COURS DE JAVA – Introduction
El Hadji Gaye - AlizNet

Avec NetBeans ou Spring Tool Suite.

Auteur	El Hadji Gaye
Pour	Ecole
Date	12/02/2018
Objet	Introduction au Java.

I) Introduction

Java est un langage de programmation orientée objet créé par James Gosling et Patrick Naughton. Ils étaient, à l'époque, des employés de Sun Microsystems. Sun a été racheté, par la suite, en 2009 par la société Oracle. Le principal avantage de java est sa portabilité. En effet, le code source Java peut être exécuté sur plusieurs systèmes d'exploitation (UNIX, Windows, Mac OS ou GNU/Linux). Java dispose d'une machine virtuelle (JVM). Lors de la compilation, le code source java est traduit en un code intermédiaire appelé « byte code ». Ce code sera interprété par la machine virtuelle Java. Il arrive par abus de langage que la JVM soit nommée **JRE** (Java Runtime Environment).
Nous allons utiliser une IDE (Integrated Development Environment) pour notre développement. Il existe plusieurs sortes d'IDE de développement, parmi-elles on peut citer : Netbeans, Spring Tool Suite, Eclipse, JCreator etc.
Dans le cadre de ce cours, nous allons utiliser l'IDE NetBeans.

II) Vocabulaire Java

Lorsqu'on commence la programmation du langage Java, il apparait de nombreux termes :

- JVM : Java Virtual Machine. Cette dernière traduit le code source java en un code intermédiaire appelé «byte code».
- JRE : Java Runtime Environment. Elle est composée de la JVM et de quelques API (Application Programming Interface).
- JDK : Java Development Kit. Il est composé du JRE plus certains outils de base pour le développement (javac : Java Programming Language compiler, javap : Java Class File Disassembler, javadoc, etc.).
- IDE : Integrated Development Environment. C'est l'outil de développement. Il contient la JVM, la JRE, la JDK permettant ainsi de développer des applications java, de les compiler et de les exécuter.
- API : Signifie Application Programming Interface. Ce qui veut dire que c'est un ensemble de bibliothèques et librairies dédiées pour implémenter une fonctionnalité donnée.

III) Installation de NetBeans 8.0.2.

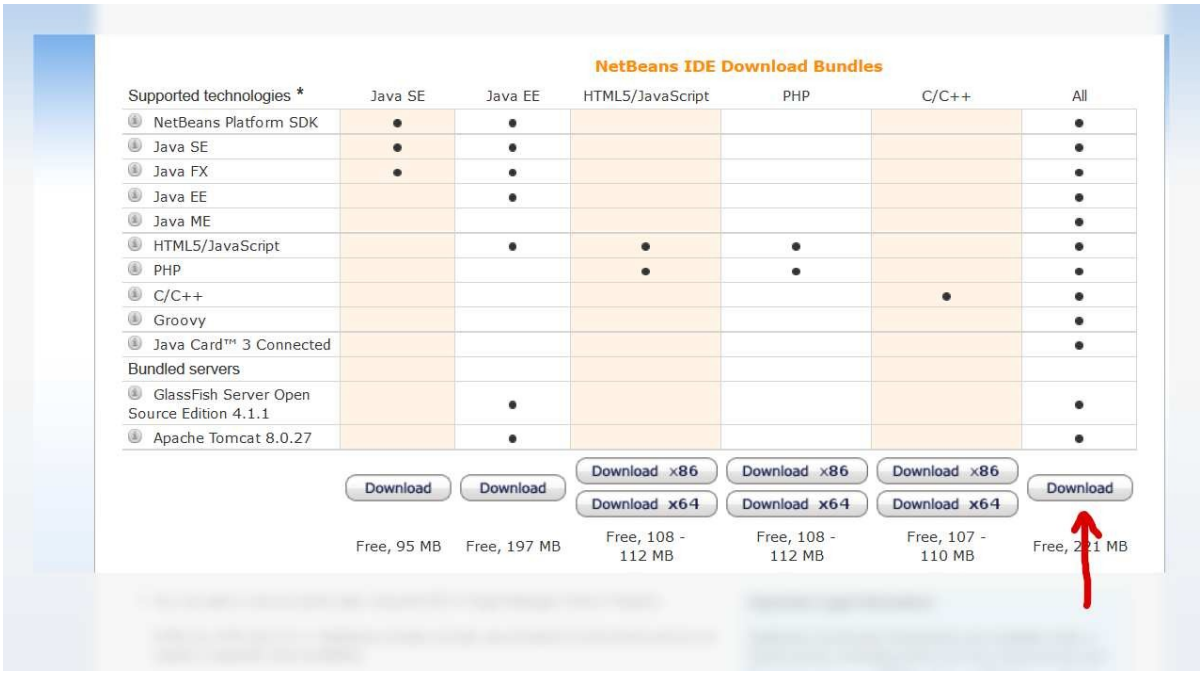
- Téléchargez le **Pack JDK 8 si vous ne l'avez pas**

http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

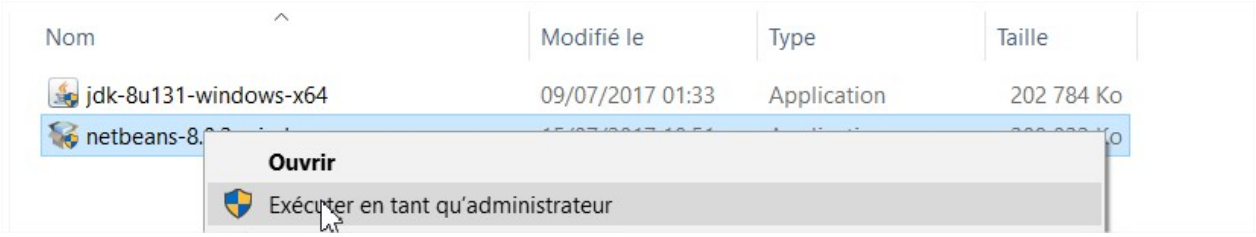
- Téléchargez le **NetBeans 8.0.2**:

