dix nouveaux espaces.

Remarquez également que nous n’avons pas répété le type de donnée que nous insérons, mais nous avons placé des chevrons vides new ArrayList<>. Depuis la version 7 de java (jre7), Il n’est pas nécessaire de répéter le type dans les chevrons du côté droit (constructeur), le compilateur comprend qu’il doit générer des données de type String puisque nous l’avons déjà déclaré du côté gauche (déclaration de variable). Si vous faites la création ainsi : new ArrayList<String>(), cela est tout à fait accepté par le compilateur, c’est juste plus long à écrire pour le programmeur. Évidemment, si vous devez travailler avec une version de Java antérieure à 7 vous devrez utiliser l'instruction complète new ArrayList<String>().

## Ajouter des éléments *arrayList*.add(…)

Pour ajouter des éléments dans le tableau, nous avons accès à un éventail de méthodes. En voici deux, les plus utilisées. La première permet d’ajouter un élément à la fin du tableau et la seconde ajoute l’élément à la position mentionnée par l’index. Notez que l'opérateur crochet [] ne fonctionne pas avec un ArrayList.

void add(Object element)

void add(int index, object element)

**Exemple**

ArrayList <Etudiant> groupe = new ArrayList<>()

groupe.add(new Etudiant("Line", "Beaulieu",19,"Informatique"));

groupe.add(new Etudiant("Jacques", "Tremblay",22,"Réseau"));

groupe.add(1, new Etudiant("Alain", "Marcoux",25,"Réseau"));

Une fois ces insertions complétées, le tableau groupe contient les éléments suivants :

"Line", "Beaulieu", 19, "Informatique"

"Alain", "Marcoux", 25, "Réseau"

"Jacques", "Tremblay", 22, "Réseau"

Comme dans un tableau statique, le premier élément d’une ArrayList porte l’index **0**. Si on tente d’insérer un élément à une position occupée, le nouvel élément prend sa place et l’ancien est poussé vers la droite. L'élément n'est donc pas écrasé, contrairement à ce qui se passait avec un tableau statique.

C'est ici que la déclaration du type d’éléments contenu dans le tableau à l'aide des chevrons "<>" permet de protéger le code contre des erreurs. Si l'on insère des éléments avec des types différents dans un même tableau, comment savoir lors de la lecture à quel type on va se retrouver? Pensez au balayage du tableau. En fait, utiliser des éléments de types différents dans un même tableau est considéré comme une très mauvaise pratique.

**Exemple**

ArrayList <Etudiant> groupe = new ArrayList<>()

groupe.add(new Carte(5, "pique")); // erreur de compilation une carte n'est pas un étudiant.

Par contre avec…

ArrayList groupe = new ArrayList<>()

groupe.add(new Carte(5, "pique")); // pas d'erreur

groupe.add(new Etudiant("Line", "Beaulieu",19,"Informatique"))); //OK

Un tableau groupe détient maintenant une Carte au lieu d'un Etudiant ! Ça va créer des problèmes un jour ou l'autre, c'est certain!

## Obtenir un élément *arrayList*.get (int index)

Pour obtenir une valeur contenue dans le tableau, on utilise la méthode suivante qui retourne l'élément à l'indice demandé:

public Object get(int index)

**Exemple**

groupe.get(1) -> retourne l’étudiant Alain

## Modifier un élément *arrayList*.set (int index, Object element)

Pour modifier la valeur d’un élément d’une ArrayList, il faut utiliser la méthode suivante :

public Object set(int index, Object element)

**Exemple**

L’étudiant Alain est remplacé par Jeanne. Alain n’existe plus.

groupe.set(1, new Etudiant("Jeanne", "Rioux",22,"Informatique"));

## Supprimer un élément *arrayList*.remove(int index)

La suppression d’un élément dans une ArrayList entraîne plusieurs actions. Une fois l'élément supprimé, la taille du tableau est ajustée, elle est diminuée d’un. De plus, les éléments qui suivent l'élément supprimé remontent pour occuper la position libérée, donc tout le tableau est réorganisé. Voici deux façons de supprimer des éléments dans une ArrayList, l’une permet de supprimer l’élément à l’index fourni alors que l’autre vide la liste entière.

public Object remove(int index)

public void clear()

**Exemples**

Soit le tableau « groupe» suivant : Line – Alain – Jacques.

groupe.remove(0);

supprime Line la nouvelle liste est : Alain – Jacques.

groupe.clear()

Supprime Alain et Jacques, la nouvelle liste est vide.

