FORMATION

Java Fondamentaux

01/06/2023

m2iformation.fr

grammation

Les fondamentaux

- **Les éléments fondamentaux :**
  - Les variables
  - Les structures conditionnelles
  - Les structures répétitives
  - Les méthodes (fonctions)
  - Les tableaux (ou array)
  - Les classes
  - L'héritage
  - Les interfaces
  - Les énumérations
  - Les expressions lambda
  - Les collections
  - Les exceptions
  - Les streams
  - JDBC

troduction

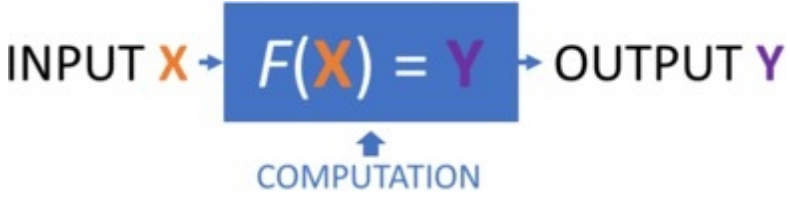
Introduction

Développer ne consiste pas à écrire des formules mathématiques (sauf si vous travailler sur des logiciels destinées aux sciences évidemment). On parle plutôt **d'algorithme**.

Même si les langages sont différents, l'objectif principal d'un langage de programmation reste celui d'instruire la machine pour qu'elle produise des outputs conformes aux objectifs de l'application.

On résume souvent par F(X) = Y, où :

- X représente l'input
- Y représente l'ouput
- F() représente la fonction qui permet de transformer X en Y.



Introduction

- La programmation permet de résoudre un problème de manière automatisée grâce à l'application d'un algorithme :

  Programme = Algorithme + Données

- Un algorithme est une suite d'instructions qui sont évaluées par le processeur sur lequel tourne le programme.
- Les instructions utilisées dans le programme représente le **code source**.
- Pour que le programme puisse être exécuté, il faut utiliser un langage que la machine peut comprendre : un **langage de programmation**.

Introduction

- **Langages procéduraux / langages objets**
  - Il existe des langages procéduraux tel que le langage C.
  - Il existe des langages fonctionnant à base d'objets tel quel le langage Java.

Documentation officielle de Java :
https://docs.oracle.com/en/java/javase/index.html

Introduction

- **Langage bas niveau vs langage de haut niveau**

  - Un langage de bas niveau est un langage qui est considéré comme plus proche du langage machine (binaire) plutôt que du langage humain. Il est en général plus difficile à apprendre et à utiliser mais offre plus de possibilité d'interactions avec le hardware de la machine.

  - Un langage de haut niveau est le contraire, il se rapproche plus du langage humain et est par conséquent plus facile à appréhender. Cependant les interactions se voient limitées aux fonctionnalités que le langage met à disposition.

  - Java est un langage de haut nveau.

Introduction

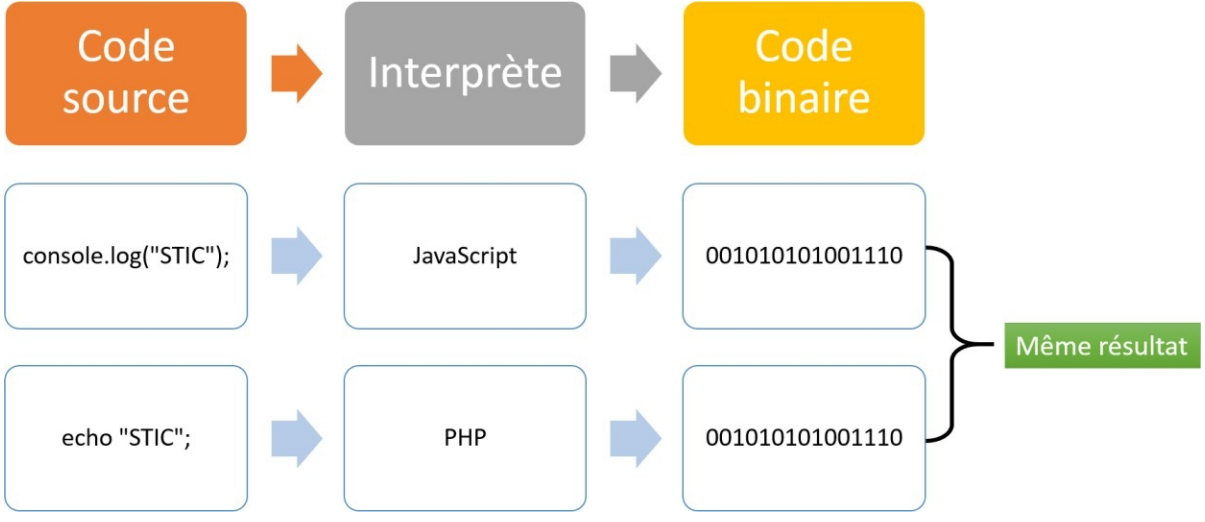
- **Compilation vs Interprétation**

  - La compilation d'un programme consiste à transformer toutes les instructions en langage machine avant que le programme puisse être exécuté. Par conséquent il sera nécessaire de refaire la compilation après chaque modification du code source.

  - Si un langage n'est pas compilé, il est nécessaire d'utiliser un interprète qui traduit les instructions en temps réel (on run time). Dans ce cas le code source est lu à chaque exécution et par conséquent les changement apportés au code seront pris en compte directement. La contrainte réside dans le fait que le machine faisant tourner le programme doit disposer de l'interprète de celui-ci.

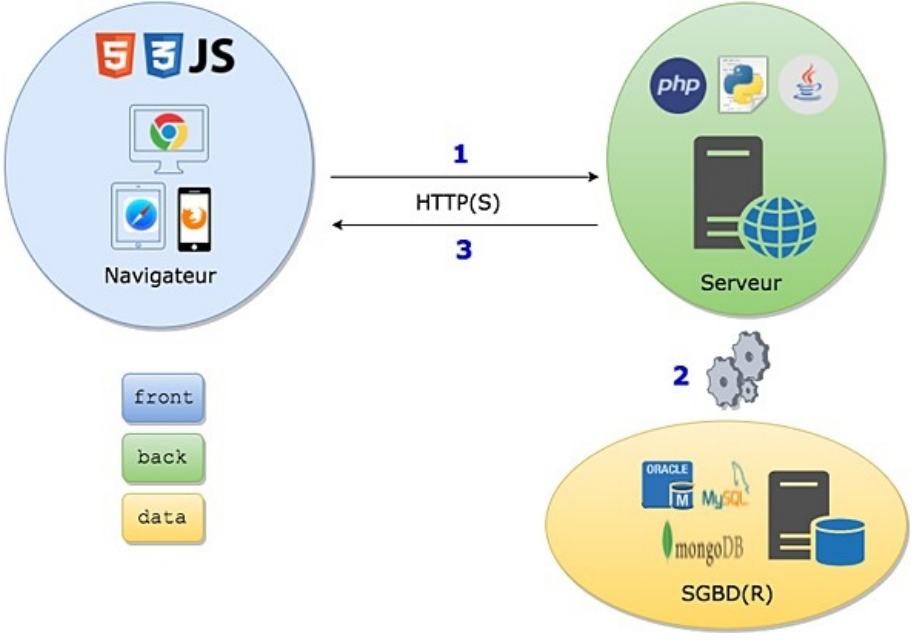
Java est un langage interpreté.

Introduction



Introduction

- Anatomie d'une application interactive



Introduction

- **Composition de Java**

Le langage Java

- Ecriture des programmes Java
- Orienté objet
- Compilation du code Java en byte code (langage machine portable en fichier .javac)

La Machine Virtuelle Java

- Interprète et exécute le byte code
- La machine virtuelle Java (JVM) est disponible pour chaque système d'exploitation
- Un fichier en byte code peut être exécuté sur n'importe quelle JVM indépendamment de l'OS

La plateforme Java

- Ensemble de classes prédéfinies
- API (Application Programming Interface) disponible pour les développeurs

Introduction

- **Gestion de la mémoire**
- L'interpréteur Java (JVM) sait quels emplacements mémoires il a alloué.
- Il sait déterminer quand un objet alloué n'est plus référencé par un autre objet ou une variable.
- Ramasse-miette (« Garbage Collector ») : détecte et détruit ces objets non référencés (libération automatique de la mémoire).

Introduction

- **Java est un langage objet**
    - Tout est classe et objet !
    - En Java, tout ce qui est produit est sous forme de classes.
    - Les fonctionnalités de base de Java sont disponible sous forme de classes.
    - C'est au développeur d'identifier ce qui doit être créé par rapport au problème à résoudre !

L'identification des classes est issue d'un processus d'analyse.
Il est possible d'utiliser une méthodologie pour identifier les classes.
Par exemple : UML (Unified Modeling language).

Introduction

- **Java est un langage objet**

Exemple :

- Un garage entretient des voitures
- Un garage est référencé par sont N° de Siret, nom, adresse, propriétaire, chiffre d'affaire et nombre de salariés
- Les voitures possèdent des caractéristiques comme la marque, un numéro de série
- Les voitures possèdent toutes un moteur et un châssis
- Le châssis a notamment un numéro de série
- Le moteur a notamment une référence et une puissance
- Les voitures roulent, démarrent, freinent

Introduction

- **Java est un langage objet**

Exemple :

Le diagramme de classes UML correspondant aurait :

- Une classe Garage, avec ses propriétés (adresse, etc...)
- Une classe Voiture, avec ses propriétés (marque, etc...)
- Une classe Chassis, avec ses propriétés (poids , etc...)
- Une classe Moteur, avec ses propriétés (puissance , etc...)
- Une relation entre Garage et Voiture (entretien)
- Une relation d'agrégation entre Voiture et Chassis et Moteur, Voiture étant l'agrégat (composée)

Introduction

- **Java est un langage objet**

Exemple :

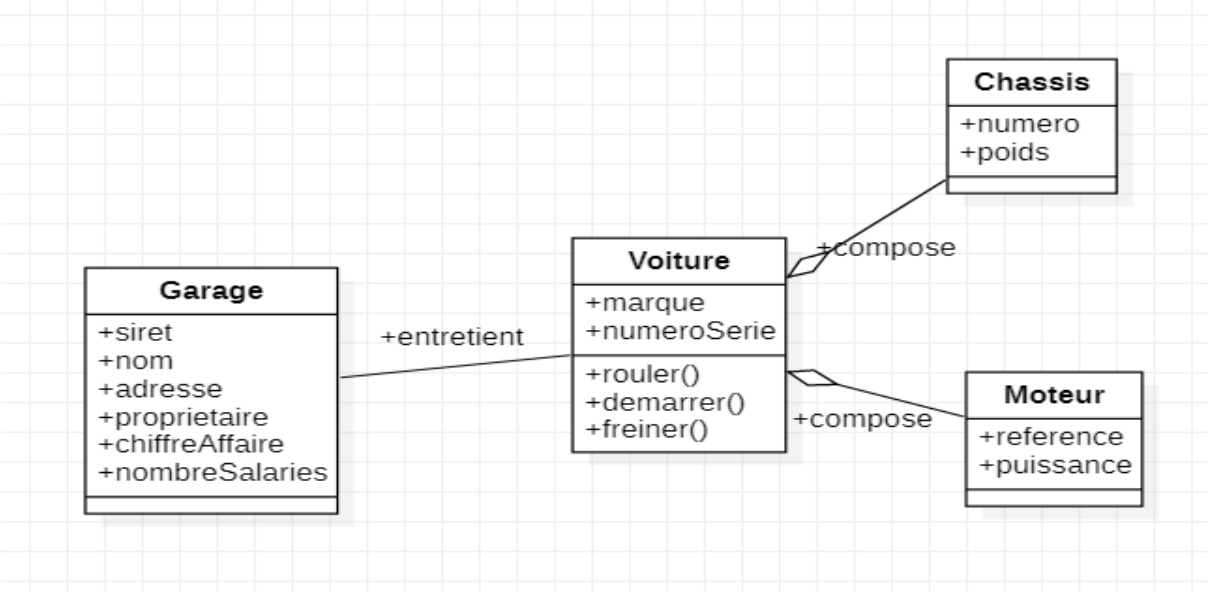
Faire le diagramme de classe avec StartUML : https://staruml.io/download

- Classe Garage avec ses attributs
- Classe Voiture, avec ses attributs et méthodes
- Classe Chassis, avec ses attributs
- Classe Moteur, avec ses attributs
- Relation entre Garage et Voiture (entretien)
- Relation d'agrégation entre Voiture et Chassis et entre Voiture et Moteur (Voiture comporte un châssis et un moteur)

Introduction

- **Java est un langage objet**

Exemple :




stallations

Installation

Pour programmer en langage Java nous avons besoin d'installer une JVM (Java Virtual Machine), un JDK (Java Development Kit) ainsi qu'un IDE (Integrated Development Environment). Il existe plusieurs IDE tel que Eclipse, Visual Studio Code, Intellij, Apache Netbeans.