https://netbeans.org/downloads/8.0.2/

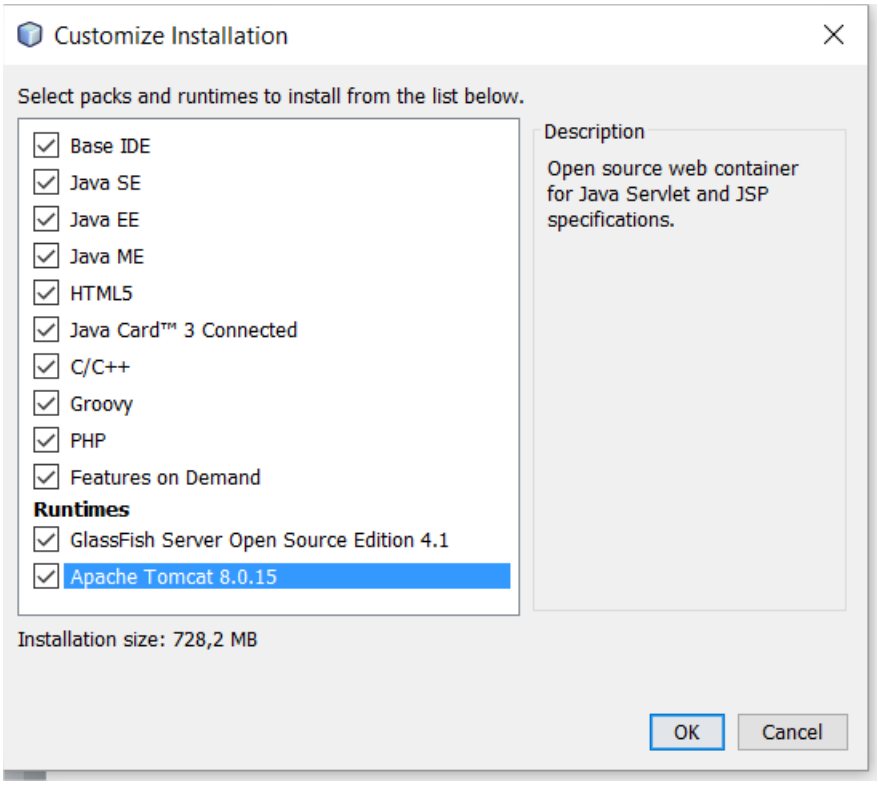
Prendre la version Full avec GlassFish installé dessus.



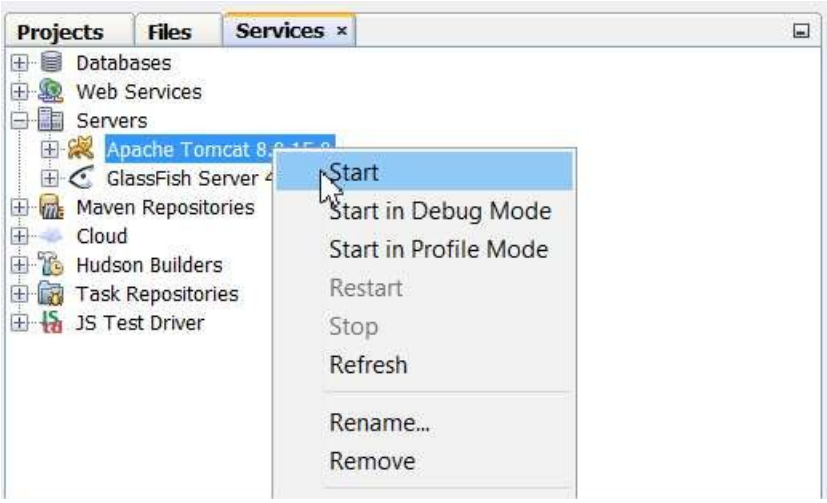
Installez **NetBeans 8.0.2** en tant qu'Administrateur.

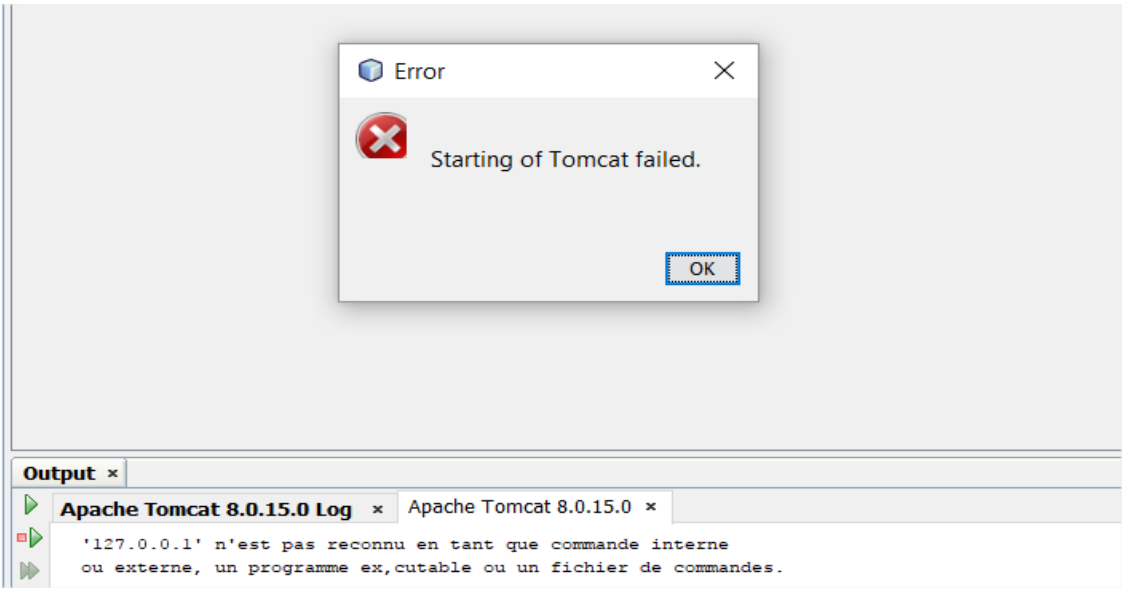


Cliquez sur **Customize** pour sélectionner **Tomcat 8.0.15** pour l'installation, puis OK, puis Next.
Si vous voulez utiliser le serveur d'application par défaut de Java, c'est-à-dire **GlassFish,** alors vous pouvez ne pas installer Tomcat et, dans ce cas-là, nul besoin d'éxécuter le reste de la procedure ci-dessous concernant l'installation de **NetBeans 8.0.2**.


Customize Installation
Select packs and runtimes to install from the list below.
Base IDE
Java SE
Java EE
Java ME
HTML5
Java Card™ 3 Connected
C/C++
Groovy
PHP
Features on Demand
Runtimes
GlassFish Server Open Source Edition 4.1
Apache Tomcat 8.0.15
Description
Open source web container for Java Servlet and JSP specifications.
Installation size: 728,2 MB
OK
Cancel


Projects
Files
Services
Databases
Web Services
Servers
Apache Tomcat 8.0.15.0
GlassFish Server 4.1
Maven Repositories
Cloud
Hudson Builders
Task Repositories
JS Test Driver

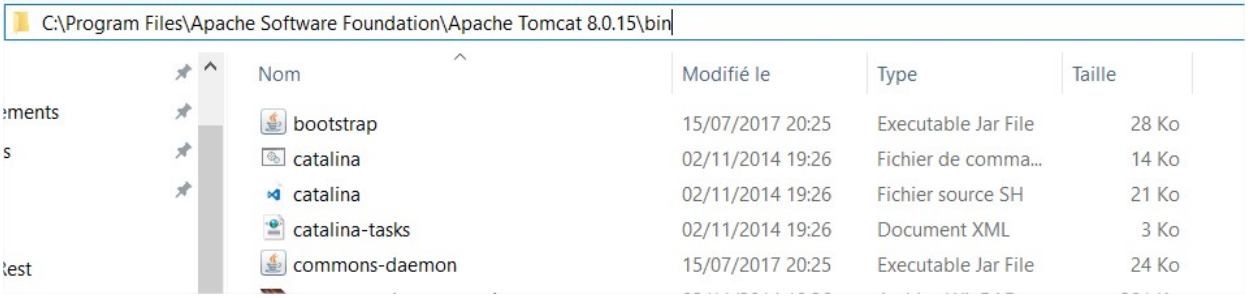




La console de Tomcat affiche l'erreur suivante :