Attention : Si vous insérez un "null" au lieu d'un objet, vous effacerez alors l'objet du tableau, mais cet espace du tableau ne sera pas récupéré. Le tableau contiendra effectivement la valeur null pour cet indice.

**Exemple**

Soit le tableau « classe » suivant : Line – Alain – Jacques.

groupe.set(1, null);

donnera :

Line – null – Jacques

et non

Line – Jacques

## Obtenir le nombre d’élément dans une une ArrayList (size())

On obtient le nombre d’éléments dans une ArrayList grâce à size().

## Parcourir les éléments d’une ArrayList

Tout comme pour le tableau statique, on parcourt une ArrayList avec une boucle.

### Boucle classique

for(int i=0; i < groupe.size(); i++){

System.out.println(groupe.get(i));

}

### Boucle for-each

Il existe une boucle spéciale, la bouche for-each qui permet un parcours complet sans se préoccuper de l’indice.

La boucle for-each est très utile, car elle évite de s’encombrer d’un index.

Voici sa syntaxe :

for (<type> objet : <ArrayList>)

**Exemple**

for(Etudiant etudiant : groupe){

System.out.println(etudiant);

}

## Transformer un tableau dynamique en tableau statique *arrayList*.toArray()

Dans certaines situations, il est utile de pouvoir convertir un tableau dynamique comme ArrayList en tableau statique, cela se fait grâce à la méthode toArray().

**Exemple**

Object [] tableauEtudiant = groupe.toArray();

# Énumération (enum)

* L’énumération est une variable qui ne peut prendre que certaines valeurs définies d’avance. L’utilisation d’une énumération comporte donc **2 étapes**, la définition de l’énumération et la déclaration des variables qui l’utilise.
* Pour définir une énumération, on utilise le mot clef **enum**. On ajoute ensuite un nom pour identifier l’énumération. La définition finit par un bloc (entre accolades) qui énumère les différentes valeurs permises. Comme les valeurs permises ne peuvent pas changer en cours d’exécution, il est fortement recommandé de mettre les valeurs possibles en majuscules (elles sont comme les constantes). La définition de l’énumération est une sorte de classe et c’est pourquoi on doit respecter les règles pour nommer les classes (première lettre en majuscule). En pratique, plusieurs programmeurs utilisent l’**enum** pour représenter des constantes :

**enum** JourSemaine {*LUNDI*, *MARDI*, *MERCREDI*, *JEUDI*, *VENDREDI*, *SAMEDI*, *DIMANCHE}*

* Notez qu’il n’y a pas de point-virgule après la définition de l’**enum**. Une fois l’**enum** définie, on peut l’utiliser comme type de variable. Cette nouvelle variable ne pourra que prendre les valeurs définies dans la définition de l’**enum**.
* Exemple :

JourSemaine conge = JourSemaine.*LUNDI*;

* (À voir plus tard lorsque la théorie sur les classes sera assimilée) On peut déclarer le type enum au besoin localement dans un fichier, dans une classe. Alors, le type énumération ne sera connu que par cette classe. Si l’on veut étendre la portée de la déclaration d’un type enum il faut la placer dans son propre fichier .
* Exemple :

**public** **enum** JourSemaine {

***LUNDI***, ***MARDI***, ***MERCREDI***, ***JEUDI***, ***VENDREDI***, ***SAMEDI***, ***DIMANCHE***

}

Note : Évidemment, ce fichier devra avoir le même nom que le type enum. Ici ce serait JourSemaine.

## enum.values() (notions avancées)

La méthode values() appliquée à une variable de type enum retourne un tableau d’énumération qui contient **des objets de l’énumération**.

Comme nous avons un tableau d’objets.