Nous installerons le JDK version LTS d'Oracle.
Si vous voulez utiliser une version maintenue dans la durée, utilisez la LTS (Long Term Support).

Pour l'IDE nous installerons Apache Netbeans parce qu'il est gratuit et support les dernières versions du JDK de Java.

Installation

- **Installation de la JVM**

Lien de téléchargement : https://www.java.com/fr/download/manual.jsp

Exécuter le fichier téléchargé, puis laisser les options par défaut et laisser faire jusqu'au bout.

Installation

- **Installation du JDK version LTS d'Oracle**

Lien de téléchargement : https://www.oracle.com/java/technologies/downloads

Exécuter le fichier téléchargé, puis laisser les options par défaut et laisser faire jusqu'au bout. Vous pouvez éventuellement changer le dossier d'installation du JDK. Pensez a bien repérer l'emplacement de l'installation du JDK.

Remarque : il est possible d'installer plusieurs versions de JDK sur une même machine.

Installation

- **Installation d'Apache Netbeans**

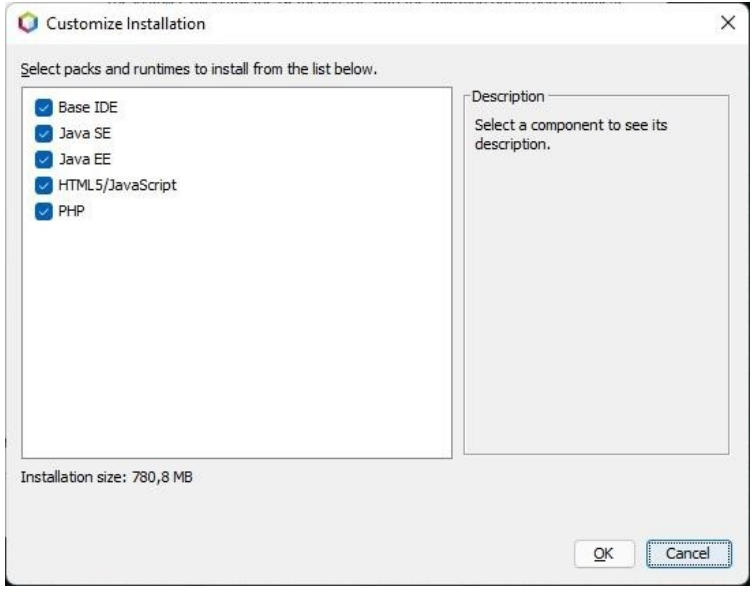
Télécharger et exécuter la dernière version du logiciel, puis suivre les étapes suivantes. Si une erreur intervient, indiquant qu'il ne trouve pas l'emplacement du JDK, il faut lancer l'installation via une fenêtre d'invite de commande en mode administrateur et saisir :

nomFichier.exe --javahome "emplacementJava"

Installation

- **Installation d'Apache Netbeans**

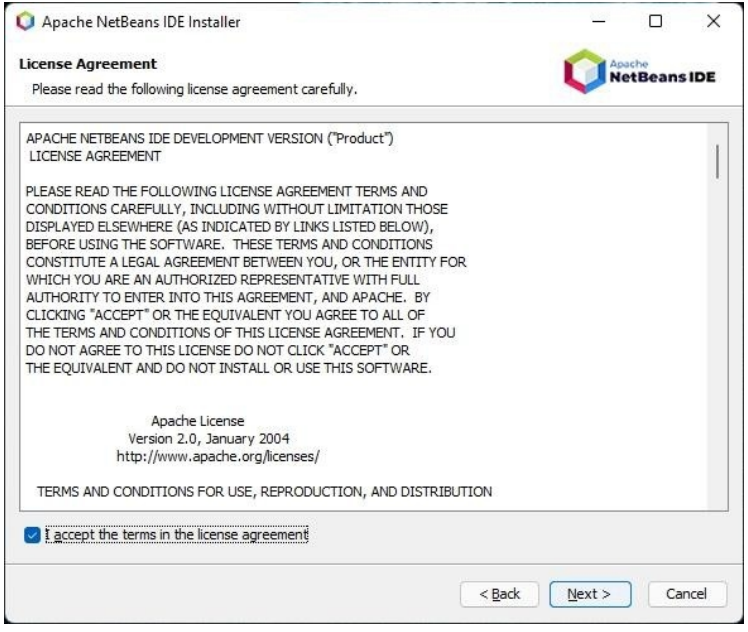
Cliquer sur le bouton « Cutomize », vérifier que tout est coché et valider.



Installation

- **Installation d'Apache Netbeans**

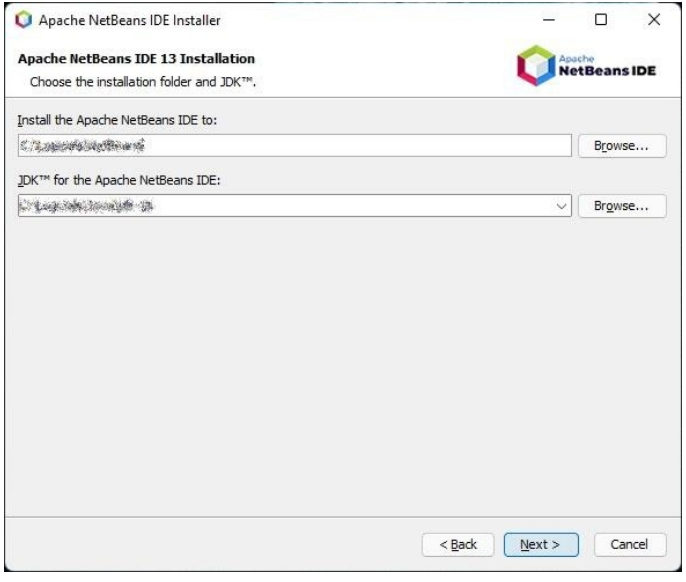
Valider les conditions d'utilisation puis cliquer sur le bouton « Next ».



Installation

- **Installation d'Apache Netbeans**

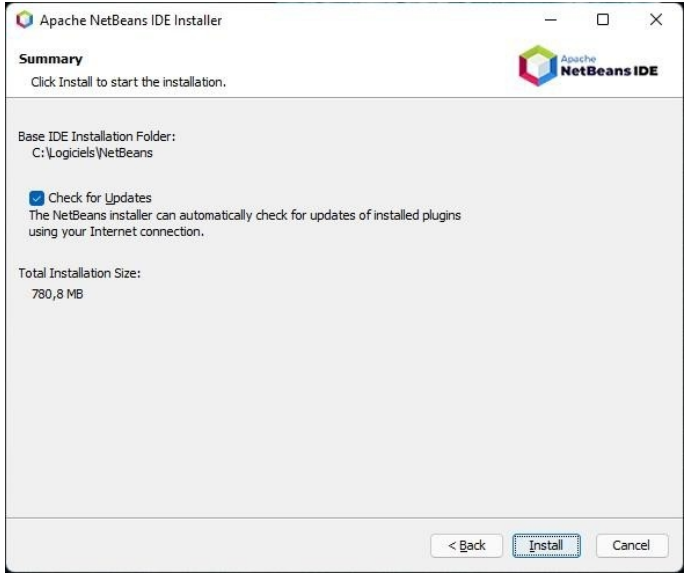
Choisir l'emplacement de l'installation de Netbeans et indiquer l'emplacement du JDK à utiliser par défaut au démarrage de Netbeans.



Installation

- **Installation d'Apache Netbeans**

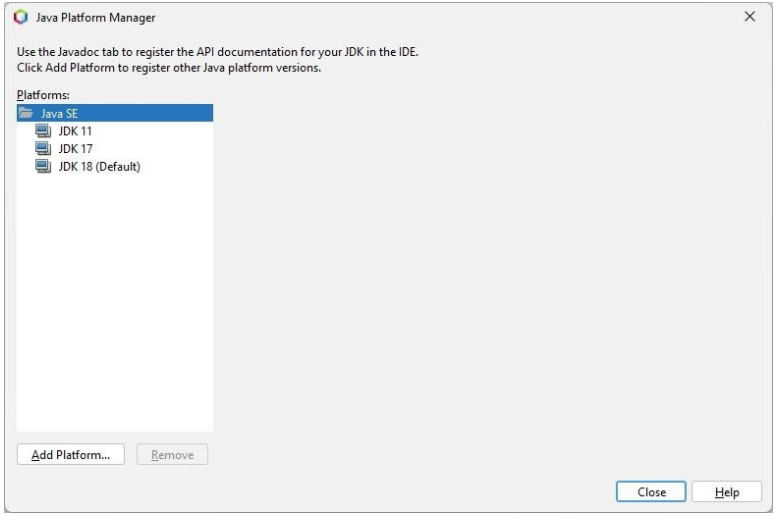
Cliquer sur le bouton « Install » afin de démarrer l'installation de Netbeans.
Puis « Finish » une fois celle-ci terminée.



Installation

- **Paramétrage d'Apache Netbeans**

Cliquer sur le menu « Tools » puis « Java Platforms »

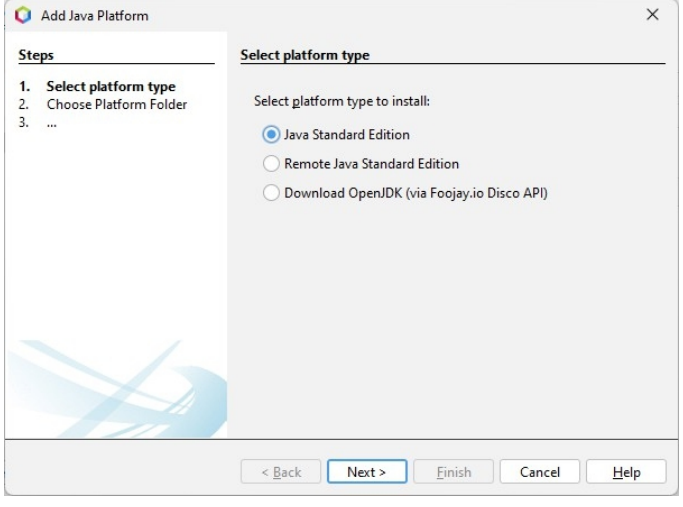


Vérifier les versions du JDK qui apparaissent. Pour ajouter un JDK, cliquer sur « Add Platform » et choisir l'emplacement du JDK à ajouter ou à télécharger.