'127.0.0.1' n'est pas reconnu en tant que commande interne

Pour résoudre ce problème, il faudra modifier le fichier catalina.bat dans **C:\Program Files\Apache Software Foundation\Apache Tomcat 8.0.15\bin**



Recherchez dans ce fichier **JAVA_OPTS**.
Regardons de plus près les lignes **196** et **201** de **catalina.bat**.

```
190
191 if not "%LOGGING_CONFIG%" == "" goto noJuliConfig
192 set LOGGING_CONFIG=-Dnop
193 if not exist "%CATALINA_BASE%\conf\logging.properties" goto noJuliConfig
194 set LOGGING_CONFIG=-Djava.util.logging.config.file="%CATALINA_BASE%\conf\logging.properties"
195 :noJuliConfig
196 set "JAVA_OPTS=%JAVA_OPTS% %LOGGING_CONFIG%"
197
198 if not "%LOGGING_MANAGER%" == "" goto noJuliManager
199 set LOGGING_MANAGER=-Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager
200 :noJuliManager
201 set "JAVA_OPTS=%JAVA_OPTS% %LOGGING_MANAGER%"
```

Remplacez :

set "JAVA_OPTS=%JAVA_OPTS% %LOGGING_CONFIG%"

Par

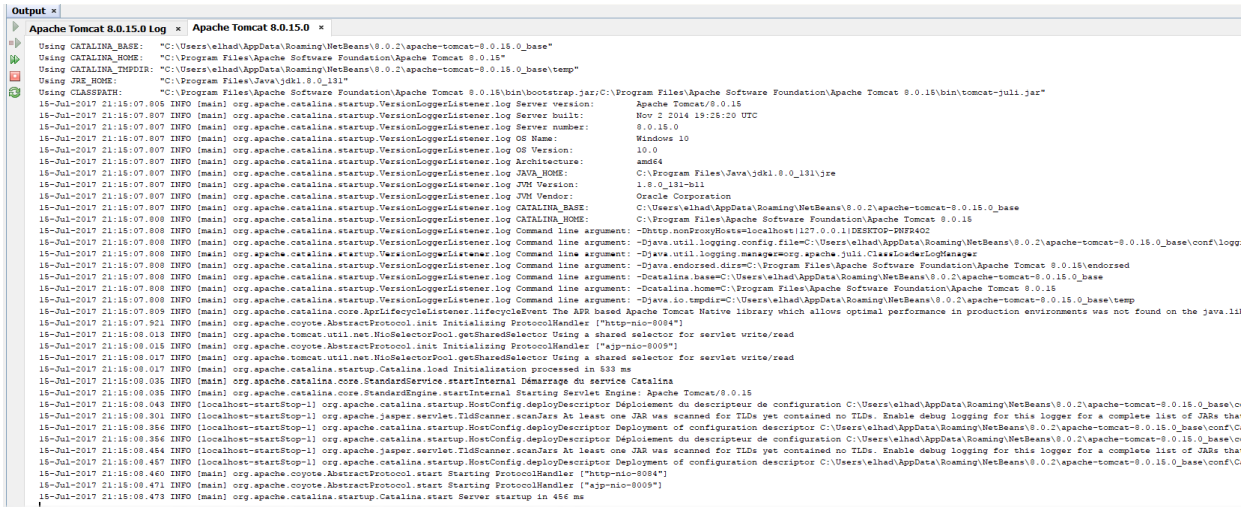
set JAVA_OPTS=%JAVA_OPTS% %LOGGING_CONFIG%

Remplacez :

set "JAVA_OPTS=%JAVA_OPTS% %LOGGING_MANAGER%"

Par

set JAVA_OPTS=%JAVA_OPTS% %LOGGING_MANAGER%



21:15:08.473 INFO [main] org.apache.catalina.startup.Catalina.start Server startup in 456 ms
Après ces modifications, le serveur d'application **Tomcat** démarre en quelques millisecondes.

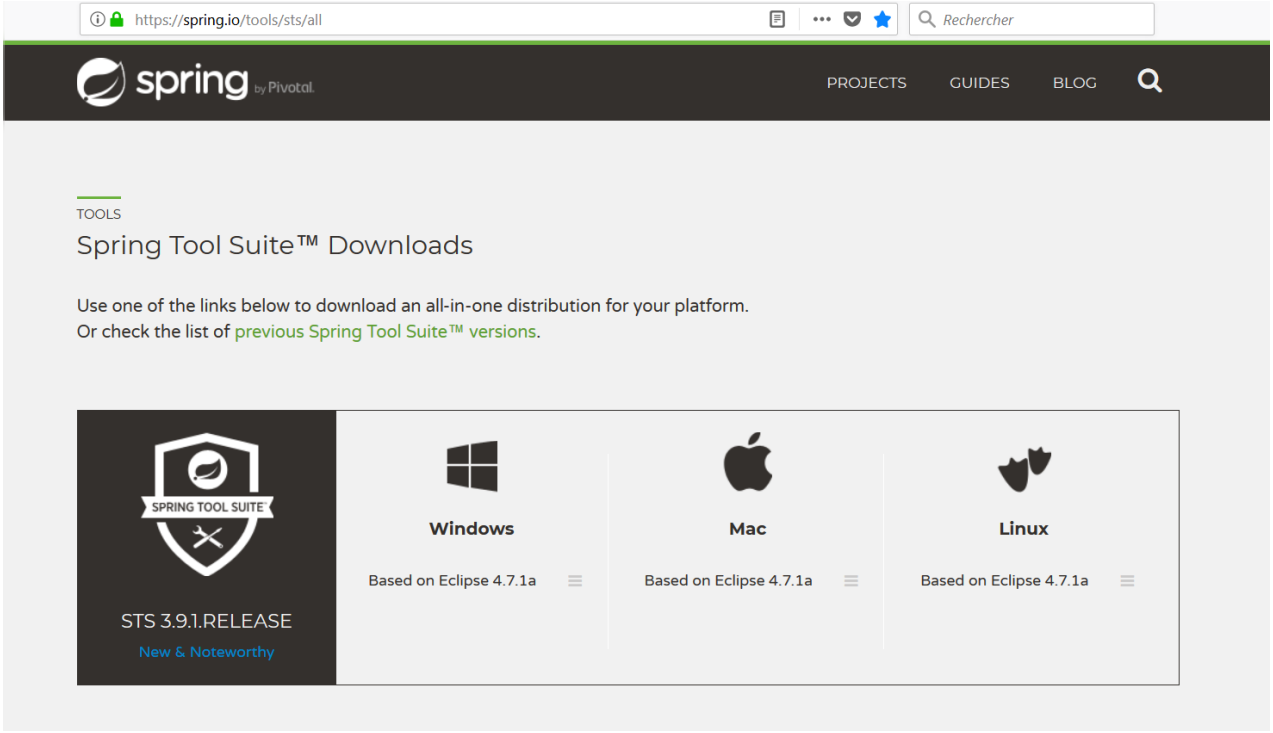
IV) Installation de Spring Tool Suite.