* *enum*.values().length retourne le nombre d’éléments dans l’énumération;
* *enum*.values()[i] retourne l’objet énumération à l’indice.

Avec un **objet énumération**, on peut…

* *enum*.values()[i].name() retourne le nom de l’objet énumération à l’indice;
* *enum*.values()[i].ordinal() retourne la position de l’objet énumération dans l’énumération. Le premier objet de l’énumération étant à la position 0.

**Exemple 1**

**for** (**int** i = 0; i < JourSemaine.*values*().length; i++) {

JourSemaine jSem = JourSemaine.*values*()[i];

System.***out***.println(jSem.**ordinal**() + " - "

+ jSem.**name**() + "\t"

+ jSem.**toString**() + "\t"

+ **jSem**);

}

Résultat :

0 - LUNDI LUNDI LUNDI

1 - MARDI MARDI MARDI

2 - MERCREDI MERCREDI MERCREDI

3 - JEUDI JEUDI JEUDI

4 - VENDREDI VENDREDI VENDREDI

5 - SAMEDI SAMEDI SAMEDI

**Exemple 2 – Semaine avancée**

**public enum** JourSemaine {  
 ***LUNDI***(**"Lundi"**, **"Lun"**, **false**),  
 ***MARDI***(**"Mardi"**, **"Mar"**, **false**),  
 ***MERCREDI***(**"Mercredi"**, **"Mer"**, **false**),  
 ***JEUDI***(**"Jeudi"**, **"Mer"**, **false**),  
 ***VENDREDI***(**"Vendredi"**, **"Ven"**, **false**),  
 ***SAMEDI***(**"Samedi"**, **"Sam"**, **true**),  
 ***DIMANCHE***(**"Dimanche"**, **"Dim"**, **true**);  
  
 **private final** String **nom**;  
 **private final** String **abreviation**;  
 **private final boolean weekEnd**;  
  
 JourSemaine(String nom, String abreviation, **boolean** weekEnd) {  
 **this**.**nom** = nom;  
 **this**.**abreviation** = abreviation;  
 **this**.**weekEnd** = weekEnd;  
 }  
}

**Exemple 3 - Mois**

**public enum** Mois {  
 ***JANVIER***(**"Janvier"**, **"janv"**, 31),  
 ***FEVRIER***(**"Février"**, **"févr"**, 28),  
 ***MARS***(**"Mars"**, **"mars"**, 31),  
 ***AVRIL***(**"Avril"**, **"avr"**, 30),  
 ***MAI***(**"Mai"**, **"mai"**, 31),  
 ***JUIN***(**"Juin"**, **"juin"**, 30),  
 ***JUILLET***(**"Juillet"**, **"juil"**, 31),  
 ***AOUT***(**"Août"**, **"août"**, 31),  
 ***SEPTEMBRE***(**"Septembre"**, **"sept"**, 30),  
 ***OCTOBRE***(**"Octobre"**, **"oct"**, 31),  
 ***NOVEMBRE***(**"Novembre"**, **"nov"**, 30),  
 ***DECEMBRE***(**"Décembre"**, **"déc"**, 31);  
  
 **private final** String **nom**;  
 **private final** String **abreviation**;  
 **private final int nbJour**;  
  
 Mois(String nom, String abreviation, **int** nbJour) {  
 **this**.**nom** = nom;  
 **this**.**abreviation** = abreviation;  
 **this**.**nbJour** = nbJour;  
 }  
  
 **public static void** main(String[] args) {  
  
 }  
}

**Exemple 4 - Planete**

**public enum** Planete {  
 ***MERCURE***(3.303e+23, 2.4397e6),  
 ***VENUS***(4.869e+24, 6.0518e6),  
 ***TERRE***(5.976e+24, 6.37814e6),  
 ***MARS***(6.421e+23, 3.3972e6),  
 ***JUPITER***(1.9e+27, 7.1492e7),  
 ***SATURNE***(5.688e+26, 6.0268e7),  
 ***URANUS***(8.686e+25, 2.5559e7),  
 ***NEPTUNE***(1.024e+26, 2.4746e7);  
  