Installation

- **Paramétrage d'Apache Netbeans**

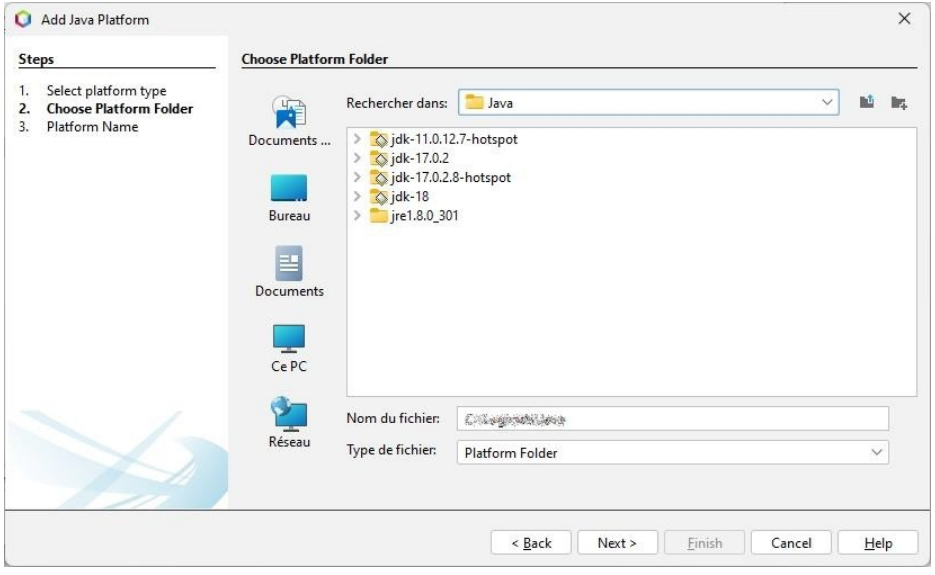
Laisser sur « Java Standard Edition » sélectionnée pour ajouter un JDK déjà installé sur votre machine et cliquer sur le bouton « Next » (Nous utiliserons cette option). Choisir « Download OpenJDK » pour installer un JDK parmi une liste fournie par le logiciel.



Installation

- **Paramétrage d'Apache Netbeans**

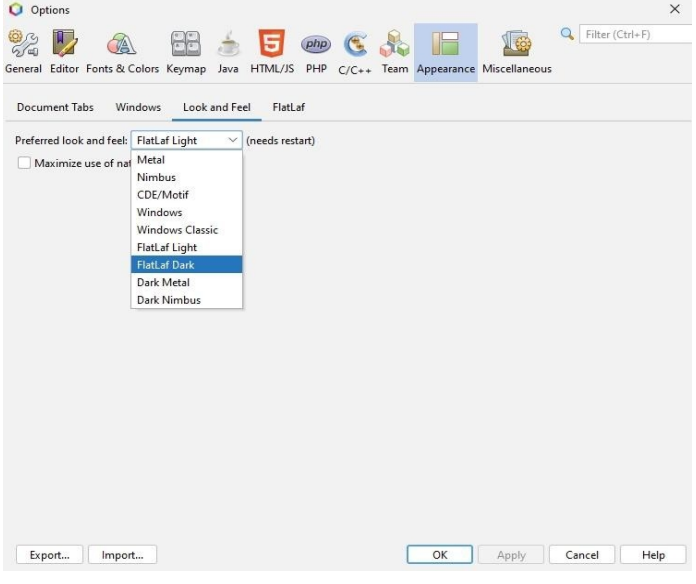
Indiquer l'emplacement du JDK déjà installé et cliquer sur « Next » puis « Finish » une fois l'ajout terminé.



Installation

- **Paramétrage d'Apache Netbeans**

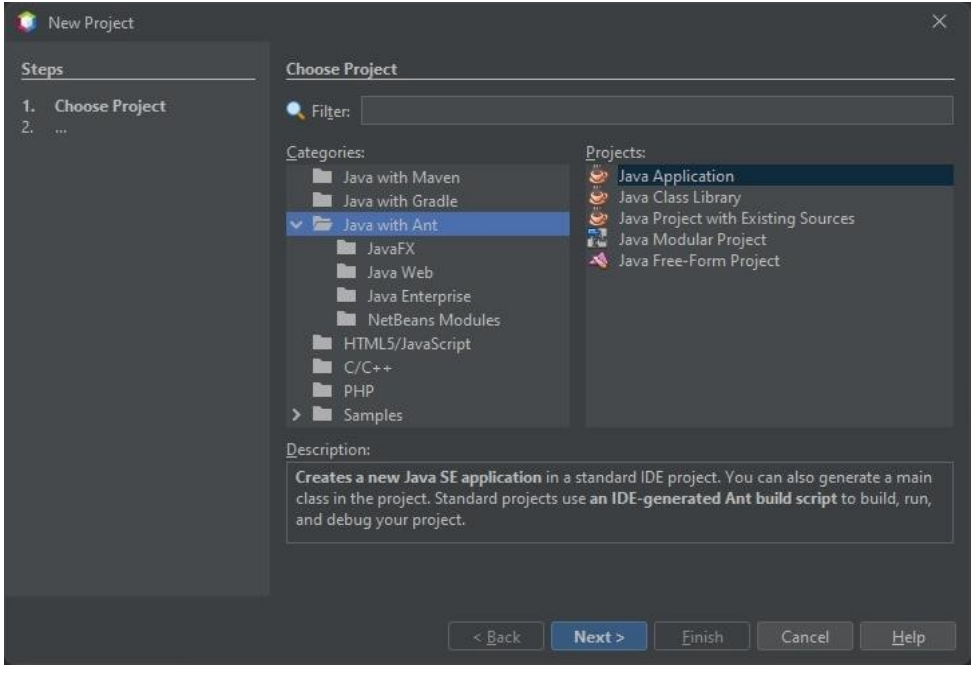
Pour ceux qui préfèrent un mode sombre pour programmer, cliquer sur « Tools » puis Options. Cliquer sur « Appearance » et « Look and Feel » puis choisir dans la liste le dark mode souhaité et valider avec « Apply » (cliquer sur le lien donné par le logiciel quand celui-ci demandera de redémarrer).



Installation

- **Utilisation d'Apache Netbeans**

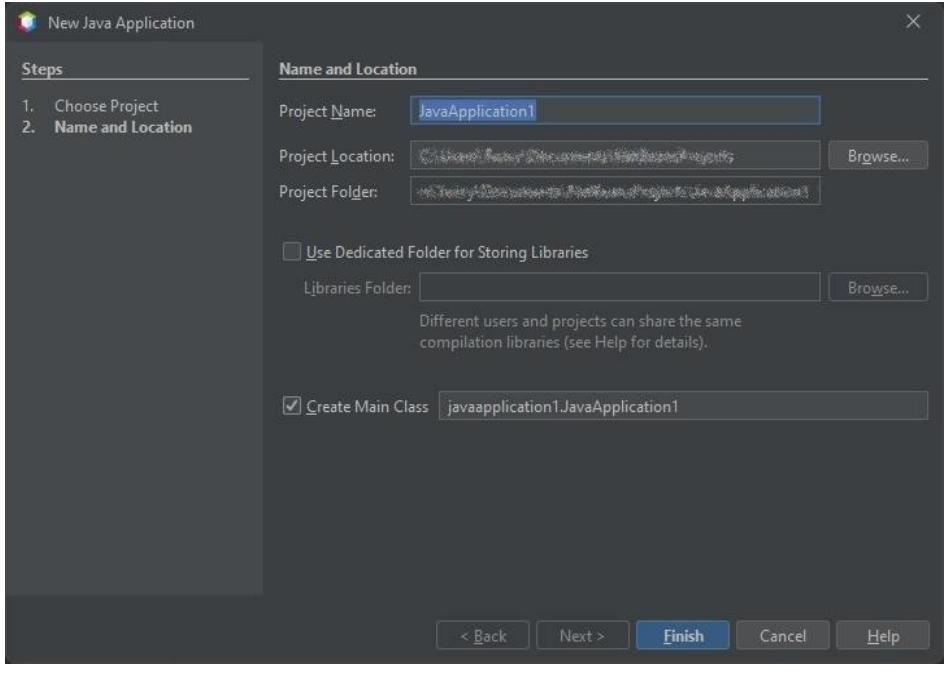
Pour commencer un projet, choisir le menu « File » puis « New Project », choisir « Java with Ant » et « Java Application ».



Installation

- **Utilisation d'Apache Netbeans**

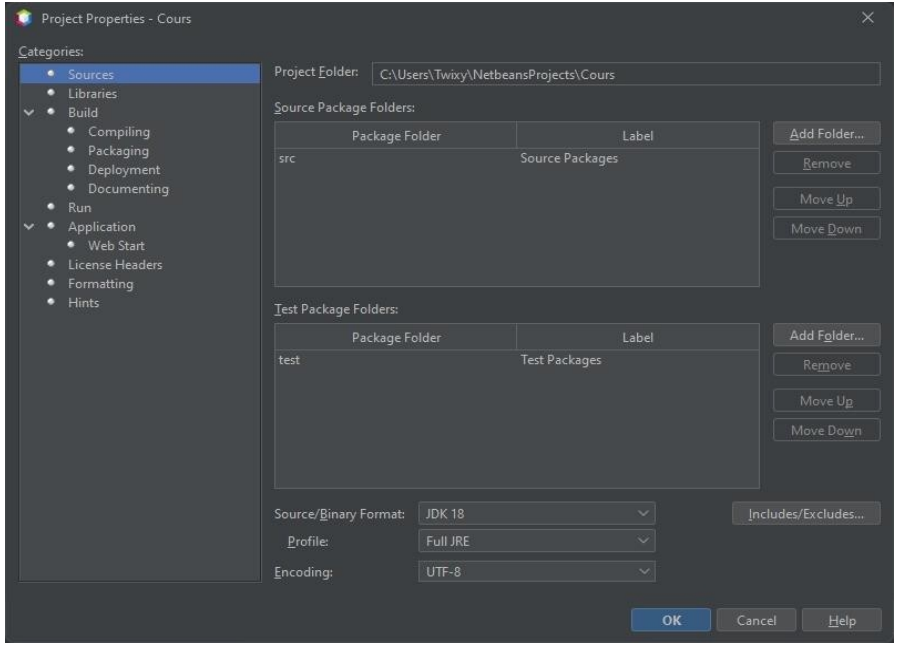
Choisir un nom de projet, un emplacement pour le projet et pour le dossier contenant le projet et cliquer sur le bouton « Finish ».