- Téléchargez le **JDK 8 si vous ne l'avez pas**

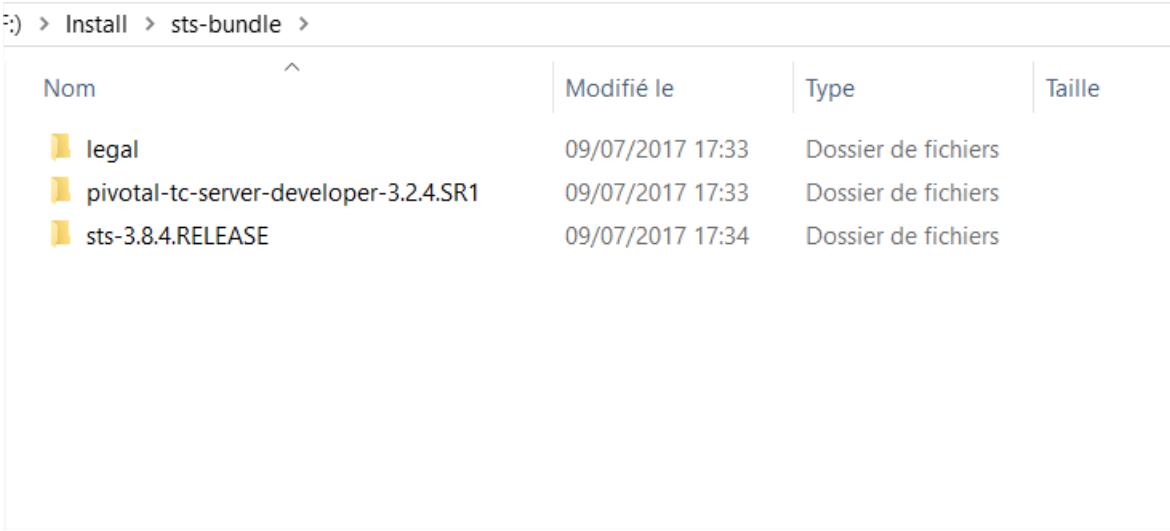
http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

- Téléchargez **Spring Tool Suite**

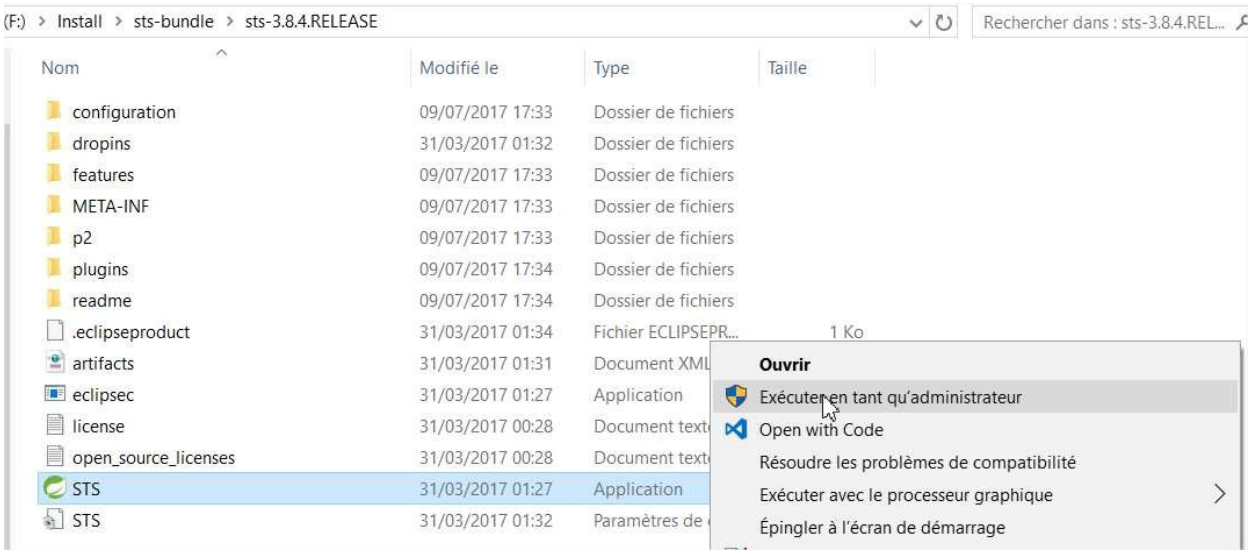
https://spring.io/tools/sts/all



Une fois que vous avez fini de télécharger le fichier, le mettre dans le répertoire que vous voulez.



Allez dans le répertoire **sts-bundle\sts-3.8.4.RELEASE** puis lancer l'éxecutable en tant qu'Administrateur.



V) Initialisation de la base de données MYSQL.

1. Téléchargement de Wamp ou MYSQL

Pour télécharger Wamp Serveur : http://www.wampserver.com/

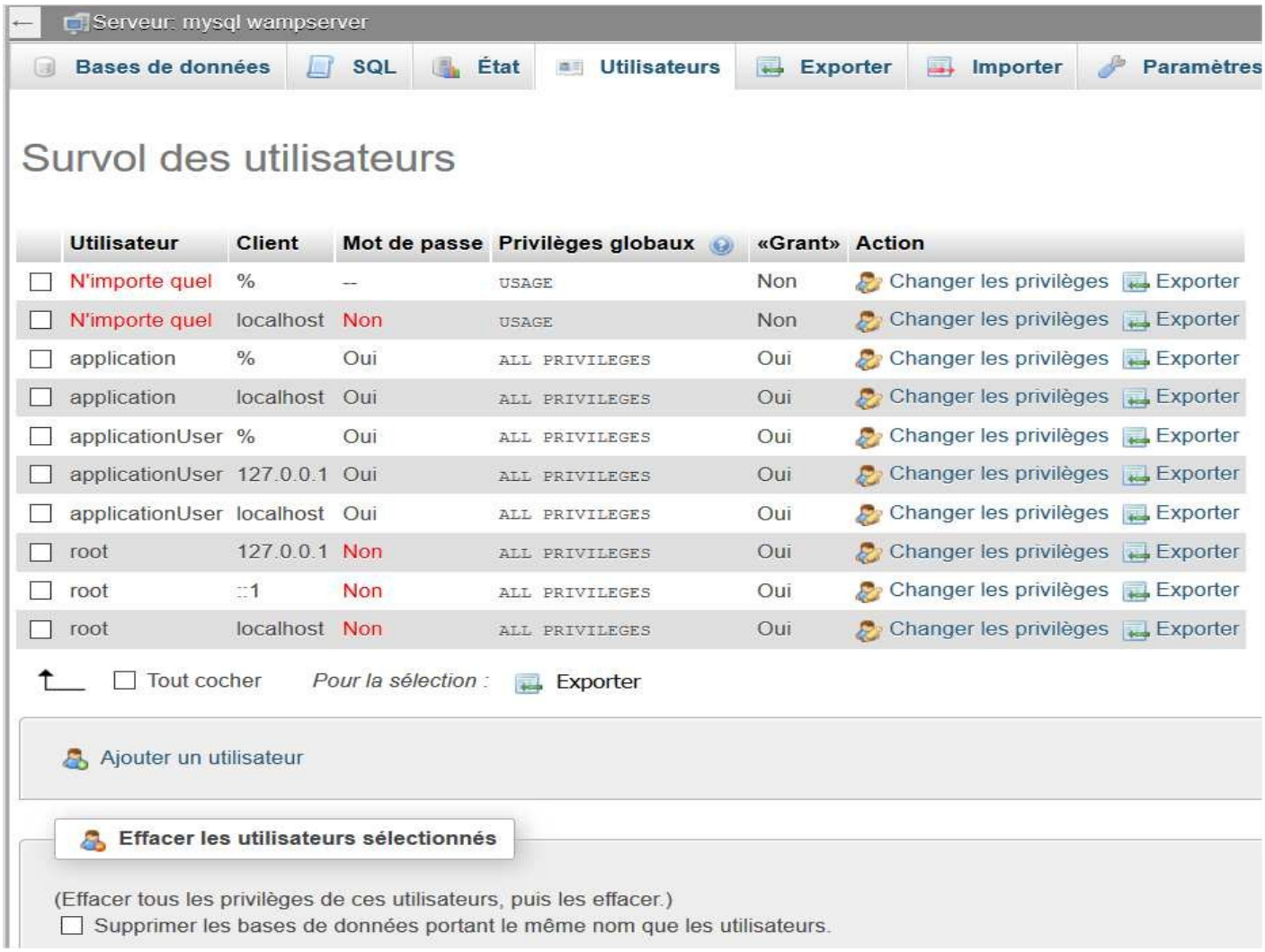
Pour télécharger MySQL : http://dev.mysql.com/downloads/