 **private final double masse**; *// en Kg* **private final double rayon**; *// en mètres* Planete(**double** masse, **double** rayon) {  
 **this**.**masse** = masse;  
 **this**.**rayon** = rayon;  
 }  
  
 **private double** getMasse() {  
 **return masse**;  
 }  
  
 **private double** getRayon() {  
 **return rayon**;  
 }  
  
 *// constante gravitationnelle universelle (m3 kg-1 s-2)* **public static final double *G*** = 6.67300E-11;  
  
 **double** calculerLaGravite() {  
 **return *G*** \* **masse** / (**rayon** \* **rayon**);  
 }  
  
 **double** calculerPoidsDeSurface(**double** autreMasse) {  
 **return** autreMasse \* calculerLaGravite();  
 }  
  
 **public static void** main(String[] args) {  
 **double** poidsSurLaTerre = 70; *//En kg* **double** masse = poidsSurLaTerre / ***TERRE***.calculerLaGravite();  
  
 *//Planete p1 = new Planete(1,1); // un type enum n'est pas instanciable* System.***out***.println(**"Il y a "** + Planete.*values*().**length** + **" planètes enregistrées."**);  
 **for** (Planete p : Planete.*values*())  
 System.***out***.printf(**"Sur la planète %s votre poids est de %f%n kg."**,  
 p, p.calculerPoidsDeSurface(masse));  
 }  
}

# Exemples de code Java

Jeu d’échec

# JUnit - Validation et correction de programmes

## Tests unitaires

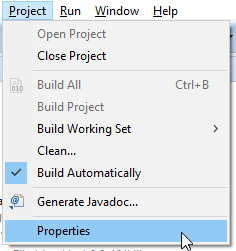
* Ils sont destinés à tester une unité du logiciel.
* On vise que ces tests se fassent de façon automatique avec le minimum de manipulation de la part du testeur.
* On voudra prévoir un jeu de test qui couvre le plus de cas possible en particulier (idéalement tous) ceux qui testent les limites de la classe.

## JUnit

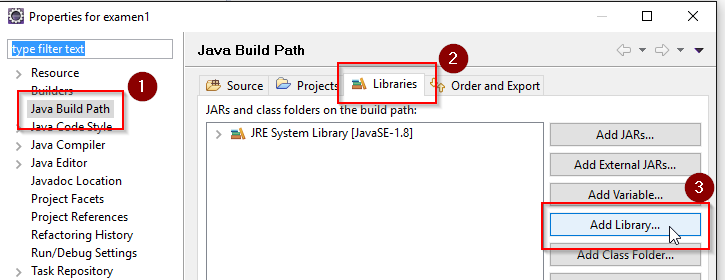
* Les tests unitaires servent à tester chacune des méthodes d’une classe pour s’assurer de leur bon fonctionnement.
* Pour faciliter la tâche et pour automatiser les tests unitaires, il existe des outils. L’un de ces outils est un ensemble de classes (framework) développé pour écrire des tests unitaires
* En Java, notre outil se nomme JUnit et est intégré à l’ensemble des environnements de développements (Eclipse, JDeveloper, NetBeans, etc.).

## Ajouter à un projet Eclipse la librairie JUnit

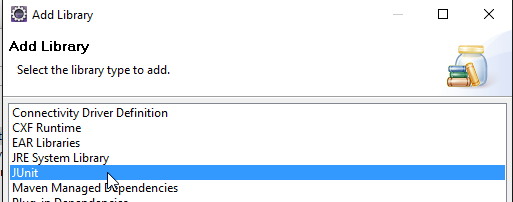
1. « Menu » /  « Project » / « Properties »  :



1. « Java Build Path » / « Librairie » / « Add Library… » :



1. « JUnit » / « Next > » / « Finish » / « Ok » :



# Références

* Langage Java
  + <http://www.ukonline.be/cours/java/apprendre-java>
* Wikipédia
  + <https://fr.wikipedia.org/wiki/Langage_machine>
  + <https://fr.wikipedia.org/wiki/Programme_informatique>

1. L’équivalent Eclipse est **Source/Format** ou **ctrl+Shift+F** [↑](#footnote-ref-1)