Installation

- **Changement de JDK dans un projet Apache Netbeans**

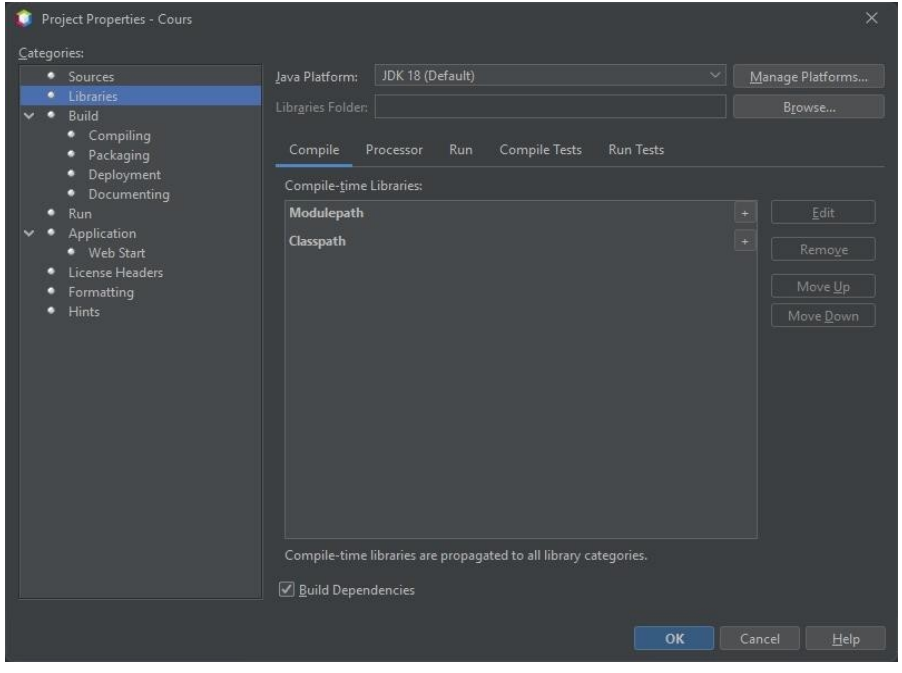
Cliquer sur « File » puis « Project Properties » (ou clic droit sur le nom du projet puis « Properties »). Dans la partie « Sources », vous pouvez modifier l'option « source/format binaire » (version JDK) que Netbeans va utiliser.



Installation

- **Changement de JDK dans un projet Apache Netbeans**

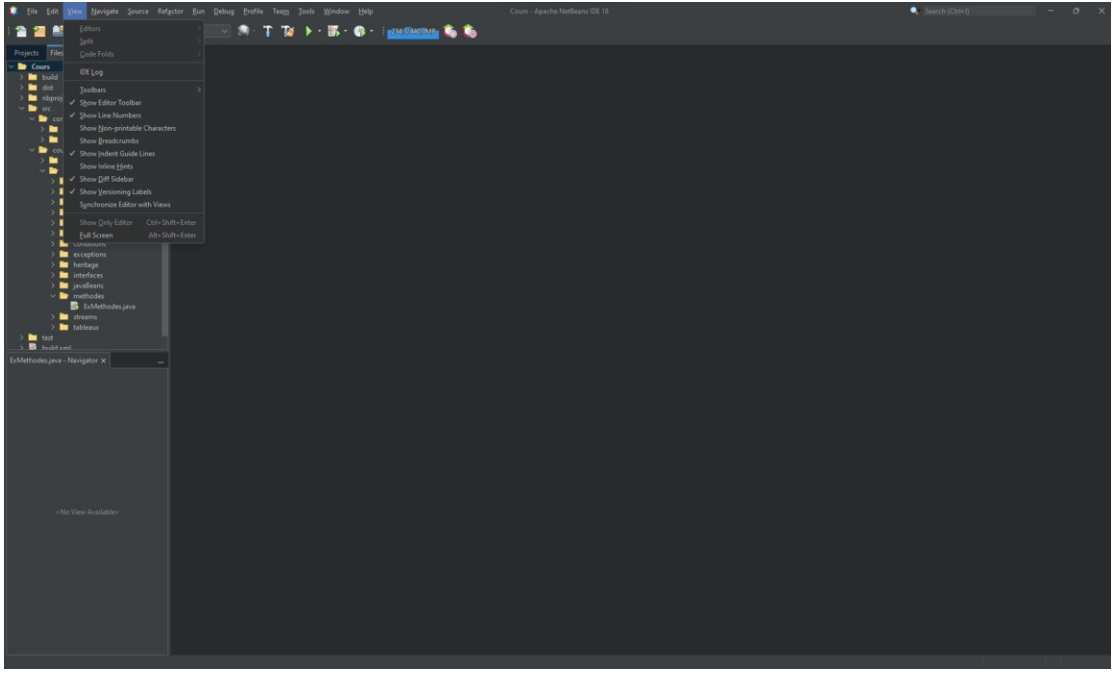
Dans la partie « Libraries » de la même fenêtre, modifier la « Java plate-forme » que le projet utilisera pour construire les classes.



Installation

- **Suppression hints**

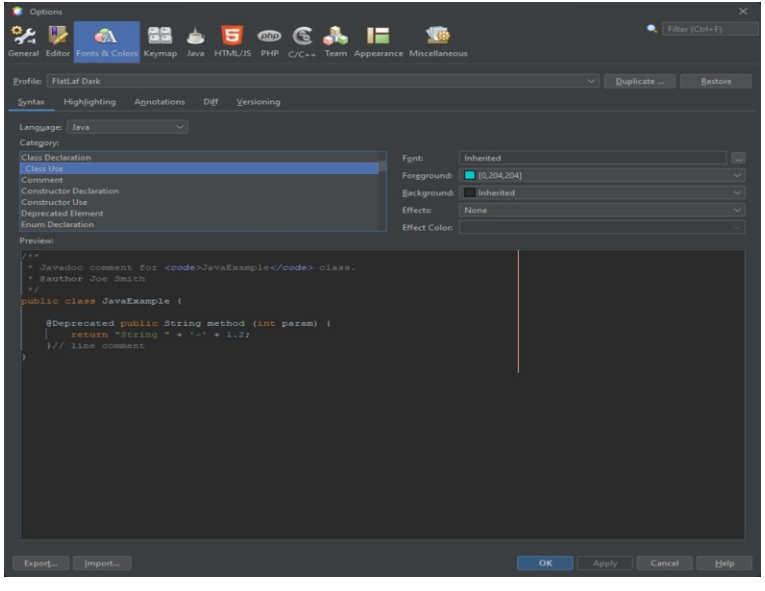
Aller dans le menu « View » et décocher l'option « show inline hints ».

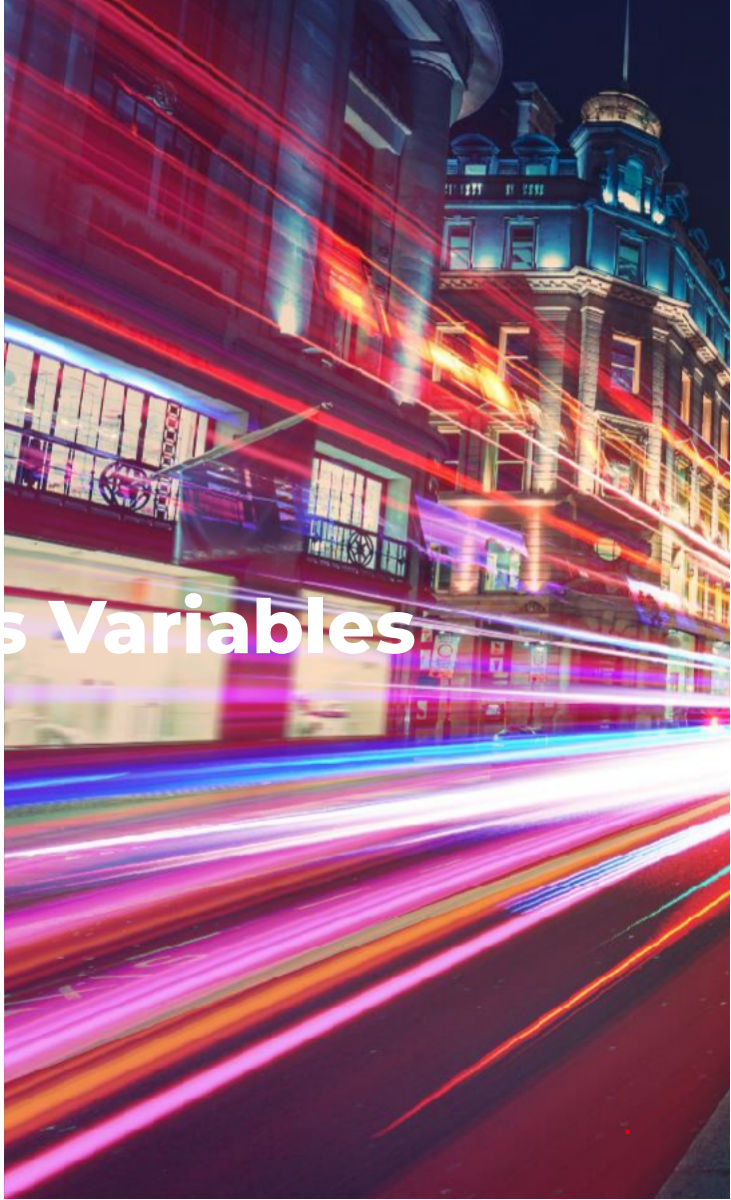


Installation

- **Changer la couleur des classes**

Aller dans le menu « Tools » puis « Option ».
Aller dans « Fonts & Colors », choisir le langage Java.
Choisir « Class use » dans la listebox « Category » situé à gauche, et choisir une couleur pour le « Foreground » situé à droite.





s Variables

Les variables

Une variable permet d'identifier une valeur avec un nom. Elle peut s'écrire avec les caractères de l'alphabet en minuscule, majuscule, les chiffres, et les caractères « _ » et « $ ». Elle ne peut pas commencer par un chiffre, ni avoir de caractères accentués ou spéciaux. Par convention, on écrit les variables en camel case.

Remarque : par convention, les constantes s'écrient en majuscule uniquement.

Utilité des variables :

- La déclaration d'une variable sert à créer une référence dans un emplacement en mémoire.
- L'affectation d'une variable sert à lui associer une valeur.

Remarque : Les types de base ne sont pas des objets, ils n'ont aucune méthode. Pour chaque type de base, il existe une classe correspondante riche en fonctionnalités.

Les variables

- Les types de variables :

  - Les **chaînes de caractères** (string) : elles sont utilisées pour représenter du texte. Une chaîne commence et finie par un guillemet double. Depuis le JDK 15, il est possible d'écrire des String multi lignes. Cela s'appel un « TextBlock ». le TextBlock commence et fini par des triples guillemets double.

  - Les **chiffres** (nombre entier, à virgule flottante, etc.) : ils sont utilisés surtout avec des opérateurs mathématiques.

  - Les **valeurs booléennes** (boolean) : elles sont des valeurs dichotomiques (soit vrai, soit faux).

Les variables

- Les types de variables :

  - Les **tableaux** (array) : ils sont utilisés pour créer des listes avec des indices qui permettent de récupérer la valeur associée.

  - Les **objets** : ils sont des conteneurs qui peuvent inclure souvent tout type de données, y compris de sous-objets, des variables (propriétés / attributs), ou des fonctions (méthodes).

Les variables

- Les types de base :

Type	Taille	Description	Intervalle
char	2 octets	Une unité de code, suffisant à représenter un grand nombre de point de code, et même un caractère Unicode (UTF-16) 'b' '\u250C'	'\u0000' à '\uFFFF'
byte	1 octet	Un nombre entier de 8 bits (soit un octet) signé	-128 à 127
short	2 octets	Un nombre entier de 16 bits signé	-32 768 à 32 767
int	4 octets	Un nombre entier de 32 bits signé	−2 147 483 648 et +2 147 483 647
long	8 octets	Un nombre entier de 64 bits signé	−9 223 372 036 854 775 808 et +9 223 372 036 854 775 807

Les variables

- Les types de base :

Type	Taille	Description	Intervalle
float	4 octets	Un nombre à virgule flottante de 32 bits signé (simple précision) 0.0F	de 2^{-149} (Float.MIN_VALUE) à $2^{128} - 2^{104}$ (Float.MAX_VALUE) $-\infty$ (Float.NEGATIVE_INFINITY) $+\infty$ (Float.POSITIVE_INFINITY) pas un nombre (Float.NaN)
double	8 octets	Un nombre à virgule flottante de 64 bits signé (double précision) 0.0D	de 2^{-1074} (Double.MIN_VALUE) à $2^{1024} - 2^{971}$ (Double.MAX_VALUE) $-\infty$ (Double.NEGATIVE_INFINITY) $+\infty$ (Double.POSITIVE_INFINITY) pas un nombre (Double.NaN)
boolean	1 octet	Une valeur logique	false (faux) ou true (vrai)

Les variables

- **« Boxing » et « Unboxing » :**

Boxing : Conversion automatique du type primitif en son équivalent

objet. Exemple :

```
int valeurPrimitive = 100;
Integer valeurObjet = valeurPrimitive;
```

Unboxing : Conversion automatique de l'objet en son type primitif.