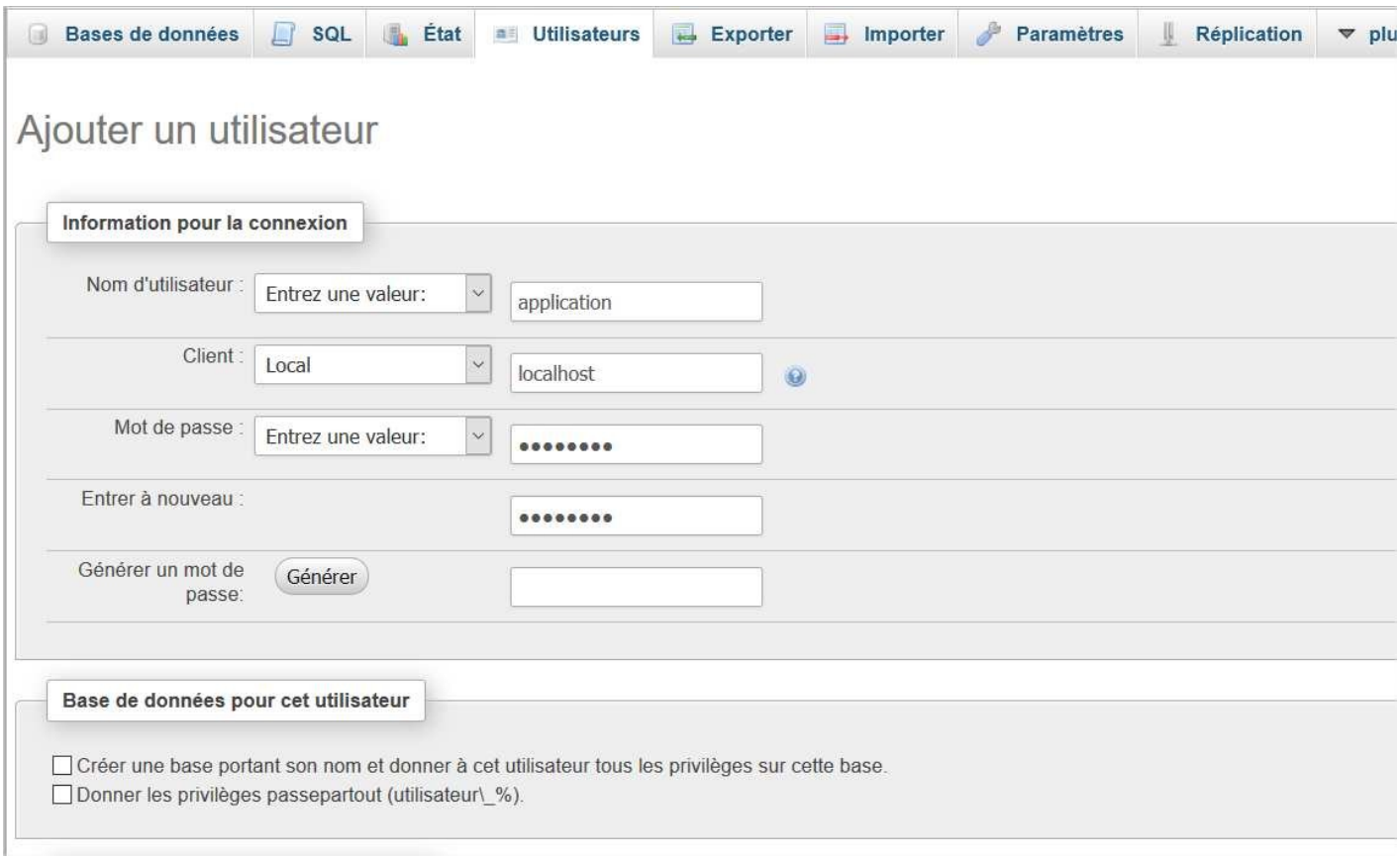
2. **Création de la base de données « base_test » :**

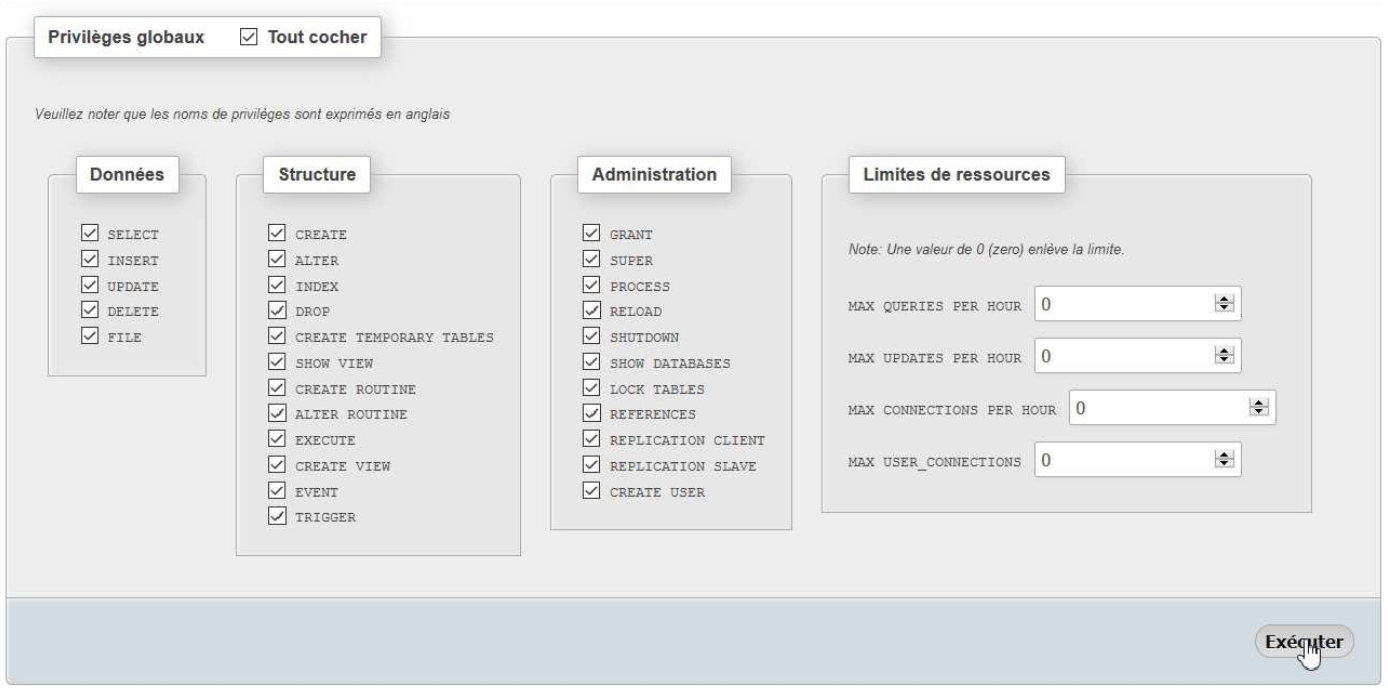
Créez la base MySQL [base_test] avec l'outil de votre choix. La base sera, par la suite, la propriété de l'utilisateur « **application** » avec le mot de passe « **passw0rd** ».

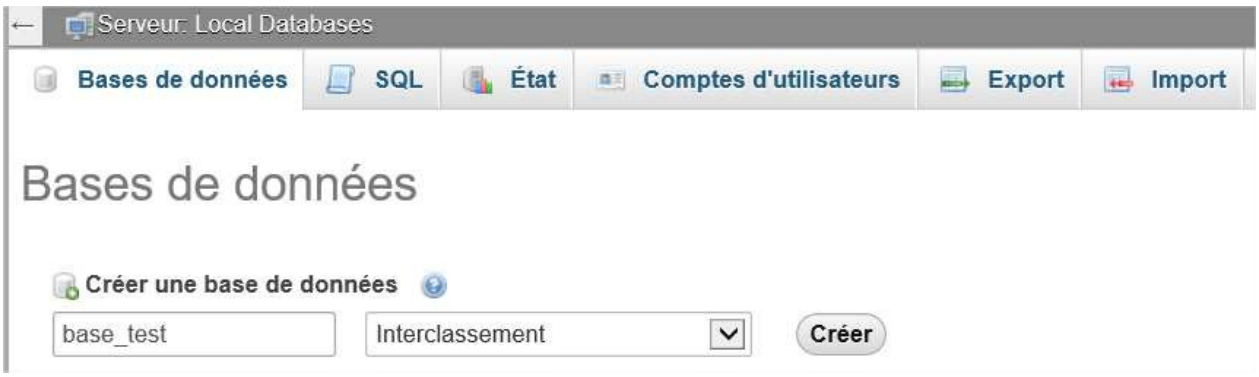
Pour ajouter un nouvel utilisateur :

- Allez dans la console d'administration de **PhpMyAdmin**, puis cliquez sur « **Utilisateur** », puis sur « **Ajouter un utilisateur** ».
- Mettre « **application** » dans « Nom d'utilisateur », « **passw0rd** » dans « Mot de passe » et « **localhost** » dans « Client », Cochez « **Tout cocher** » dans « Privilèges globaux » et cliquez sur « **Exécuter** ».
- Réitérer l'opération précédente en mettant pour le client « **Tout Client** » c'est-à-dire la valeur **%**, puis la valeur « **127.0.0.1** ».



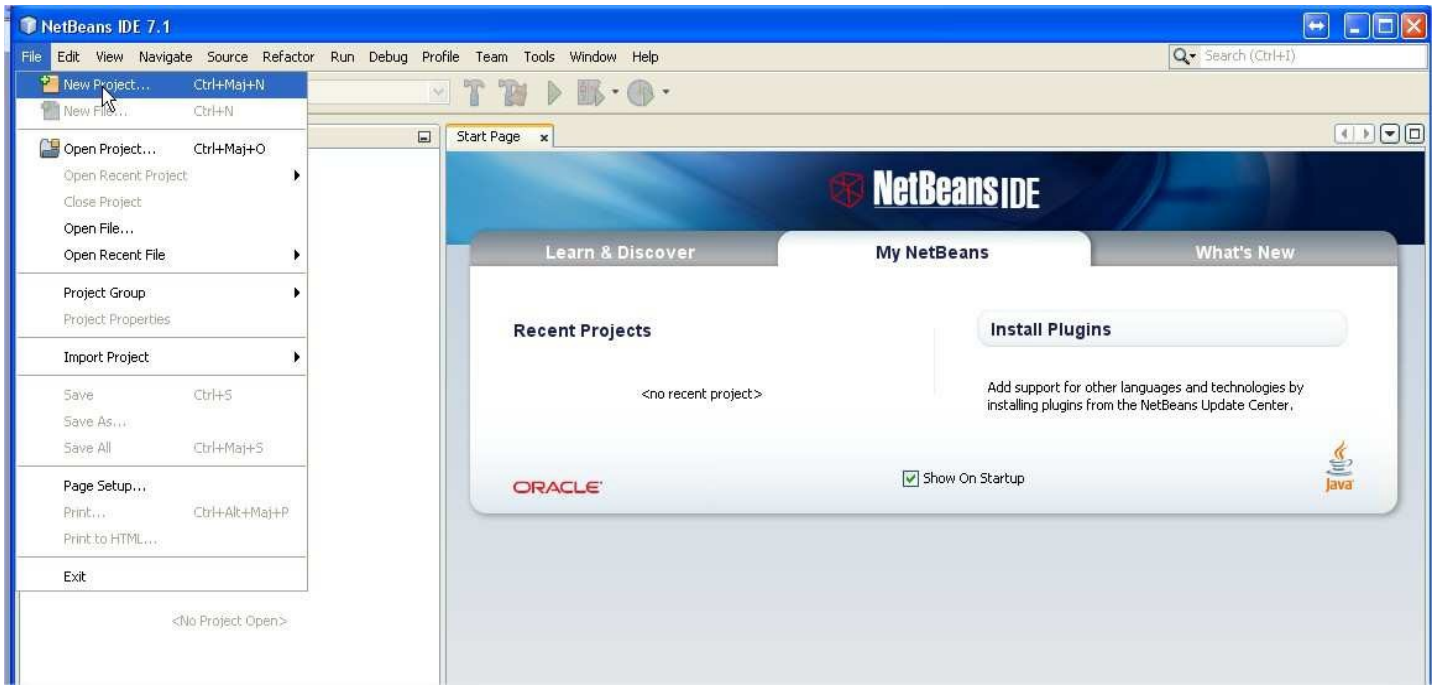

Bases de données
SQL
État
Utilisateurs
Exporter
Importer
Paramètres
Réplication
plu
Ajouter un utilisateur
Information pour la connexion
Nom d'utilisateur :
Entrez une valeur:
application
Client :
Local
localhost
Mot de passe :
Entrez une valeur:
Entrer à nouveau :
Générer un mot de passe:
Générer
Base de données pour cet utilisateur
Créer une base portant son nom et donner à cet utilisateur tous les privilèges sur cette base.
Donner les privilèges passepartout (utilisateur_%).