Exemple :

```
Integer valeurNumerique = 100;
int valeur = valeurNumerique;
```

Les variables

- **Conversions de type :**

Il est possible de convertir un type de variable en un autre.

Exemple de conversion de type entre chaîne de caractères et valeur numérique :

```
        double monDouble = 3.14159;
String pi = Double.toString(monDouble);

String valeurNumerique = "100";
Integer valeur = Integer.valueOf(valeurNumerique);
```

Remarque : il est possible de forcer la conversion temporaire d'une variable, on appel cela le casting.

Syntaxe :

```
float valFloat = (float)valInt;
```

Les variables

- **Les opérateurs :**

Les opérateurs permettent de manipuler ou comparer des valeurs, notamment des variables. Parmi les opérateurs utilisés fréquemment dans tous les langages de programmation vous trouverez :

- L'opérateur d'affecation
- Les opérateurs arithmétiques
- Les opérateurs binaire (comparaison)
- Les opérateurs logiques
- L'opérateur de négation

En Java il existe d'autres opérateurs très utilisés :

- Les opérateurs unitaires
- Les opérateurs avec assignation

Les variables

- **Opérateur d'affectation :**

L'affectation d'une variable avec le signe « = »
Syntaxe :
« type » variable = valeur;

Remarque : Pour que l'affectation soit correcte il faut que les 2 opérandes soient de même type.

Exemple :
Integer variable1 = 100; Integer variable2 = variable1;

Affectation multiple :
« type » var1, var2, …, varN = … = var2 = var1 = valeur;

Les variables

- **Opérateurs arithméthiques :**

Les opérateurs arithméthiques de base sont : +, -, *, / et % (modulo = reste de la division).
Remarque : l'opérateur "+" permet aussi de concatener des chaînes de caractères.

- **Opérateurs unitaires (++ et --) :**

Les opérateurs unitaires permettent de raccourcir les expressions d'incrémentation et de décrémentation ainsi que faire des post ou pré affectations selon la position de l'opérateur unitaire par rapport à la variable concernée.

Les variables

- **Opérateurs binaires (comparaison) :**

Egalité	→	==
Différent	→	!=
Inférieur	→	<
Supérieur	→	>
Inférieur ou égale	→	<=
Supérieur ou égale	→	>=

- **Opérateur logique :**

&&	→	ET logique.
||	→	OU logique.

Les variables

- **Opérateur de négation :**

! → Négation du résultat d'une évaluation.

- **Opérateurs avec assignation :**

Cela permet de raccourcir une expression lorsque la variable ce trouve des 2 cotés de l'expression. Cela fonctionne avec tout les opérateurs de base.

Exemple :

i = i + 2; → i += 2;

4. Les structures conditionnelles

Les structures conditionnelles

- **Les conditions :**

Les structures de contrôle permettent d'exécuter seulement certaines instructions d'un programme selon la vérification d'une ou plusieurs conditions. Il existe 3 structures conditionnelles. Ces structures sont imbriquables entre elles et les autres types de structures.

  - if / else
  - switch / case
  - (? :)

Les structures conditionnelles

- **if / else :**

Cette structure exécute un bloc d'instructions si la condition au niveau du if est vérifiée. Le else est optionnel et permet d'exécuter un autre bloc d'instructions si la condition du if correspondant est fausse. Toute évaluation qui retourne un booléen peut être utilisée avec le if / else.

Syntaxe d'un if / else :

```
if (variable == valeur) {
      instructions si la condition est vraie;
}
else {
      instrcutions si la condition est fausse;
}
```

Les structures conditionnelles

- **switch / case :**

Le switch apporte de la clarté au code par rapport au « if / else » dans le cas où un traitement différent doit être appliqué selon les différentes valeurs d'une variable. Plutôt que d'imbriquer des « if / else » on préférera utilisée le switch / case. Cette structure peut également être utilisé pour tester des chaînes de caractères.

Important : Il faut stipuler la sortie du bloc switch dès que l'on a exécuter le traitement voulu. Pour cela nous utilisons l'instruction break.
Si aucun cas ne correspond à la valeur testée, alors c'est celui par défaut (default) qui est exécuté.

Les structures conditionnelles

- **switch / case :**

Syntaxe d'un switch / case :

```
switch (variable) {
    case "A":
            instructions;
            ...
            break;
    case "B":
            instructions;
            ...
            break;
    ...
    default:
            instructions;
            ...
            break;
    }
```

Les structures conditionnelles

- **switch / case :**

A partir du JDK 14, la structure switch a changé et n'a plus besoin de break. Un case peut accepter plusieurs valeurs et le « : » devient « -> ». De plus, s'il y a plusieurs instructions pour un « case », ces instructions devront être placées entre accolades comme n'importe quel autre bloc d'instructions.

Les structures conditionnelles

- **switch / case :**

Syntaxe :

```
switch (valeur) {
    case "A", "B", …          ->          instruction;
            case "C", …       ->          {
                              instructions;
                              …;
                                    }
      …
      default                 ->          instruction;
}
```

Les structures conditionnelles

- **switch / case :**

Ce nouveau switch / case permet aussi d'être utilisé sous forme d'expression. Cela permet par exemple, d'initialiser une variable avec une valeur qui dépend de plusieurs conditions. S'il faut plusieurs lignes on utilisera les accolades pour délimiter le bloc d'instructions et, pour retourner la valeur on doit employer « yield » (comme un return).

Remarque : dans ce cas d'utilisation attention au ; à la fin du switch.

Les structures conditionnelles

- **switch / case :**

Syntaxe :

```
<type> variable = switch (valeur) {
    case "A", "B", …    ->     valeur_de_retour;
    case "C", …         ->     {
                                    instructions;
                                    …;
                                    yield valeur_de_retour;
                               }
    …
    default             ->     instruction;
};
```

Les structures conditionnelles

- `( ? : ) :`

Cette structure conditionnelle est appelée ternaire. Elle permet de remplacer une structure conditionnelle if / else dont le résultat peut être affecté directement à une variable. Le type de variable doit correspondre avec la valeur retournée.

Syntaxe du ternaire :

« type » resultat = (a==b) ? « valeur retournée si vrai » : « valeur retournée si faux »;


répétitives

Les structures répétitives

- **Les boucles :**

Ces structures sont appelées boucles. Elles sont la base d'un concept très utile en programmation : l'**itération.** Cela permet d'exécuter plusieurs fois des instructions.

Il existe 4 structures répétitives. Ces structures sont imbriquables entre elles et les autres structures.

- for
- do ... while
- while
- for (:)

Les structures répétitives

- **for :**

Ces structures existe dans beaucoup de langages. Elle fonctionne ainsi :

1 - L'initialisation est exécutée, une seul fois.
2 - Le test est évalué, et s'il est faux on quitte la boucle.
3 - Le bloc de commandes du for est exécuté.
4 - L'incrémentation est effectuée.
5 - Le test est évalué. Si c'est faux on quitte la boucle, sinon on revient à l'étape 3.

Syntaxe d'une boucle for :

```
for (initialisation variable; test de sortie; incrémentation) {
      traitements a faire en boucle si test est vrai;
}
```

Les structures répétitives

- **do ... while :**

La particularité de cette boucle, c'est que le bloc d'instruction est exécuté une fois minimum. Le while situé à la fin de la boucle permet d'évalué la condition de sortie de la boucle.

Structure d'un do … while : Initialisation valeurTest; do {

```
    instructions;
    incrémentation de la valeurTest;
}
while (valeurTest < valeur);
```

Les structures répétitives

- **while :**

La particularité de cette boucle, c'est qu'elle peut ne pas être exécutée. Le while permet d'évaluer la condition de sortie de la boucle.

Structure d'un while :

Initialisation de valeurTest;

```
while (valeurTest > valeur) {
        instructions;
        décrémentation de valeurTest;
}
```

Les structures répétitives

- **for(:) :**

Elle est appelée boucle for « intelligente ». La particularité de cette boucle, c'est qu'elle s'utilise avec des tableaux ou des collections uniquement, et qu'il n'y a pas à gérer la sortie de la boucle. A chaque itération, l'élément du tableau / collection est chargé dans la variable. La progression est automatique jusqu'à la fin du tableau / collection.

Structure d'un for (:) :

for (<type> variable : tableau ou collection) { instructions;
}

Les structures répétitives

- **Les instructions break et continue :**

continue : permet d'arrêter les instructions du bloc de la boucle et de recommencer la boucle avec la valeur suivante.

break : permet de sortir définitivement de la boucle en cours.

Ces deux instructions sont utilisable dans tous les types de boucles.


odes

Les méthodes

Les méthodes sont des fonctions situées dans une classe. Or Java ne fonctionne qu'avec des classes, donc avec des méthodes. Elles représentent une sorte de programme dans le programme, car elles sont la première forme d'organisation du code. On utilise des méthodes pour regrouper des instructions et les appeler sur demande : chaque fois qu'on a besoin de ces instructions, il suffira d'appeler la méthode au lieu de répéter toutes les instructions. Pour accomplir ce rôle, le cycle de vie d'une méthode se divise en deux phases :

- Une phase unique dans laquelle la méthode est définie. On définit à ce stade toutes les instructions qui doivent être groupées pour obtenir le résultat souhaité.

- Une phase d'appel de cette méthode qui peut être répéter autant de fois que désiré. On demande à la méthode de mener à bien toutes les instructions dont elle se compose à un moment donnée dans la logique de notre programme.

Les méthodes

Une méthode peut optionnellement retourner une valeur.

Si elle ne retourne pas de valeur, elle doit être du type « void ».
Si elle retourne une valeur, la dernière instruction dans la méthode devra être « return » suivi de la variable ou valeur à retourner qui doit correspondre au type de la méthode.

Elle peut aussi avoir des paramètres (variables) de façon optionnelle.
S'il y a des paramètres alors l'appel de la méthode doit contenir le même nombre et type de paramètres.

Exemple de méthode Java très utilisée :
System.out.println(x); → x représente ce qu'il faut afficher dans la console Java.

Les méthodes

Syntaxe de la définition d'une méthode :

```
« type » nomMethode(« type » variable1, …) {
        instructions;
        …
        return valeurOuVariable;
}
```

Syntaxe de l'appel d'une méthode :

```
resultat = nomMethode(variable1, …);
```

Remarque : resultat doit être du même type que la méthode.


s tableaux

Les tableaux

Un tableau de base est une suite ordonnée ou non d'élément du même type. Le type du tableaux peut être ce qu'on veut, même une classe personnalisée. Il est possible de les passer en paramètre d'une méthode.
La particularité d'un tableau, c'est qu'il faut le déclarer avant utilisation, et son allocation mémoire est faite à ce moment là. Il est possible de déclarer et alimenter un tableau sans préciser sa taille au moment de son initialisation.
Pour récupérer la taille du tableau on utilisera la propriété length.
Le nombre d'éléments chargés dans un tableaux peut être différent de la taille de celui-ci.
On peut accéder à un élément d'un tableau via son index. Le premier élément d'un tableau à toujours l'index 0. L'accès à un index du tableau pour lequel il n'y a pas d'objet chargé retourne la valeur null.
Une tentative d'accès en dehors des bornes du tableau lève une exception (erreur). Il est possible de créer des tableaux à plusieurs dimensions. Ce n'est pas forcément très pratique et lisible selon la situation.