Privilèges globaux
Tout cocher
Veuillez noter que les noms de privilèges sont exprimés en anglais
Données
SELECT
INSERT
UPDATE
DELETE
FILE
Structure
CREATE
ALTER
INDEX
DROP
CREATE TEMPORARY TABLES
SHOW VIEW
CREATE ROUTINE
ALTER ROUTINE
EXECUTE
CREATE VIEW
EVENT
TRIGGER
Administration
GRANT
SUPER
PROCESS
RELOAD
SHUTDOWN
SHOW DATABASES
LOCK TABLES
REFERENCES
REPLICATION CLIENT
REPLICATION SLAVE
CREATE USER
Limites de ressources
Note: Une valeur de 0 (zero) enlève la limite.
MAX QUERIES PER HOUR
0
MAX UPDATES PER HOUR
0
MAX CONNECTIONS PER HOUR
0
MAX USER_CONNECTIONS
0
Exécuter

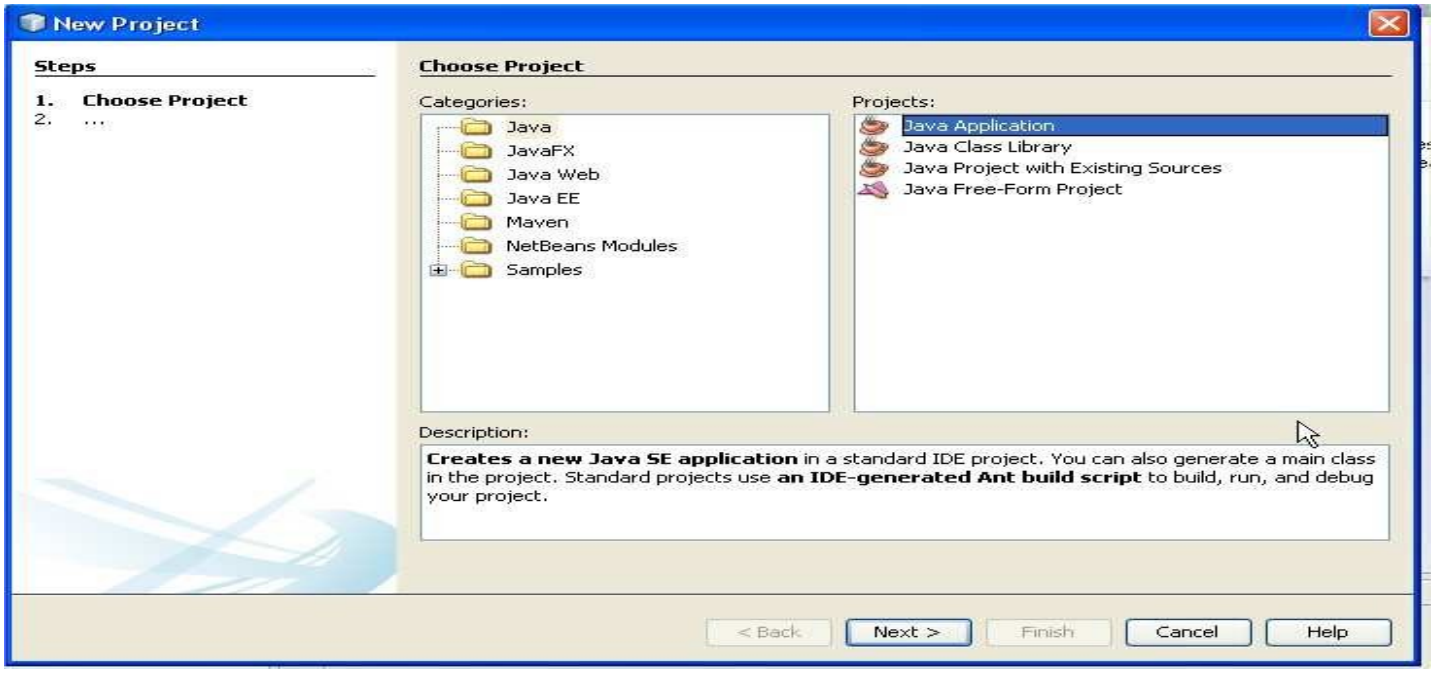

Serveur: Local Databases
Bases de données
SQL
État
Comptes d'utilisateurs
Export
Import
Bases de données
Créer une base de données
base_test
Interclassement
Créer

VI) Notre première application Java

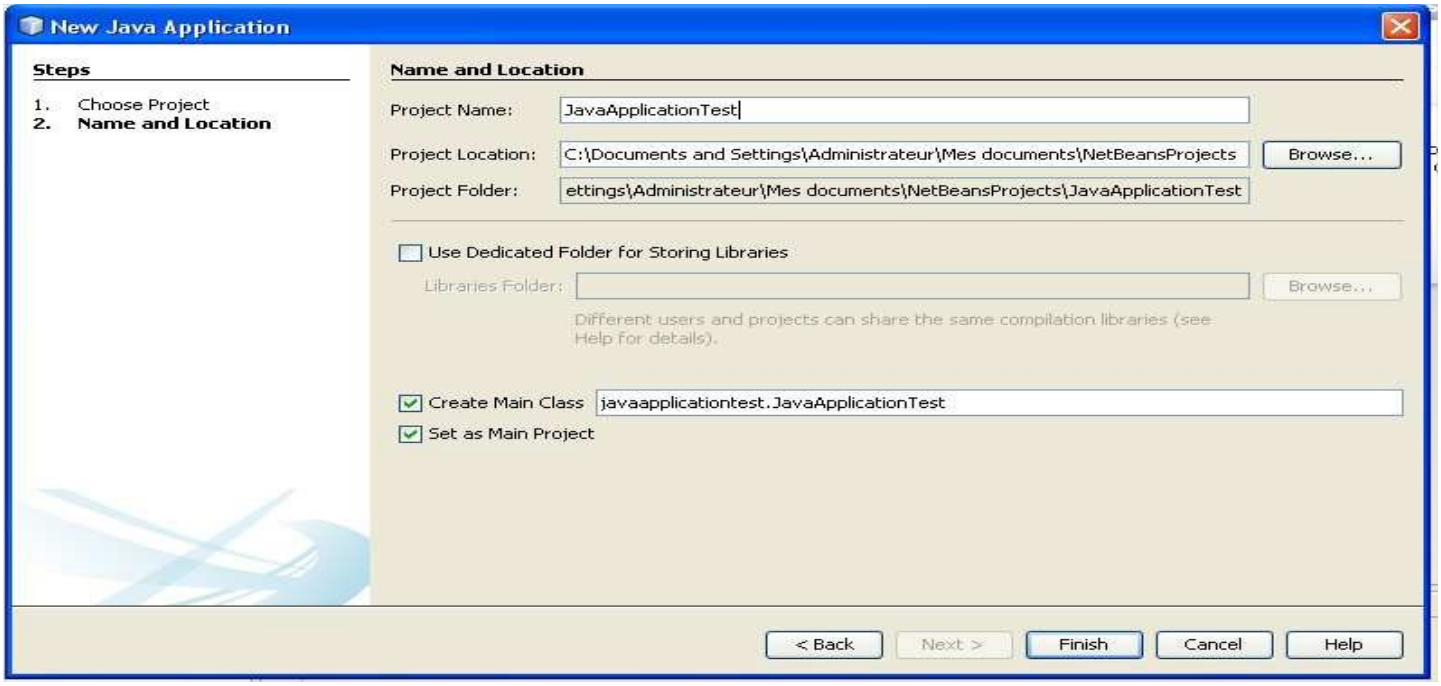
Ouvrir l'IDE et créer votre première application Java en cliquant sur 👁 File 👁 New Project.