Les tableaux

Syntaxe d'un tableau :

Visibilité type[] nom = new type[nbrElements]; // 1 dimension;
Visibilité type[][] nom = new type[nbrElements1][nbrElements2]; // 2 dimensions
Ou
Visibilité type nom[] = new type[nbrElements]; // 1 dimension;
Visibilité type nom[][] = new type[nbrElements1][nbrElements2]; // 2 dimensions

Pour initialiser directement un tableau avec des éléments, on peut faire :

Visibilité type[] nom = {elem1, elem2, elem3, …};
Ou
Visibilité type nom[] = {elem1, elem2, elem3, …};

Les classes

Les classes

Java met en œuvre les concepts suivants :

- Les classes
- Les objets
- L'encapsulation
- Les JavaBeans
- L'héritage
- L'abstraction
- Le polymorphisme

Les classes

Une classe est le modèle (un moule) qui va recevoir essentiellement les attributs (variables) et méthodes (fonctions) de l'objet. On peut créer autant d'objet qu'on veut à partir d'une classe. Ils seront indépendant.

Dans Java il est possible de définir plusieurs classes dans un même fichiers suffixé par .java mais, parmi ces classes, une seule sera définie comme « public » et c'est elle qui donnera le nom au fichier. Une classe a un nom commençant par une majuscule.

Client.java

```
1  package gestion;
2  public class Client {
3      private String nomclient;
4      private Integer statut;
5      public Boolean creerClient(String nomclient, Integer statut) {
6          this.nomclient=nomclient;
7          this.statut=statut;
8          return true;
9      }
10     public Boolean changerStatut(Integer statut) {
11         if (this.statut==0) {
12             return false;
13         }
14         this.statut=statut;
15         return true;
16     }
17 }
18 class Adresse{
19     public String adresse;
20 }
21
```

Les classes

- **La visibilité :**

On trouve 3 mots-clés pour la visibilité des attributs et méthodes mais 4 visibilités différentes.

- **public** : La portée est totale, même en dehors du package.
- **private** : Opposé de public, inaccessible depuis l'extérieur d'une classe.
- **protected** : Accès depuis toutes les classes du même package ou des classes qui héritent de celle-ci.
- **« friendly »** : En cas d'absence de mot clé, l'accès est par défaut depuis toutes les classes du même package (Il est appelé friendly mais ne s'écrit pas).

Une classe ne peut avoir que la visibilité public ou « friendly »

```
2  public class Client {
3      private String nomclient;
4      private Integer statut;
```

```
23         Client monclient = new Client();
24         monclient.nomclient="Sarl";
```

Erreur de compilation !!!

Les classes

- **Les constructeurs :**

Le constructeur est une méthode particulière permettant de créer un objet. En cas de passage de paramètres dans le constructeur, il permet aussi de l'initialiser avec ceux-ci.

Un constructeur est une méthode spéciale qui porte obligatoirement le nom de la classe dans laquelle il est défini.

Il n'a pas type de retour.
Il n'est pas obligatoire, mais très utile.

Les classes

- **Les constructeurs :**

Il est appelé automatiquement lorsqu'on demande la création d'un objet de la classe de ce constructeur (utilisation du mot clé « new »).

Il peut y avoir plusieurs constructeurs dans une classe. Ils se différencient par le nombre et type de paramètres passés. Comme une surcharge de méthode classique.

Il peut être appelé depuis un autre constructeur de façon explicite avec le mot-clé « this ». Comme avec des méthodes surchargées classique.

Les classes

- **Le mot-clé this :**

Le mot-clé « this » se réfère à l'instance de l'objet en cours.

```
2  public class Client {
3      private String nomclient;
4      private Integer statut;
5      private Adresse adresse;
6      public Boolean creerClient(String nomclient, Integer statut, String adresse) {
7          this.nomclient=nomclient;
8          this.statut=statut;
9          this.adresse = new Adresse();
10         this.adresse.setAdresse(adresse);
11         return true;
12     }
13     public Boolean changerStatut(Integer statut) {
14         if (this.statut==0) {
15             return false;
16         }
17         this.statut=statut;
18         return true;
19     }
20     public String getAdresse() {
21         return this.adresse.getAdresse();
22     }
23 }
24 class Adresse{
25     private String adresse;
26     public String getAdresse() {
27         return adresse;
28     }
29     public void setAdresse(String adresse) {
30         this.adresse = adresse;
31     }
```

Déclaration de la propriété de type Adresse

Stockage de l'adresse dans l'objet de type Adresse, nommé adresse

Accesseur, ou getter, permettant l'accès au contenu de l'adresse via la propriété de type Adresse

Propriété adresse de la classe Adresse, avec ses deux accesseurs.

Les classes

- **L'encapsulation :**

L'encapsulation en Java est un terme qui recouvre une chose simple : Protéger les données d'un objet. Cela permet de créer un couplage faible au lieu d'un couplage fort entre objets et classes. Le couplage faible est recommandé car il permet une meilleur maintenabilité du code. Une classe peut contenir des données de type public et private. Si on déclare toutes les propriétés en public, alors l'encapsulation est inutile mais nous auront un couplage fort.

```
1  package nonencapsule;
2  public class Client{
3      public void crediterCompte(double credit) {
4          CompteBancaire cb = new CompteBancaire();
5          cb.crediter(100.12);
6          cb.valeur=200;
7      }
8  }
9  class CompteBancaire {
10     public double valeur;
11     public double crediter(double credit) {
12         this.valeur+=credit;
13         return this.valeur;
14     }
15 }
16
```

Accessible directement !

Non protégée !

Les classes

- **L'encapsulation :**

Les accesseurs (ou Getters et Setters en anglais) sont des méthodes qui permettent de ne pas accéder directement à la variable de la classe qui elle, est privée. C'est donc un couplage faible.

```
1  package nonencapsule;
2  public class Client{
3      public void crediterCompte(double credit) {
4          CompteBancaire cb = new CompteBancaire();
5          cb.crediter(100.12);
6      }
7  }
8  class CompteBancaire {
9      private double valeur;
10     public double crediter(double credit) {
11         this.valeur+=credit;
12         return this.valeur;
13     }
14     public double getValeur() {
15         return valeur;
16     }
17 /*  public void setValeur(double valeur) {
18         this.valeur = valeur;
19     }*/
20 }
21
```

Protégée !

Accesseur pour accéder à la valeur
Encapsulation

Les classes

- **JavaBean :**

C’est une représentation unique d’une entité fonctionnelle utilisable à divers endroits du programme.

Exemple :

- Un client
- Une commande
- Un article

Acronymes similaires pour dire JavaBean :

- POJO (Plain Old Java Object)
- DTO (Data Transfer Object)
- DAO (Data Access Object)

Les classes

- **JavaBean :**

Un JavaBean est une classe qui doit respecter les conventions suivantes :

- Implémenter l'interface Serializable.
- Proposer un constructeur sans paramètre.
- Disposer d'accesseur publiques pour les attributs privés (getters et setters).
- Classe non déclarée « final ».

Les classes

- **JavaBean :**

Exemple d'utilisation :

- Classe métier ArticleService : réalisera des opérations sur un article (création, modification, suppression, …).
- Les méthodes de cette classe travaillerons sur le JavaBean Article.

Les classes

- **Surcharge :**

La surcharge en Java c'est la capacité d'une classe à accepter d'avoir des méthodes avec le même nom. Il est possible d'utilisé le mot clé this pour appeler une méthode surchargée dans une autre.

Contrainte : différenciation sur le nombre et/ou la nature des types qui sont déclarés pour cette méthode.

Attention : le type de retour ne peut servir de discriminant !

Exemple de surcharge dans les méthodes du JDK :

La méthode valueOf de la classe String

Les classes

- **Le mot clé static**

static lie un attribut ou une méthode à la classe. Cela à pour utilité de pouvoir définir des propriétés au niveau global de la classe. Donc ces attributs ou méthodes sont des objets uniques. Modifier un attribut « static » d'une classe, modifie cet attribut pour tout les objets instanciés de cette classe. Par convention, on y accède via le nom de la classe au lieu du nom de l'objet.

- **Le mot clé final**

Ce mot clé s'utilise sur un attribut, une méthode ou une classe. Après initialisation sur un attribut celle-ci ne peut plus être changée ! Cela provoquerait une erreur de compilation. En résumé, ce mot clé permet de déclarer une constante.

Les classes

- **Les mots clés static et final :**

Remarque : associé « static » et « final » est possible. Cela permet de créer une constante unique à la classe.

Le mot clé final appliqué sur une méthode, permet l'interdiction de la redéfinition de cette méthode dans le cas de l'héritage.

Le mot clé final appliqué sur une classe, permet l'interdiction de l'héritage depuis celle-ci.

Sera vu plus tard ;)

L’héritage

L’héritage

C’est une notion indissociable du langage Java et de la Programmation Orienté Objet (POO). La question à se poser quand on parle d’héritage est :

Est-ce que ma classe est une sorte d’autre classe plus générique ?

Exemple : Un Smartphone est une sorte de téléphone.

Cela induit donc une notion de parenté. La classe la plus générique sera appelée classe mère et la classe qui en héritera sera appelée classe fille. On ne peut hériter que d’une seule classe, l’héritage multiple est interdit en Java. Les types génériques en Java héritent tous d’une classe nommée Object. Seuls les types de bases n’héritent pas de la classe Object. Par conséquent, la création d’une classe qui n’hérite de rien explicitement, hérite quand même de la classe Object. L’instruction
« extends Object » n’est donc pas nécessaire car implicite.

L’héritage

Syntaxe de l’héritage :

```
class Smartphone extends Telephone {
        …
}
```

L’héritage

Avec l’héritage, on peut redéfinir les méthodes héritées. L’annotation « @Override » n’est pas obligatoire mais c’est une bonne pratique pour indiquer :

- Aux développeurs qu’il s’agit d’une méthode redéfinie.
- Au compilateur qu’il s’agit d’une méthode redéfinie pour qu’il vérifie son exactitude.

Le mot clé super : permet d’accéder aux attributs et méthodes de la classe mère si la visibilité de celle-ci le permet.

Le mot clé this : permet de référencer les attributs et méthodes de l’objet en cours et de l’objet dont il hérite si la visibilité de celui-ci le permet.

L’héritage

Il est possible de bloquer l’héritage en Java via le mot clé final sur la classe. En effet, cela provoque un blocage complet. Aucune classe ne peut hériter d’une classe définie avec final. On peut aussi limiter le blocage au niveau d’une méthode ou même simplement d’un attribut (devient une constante).

Si une méthode est définie dans la classe mère, et qu’elle n’est pas redéfinie dans la classe fille, alors cette méthode est accessible depuis un objet de la classe fille si la visibilité le permet.

L’héritage

- **L’abstraction :**

Dans une classe, il est possible de déclarer des méthodes sans code associé.
→ parce qu’il n’est pas possible de fournir un code pour ce niveau d’abstraction.
→ parce qu’on veut mettre en œuvre le polymorphisme.

Exemple :
une classe Voiture avec une méthode rouler()
une classe Camion avec une méthode rouler()
une classe Vehicule avec une méthode abstraite rouler(), c’est à dire sans code car on ne sait pas dire comment le véhicule roule. C’est trop abstrait à ce niveau.

Remarque : si on déclare une méthode abstraite, alors la classe aussi doit être abstraite.

L’héritage

- **L’abstraction :**

Syntaxe de classe abstraite :

```
abstract class NomClasse {
        …
}
```

L'héritage

- **Le polymorphisme :**

Le polymorphisme dans java veut simplement dire qu'une classe peut prendre plusieurs formes et c'est d'autant plus vrai avec les classes qui hérite d'une classe supérieure.

Exemple :

```
25 public class Garage{
26     public static void main(String[] args) {
27         Vehicule v1 = new Voiture(true);
28         Vehicule v2 = new Camion(false);
29         v1.rouler();
30         v2.rouler();
31     }
32 }
```

Création de 2 objets de type Vehicule :
1 compatible avec Voiture et 1 compatible avec Camion

Appel de la méthode rouler() possible car rouler() a été définie dans Vehicule

A l'exécution, c'est la méthode définie sur l'objet réel qui est appelée ! ➔ **polymorphisme**


interfaces

Les interfaces

Une interface peut représenter un point d'échange entre un fournisseur et un ou plusieurs clients (contrat).

L'interface comporte uniquement les déclarations des méthodes présentes (il n'y a pas de code interne aux méthodes) et ne peut pas comporter des variables. Par contre, on pourra définir des constantes ou des énumérations pour créer des constantes dans l'interface.

La classe qui implémente l'interface, définit les méthodes de celle-ci.
Par convention, on nomme l'interface avec un « I » en préfixe.

L'utilisation d'une interface permet de créer une dépendance faible entre l'interface et la classe qui l'implémente. Une classe qui appelle une autre classe, crée une dépendance forte à cette classe.

Les interfaces

Syntaxe d'une interface :

```
interface INomInterface {
        « type » nomMethode1(paramètres);
        « type » nomMethode2();
        …
}
```

Les interfaces

Exemple avec métaphore :

Plusieurs fabricants décident de proposer une façade de machine à laver standard. L'objectif est de retrouver les mêmes fonctionnalités au même endroit sur n'importe qu'elle machine à laver.
Les fabricants définissent l'interface de la façade, qu'il peuvent donner à un sous-traitant pour la fabrication (plan, modèle).
Le sous-traitant va savoir quoi faire pour construire les façades, sans connaître les fonctionnalités propre à chaque fabriquant (démarrage, arrêt, programmes, ouverture, …)
Chaque fabricant pourra utiliser la façade pour construire ses modèles de machine à laver.
→ La façade est l'interface (contrat).
→ La machine à laver propre à un fabricant est une implémentation de cette interface.

Les interfaces

Une interface qui n'a qu'une seule méthode abstraite est appelée une interface fonctionnelle. Java fournit une annotation @FunctionalInterface, qui est utilisée pour déclarer une interface comme interface fonctionnelle.

En Java, la classe qui implémentera une interface devra utiliser la syntaxe suivante :

```
class MaClasse implements IMonInterface {
        …
}
```

Les interfaces

Une classe qui implémente une interface doit implémenter toutes les méthodes de celle-ci. Sinon, il y a une erreur de compilation. Pour ne pas implémenter toutes les méthodes, il faut déclarer la classe abstraite.

Une même classe peut hériter d'une autre classe et implémenter une ou plusieurs interfaces.
Mais une classe ne peut pas hériter de plusieurs classes. L'héritage multiple n'est pas autorisé en Java.

Remarque : une interface peut héritée d'une autre interface en utilisant la même syntaxe que celle utilisée pour les classes.

Les interfaces

Depuis la version 8 de Java, il est possible d'ajouter des méthodes statiques dans une interface et des méthodes « default ».

Les méthodes « default » et « static » ont pour particularité d'être définie dans l'interface et pourront être utilisées directement (ou redéfinie) par la classe qui implémente l'interface.

Lors d'une création d'une interface dérivée de l'interface contenant une méthode « default » on peut :

- Ne rien mentionner : l'interface hérite de la méthode par défaut.
- Redéfinir la méthode
- Redéclarer la méthode : la méthode devient abstraite.

Les interfaces

Les méthodes « static » sont définies dans l'interface comme les méthodes « default » mais elles ne peuvent pas être redéclarées lors de la création d'une interface dérivée. Par conséquent, elles ne peuvent pas être abstraites. En effet, il n'est pas possible en Java de créer des méthodes statiques abstraites.

Remarque : Les classes abstraites et les interfaces se ressemblent. La principale différence est que les interfaces ne peuvent pas avoir d'état (variables d'instances) ni de constructeur. Elles ne peuvent avoir que des constantes.

umerations

Les enumerations

Une énumération est un ensemble fini de valeurs. La particularité de l'énumération c'est qu'on ne peut utiliser que les valeurs définies au sein de celle-ci. C'est une sorte de liste de constantes, d'où le fait décrire le valeurs en majuscule dans l'énumération.

Une énumération utilise le type « enum » de Java.

Remarque : Pour que le type « enum » soit reconnu il faut utiliser Java 8 minimum.

Les enumerations

Syntaxe d'une énumération :

enum Nom { VAL1, VAL2, …, VALn }

Appels :

```
Nom.VAL1;                 // retourne VAL1.
Nom.VAL1.ordinal();       // retourne l'index de VAL1.
Nom.values()              // retourne un tableau avec toutes les valeurs.
```

Il est également possible d'intégré des attributs, constructeurs et méthodes dans une énumération comme pour les classes. Dans ce cas, les valeurs doivent être indiquées en premier et se terminer par un point-virgule.


ns lamdba

Les expressions lambda

Les expressions lambda sont introduites dans Java 8 et sont utilisées dans la programmation fonctionnelle. Une expression lambda est donc une fonction qui peut être créée sans appartenir à aucune classe. Une expression lambda peut être transmise comme s'il s'agissait d'un objet et exécutée à la demande.
Une expression lambda est traitée comme une fonction, donc le compilateur ne crée pas de fichier « .class ».
L'expression lambda fournit l'implémentation d'une interface fonctionnelle.

Syntaxe :

(liste d'arguments) → { instructions; }

La liste d'arguments accepte 0 à n paramètres.

Les expressions lambda

Le type d'un paramètre est facultatif. Le compilateur peut déduire le type de la valeur du paramètre.

Pas besoin d'utiliser des accolades dans le corps d'une expression si le corps contient une seule instruction.

Le mot clé « return » est facultatif si le corps a une seule expression.


collections

Les collections

Les tableaux Java vu précédemment permettent de stocker des « collections » de données. Il existe en Java une façon plus simple de gérer ces collections. « Java Collections » désigne un ensemble d'interface et de classes permettant de stocker, trier et traiter les données (classes utiles avec de multiples méthodes).

La plupart des collections se trouvent dans le package java.util

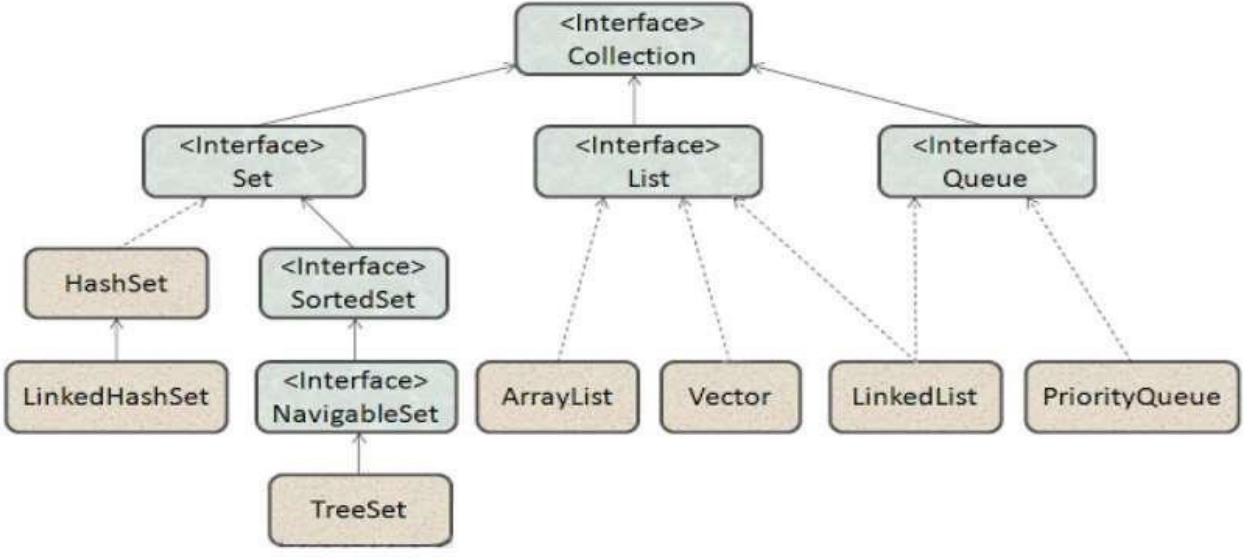
Une collection est un « objet » qui collecte des éléments dans une liste.
Une collection représente un ensemble de données de même type.

La composition des collections de Java est une structure hiérarchique établie à partir de deux grands ensembles.

- Collection (java.util.Collection)
- Map (java.util.Map)

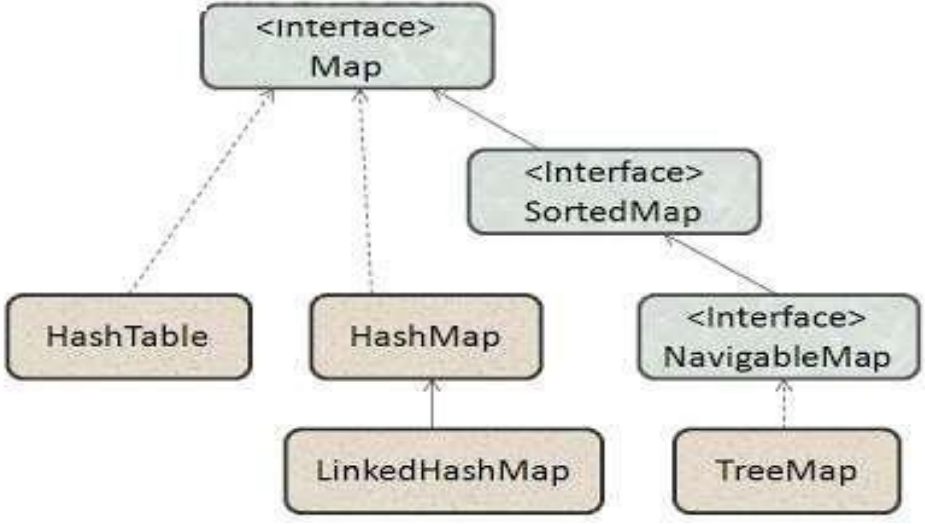
Les collections

- **Java.util.Collection :**



Les collections

- **Java.util.Map :**



Les collections

- **Le type List en Java :**

La classe ArrayList implémente l'interface List, qui elle-même hérite de l'interface Collection. Elle dispose donc, de l'ensemble des méthodes de l'interface List et de l'interface Collection. Une ArrayList peut-être vue et traitée comme une interface List.

On peut écrire :
List listeDeChaines = new ArrayList(); // polymorphisme !

Il n'y a pas encore le type de donnée à stocker de précisé dans cette liste. On le précise en utilisant les « Generics » de Java :
List<String> listeDeChaines = new ArrayList<>();

L'appel d'une méthode d'une liste se fait comme suit :
maListe.nomDeLaMethode(parametres);

Les collections

- **Le type List en Java :**

Liste de méthodes utiles pour gérer les ArrayList :

add(x, val)	→ charge la valeur val dans la liste à la position x. (si x n'est pas préciser, charge la valeur au premier index disponible).
addAll(liste2)	→ charge la liste2 dans une liste.
get(x)	→ accède à l'élément de la liste ayant l'index x.
set(x, val)	→ modifie l'élément à la position x de la liste par la valeur val.
indexOf(val)	→ obtient l'index du premier élément ayant pour valeur val.
lastIndexOf(val)	→ obtient l'index du dernier élément ayant pour valeur val.
contains(val)	→ vérifie si la valeur val est contenu dans la liste (retourne vrai ou faux).
remove(x)	→ supprime de la liste l'élément situé à l'index x.
remove(val)	→ supprime de la liste le premier élément égale à val.
clear()	→ supprime tous les éléments de la liste.
size()	→ retourne la taille de la liste.

Les collections

- **Le type Set en Java :**

Le type Set en Java est une liste ayant pour particularité de contenir que des éléments uniques. Si on converti une liste de type List en Set, les doublons de la List seront supprimés dans le Set. Les collections sont convertible d'un type à l'autre.

Attention : l'ordre d'insertion n'est pas garantie !

Les collections

- **Le type Map en Java :**

L'interface Map permet de stocker un ensemble de paires « clé / valeur » dans une table de hachage. Une Map ne peut pas contenir des éléments dupliqués, chaque clé à une unique valeur. Map est implémenté avec les classes suivantes :

HashMap → ordre d'insertion non garantie.
LinkedHashMap → ordre d'insertion garantie.
TreeMap → stockage des éléments triés selon leur valeur.

L'interface SortedMap hérite de Map et implémente les méthodes pour ordonner les éléments dans l'ordre croissant et décroissant.

Remarque : TreeMap est une implémentation de SortedMap.

Les collections

- **Le type Map en Java :**

Liste de méthodes utiles pour gérer les Map :

entrySet()	→ retourne un objet Set contenant les paires clé / valeurs du Map.
put(cle, val)	→ ajout d'un ensemble clé / valeur.
getKey()	→ obtient la clé de l'entrée en cours.
getValue()	→ obtient la valeur de l'entrée en cours.
replace(cle, val)	→ charge la valeur de l'entrée en cours avec la valeur val.

Les collections

- **Le tri d'une collections**

Le tri d'une collection se fait par la méthode sort().

Exemple :

Tri simple : Collections.sort(liste);
Tri Personnalisé : Collections.sort(liste, methode_implementant_Comparator);

Attention : Si le type d'objet utilisé n'implémente pas l'interface Comparator, ou si on souhaite trier selon un ordre différent, alors il faut fournir une implémentation spécifique de l'interface Comparator !

Les collections

- **L'itération de collection :**

Il existe plusieurs possibilités d'itérer sur une liste en Java.

  - Utilisation de l'interface Iterator
  - Utiliser la boucle « intelligente » for (:)
  - Utiliser les boucles classiques (for (;;), while, do...while)
  - Utiliser la méthode forEach() d'une expression lambda


exceptions

Les exceptions

Les exceptions c'est la gestions des erreurs.
Dans un langage non objet on va traiter les erreurs de cette façon :

```
retour = fonction1(parametre x);
si (retour == erreur) {
      traiter l'erreur;
}
```

En langage objet, non traitons l'erreur via une classe nommée Exception.

- Une exception est un objet instancié de la classe Exception.
- Il faut informer le système de ce qui déclenche la levée d'une exception.
- Il faut savoir récupérer et traiter une exception.

Toutes les exceptions levées dans une méthode et qui n'ont pas été traitées localement, doivent être ajoutées à la signature de la méthode via le mot-clé « throws » qui permet de déléguer la responsabilité des erreurs à la méthode appelante.

Les exceptions

- **try / catch / finally :**

La récupération se fait en encapsulant le code risquant de lever une exception avec les instructions try / catch / finally.

Remarque : le finally est très utile pour libérer des ressources (ex : mémoires ou fichiers).

Les exceptions

- **try / catch / finally :**

Syntaxe d'un try catch :

```
try {
	instructions pouvant provoquer une exception;
}
catch (Exception1 e) {
	instructions si Exception1 est survenue;
}
catch (Exception2 e) {
	instructions si Exception2 est survenue;
}
...
finally {
	instructions à exécuter qu'une exception soit survenue ou non;
}
```

Les exceptions

- **try / catch avec instanceof :**

Une alternative à l'utilisation de plusieurs blocs « catch » consiste à utiliser instanceof.

Les exceptions

- **try / catch avec instanceof :**

Syntaxe avec instanceof :

```
try {
        instructions pouvant provoquer une exception;
}
catch (Exception e) {
        if (e instanceof Exception1) {
                instructions si Exception1 est survenue;
        }
        if (e instanceof Exception2) {
                instructions si Exception2 est survenue;
        }
}
...
finally {
        instructions à exécuter qu'une exception soit survenue ou non;
}
```

Les exceptions

- **Les exceptions avec ressource :**

Le traitement d'exception avec ressource permet de libérer automatiquement la ressource utilisée à la fin du « try / catch ». Ainsi, il n'est pas nécessaire d'avoir un bloc finally pour libérer la ressource. La ressource s'indique à la ligne du try comme une création d'objet classique. Cette objet disparaîtra à la fin du « try / catch ».
Syntaxe d'un try avec ressources :

```
try (<type> obj = new UneClasse(<type> param, ...) {
      instructions pouvant provoquer une exception;
}
catch (Exception1 e) {
      instructions si Exception1 est survenue;
}
…
```

Les exceptions

- **Levée d'exceptions :**

La signature d'une méthode doit inclure la déclaration des diverses exceptions levées dans le code qui ne sont pas traitées localement, et ceci se réalise avec l'instruction « throws ».
Syntaxe d'une levée d'exceptions :

```
<Visibilité> <Type> nomMethode(<Type> param) throws Exception1, Exception2 {
    instructions;
    if (condition 1) {
                    throw new Exception1("Message de l'exception 1");
        }
        if (condition 2) {
                    throw new Exception2("Message de l'exception 2");
        }
    ...
}
```

Les exceptions

- **Les exceptions personnalisées :**

Pour des besoins applicatifs spécifiques, il est possible de créer soit même sa gestion d'exceptions en Java.

Les classes d'exceptions personnalisées doivent hériter de la classe Exception, ou d'une classe qui hérite de cette classe.

La gestion de ces exceptions est identique aux autres.


es streams

Les streams

L'API stream (flux) est introduite dans Java 8, et est utilisée pour traiter des collections d'objet. Un flux est une séquence d'objets qui prend en charge diverse méthodes pouvant être enchaînées pour produire un résultat. Attention, on ne peut parcourir les flux qu'une fois. Si on veut les relire, il faut les recréer.

Les fonctionnalités sont les suivantes :

- Prise en charge d'une collection, d'un tableau ou des entrée / sorties.
- Pas de modification de la structure de données d'origine. Ils fournissent uniquement le résultat selon les méthodes enchaînées.
- Chaque opération intermédiaire est exécutée et renvoie un flux en conséquence. Ces opérations peuvent être enchaînées. Les opérations terminales marque la fin du flux et renvoient le résultat.

Les streams

Quelques opérations intermédiaires :

map() : Renvoie le flux constitué des résultats de l'application d'une fonction à ses éléments.
filter() : Renvoie les éléments d'un flux qui correspondent à un prédicat passé en argument.
sorted() : Renvoie un flux trier.
limit(n) : Limite le nombre d'élément retourner en cas de nombre de valeurs infinies.

Les streams

Quelques opérations terminales :

forEach() : Utilisée pour parcourir chaque élément du flux.
reduce() : Utilisée pour réduire le flux à une seule valeur. La méthode prendre un opérateur binaire en paramètre.
min() : Renvoi le plus petit élément du flux.
max() : Renvoi le plus grand élément du flux.
count() : Renvoi le nombre d'élément contenu dans le stream.

collect() : Renvoi le résultat des opérations intermédiaires effectuées sur le flux.
collect(Collectors.toList()) : Renvoi le flux sous forme de List.
collect(« type »[]::new) : Renvoi le flux sous forme de tableau.

16. JDBC

JDBC

Cela signifie Java Database Connectivity. C'est une API (Application Programming Interface) Java pour connecter et exécuter des requêtes sur une base de données relationnelle (BDDR). Il fait partie de JavaSE (Java Standard Edition). Cette API utilise des pilotes JDBC pour se connecter à une BDDR.

L'API JDBC permet d'accéder aux données stockées dans n'importe qu'elle BDDR. Il est possible d'insérer, mettre à jour, supprimer et lire des données d'une BDDR. C'est comme Open Database Connectivity (ODBC) de Microsoft.

Avant JDBC, l'API ODBC était utilisée, mais celle ci utilise un pilote écrit en C (c'est à dire dépendant de la plateforme et non sécurisé). C'est pourquoi Java a défini sa propre API (JDBC) qui utilise des pilotes JDBC (écrits en langage Java et par conséquent sécurisé et non dépendant de la plateforme).

JDBC

Il existe 5 étapes pour connecter une application Java à une BDDR à l'aide de JDBC. Ces étapes sont les suivantes :

- Enregistrer la classe de pilote
- Créer une connexion
- Créer un objet Statement / PreparedStatement
- Exécuter des requêtes
- Fermer la connexion

Pour pouvoir utiliser JDBC il faut :

Un Système de Gestion de Base de Données Relationnelle (SGBDR), télécharger le pilote correspondant au SGBDR utilisé, puis l'intégrer dans le projet via notre IDE.

JDBC

Pour le SGBDR nous utiliserons Laragon (version complète), c'est un logiciel gratuit, local, intégrant les serveurs : MySQL, PHP, Apache et NodeJS.

Lien de téléchargement : https://laragon.org/download/index.html

Lien de téléchargement du pilote JDBC pour MySQL :

https://dev.mysql.com/downloads/connector/j/

Dézipper et stocker le pilote JDBC pour MySQL (le fichier avec l'extension « .JAR ») dans l'emplacement voulu sur le PC.

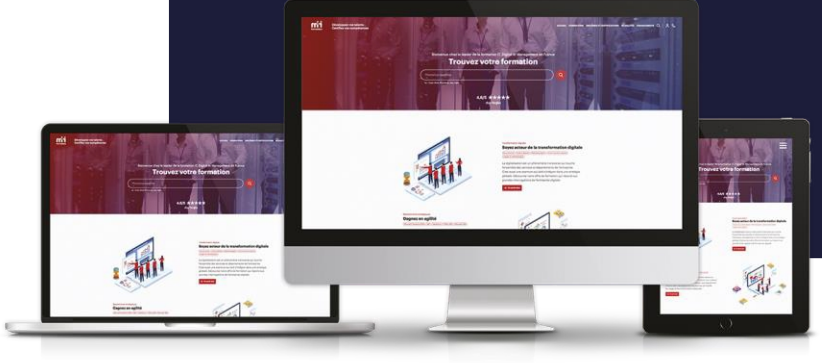
JDBC

Procédure pour intégrer le pilote JDBC MySQL dans notre projet via l'IDE NetBeans :

Cliquer sur l'onglet Projects, dans la fenêtre latérale, faire un clique droit sur librairies, puis choisir « add JAR Folder... ».

Une fenêtre s'ouvre, sélectionner le fichier « .JAR » téléchargé et stocké précédemment et cliquer sur le bouton « Ouvrir ».

FIN



m2iformation.fr