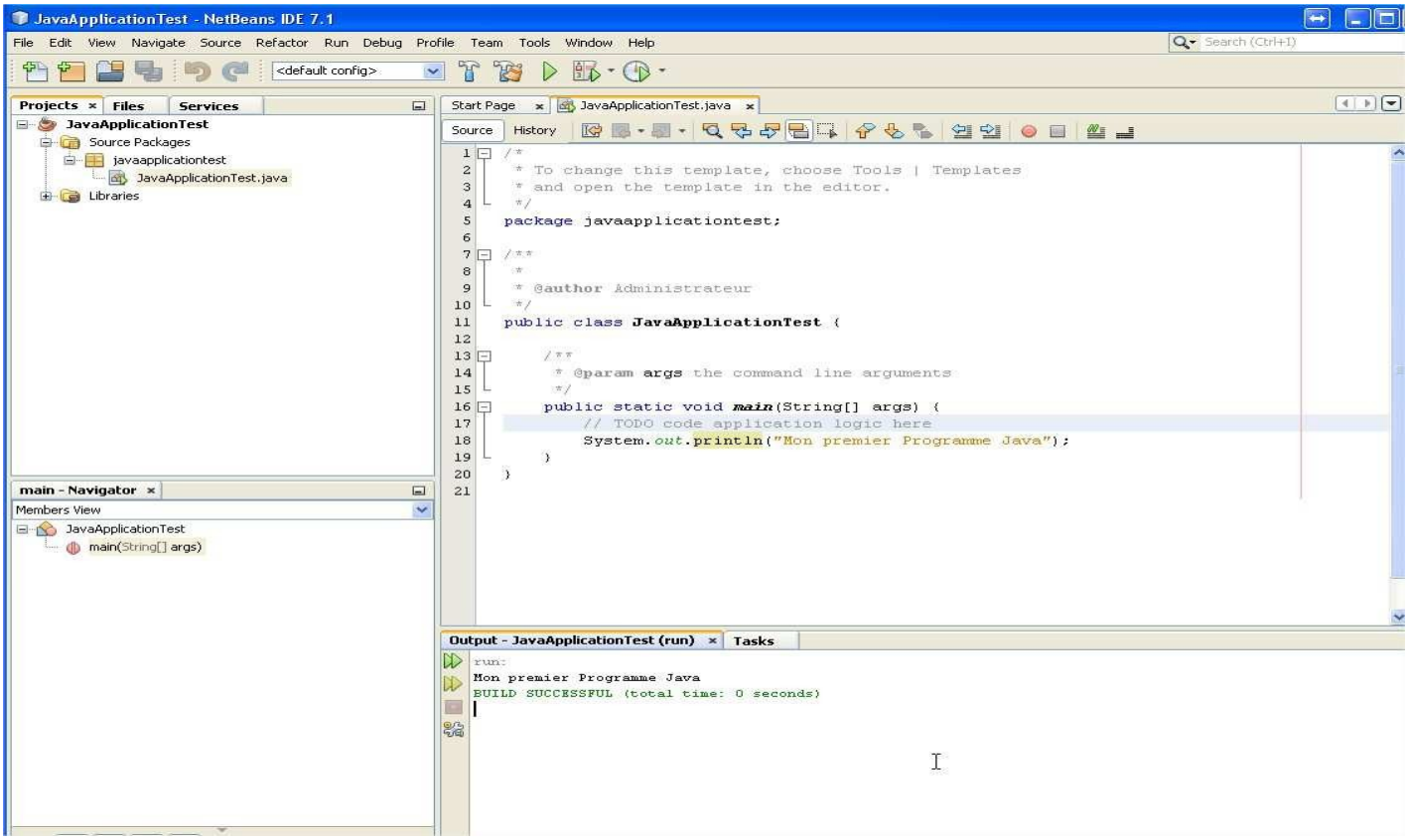
Choisir «Java Application»



Mettre le nom de votre Application, vous pouvez cocher « Set as Main Project » pour créer avec la classe « JavaApplicationTest » avec une méthode main pour exécuter les programmes en mode console.



Réalisons notre premier programme java, ce programme va juste afficher « Mon premier Programme Java »



Ligne 16: déclare la méthode « main » de l'application, comme le mot l'indique « main » veut dire principale. Cette méthode est le point d'entrée des tous les appels de l'application.
Pour compiler et exécuter le programme il suffit d'appuyer sur F6 en mode « Debug » il suffit de faire Ctlr+F5.

VII) Les types primitifs de Java

Le type int

Type	Plage	Taille	Objet correspondant	Valeur par défaut
Int	-2 147 483 648 à 2 147 483 647	4 octets	java.lang.Integer	0

Le type long

Type	Plage	Taille	Objet correspondant	Valeur par défaut
Long	-9,223,372,036,854,775,808 à 9,223,372,036,854,775,807	8 octets	java.lang.Long	0

Le type float

Type	Plage	Taille	Objet correspondant	Valeur par défaut
Float	-1.4 * 10-45 à +3.4 * 1038	4 octets	java.lang.Float	0

Le type double

Type	Plage	Taille	Objet correspondant	Valeur par défaut
double	4.9 * 10-324 à +1.7 * 10308	8 octets	java.lang.Double	0

Le type byte

Type	Plage	Taille	Objet correspondant	Valeur par défaut
byte	-128 à +127	1 octet	java.lang.Byte	0

Le type short

Type	Plage	Taille	Objet correspondant	Valeur par défaut
short	-32768 à +32767	2 octets	java.lang.Short	0

Le type char

Type	Plage	Taille	Objet correspondant	Valeur par défaut
Char	Caractère Unicode (65536 caractères possibles)	2 octets	java.lang.Character	

Le type boolean

Type	Valeurs	Taille	Objet correspondant	Valeur par défaut
boolean	true ou false	1 octet	java.lang.Boolean	

VIII) Structure et condition

Pour les conditions, il suffit de respecter les syntaxes suivantes:

```
if (condition1){
    code1;
} else {
    code2;
}
```

Exemple :

```
30
31          int entier1 = 4, entier2 = 6;
32          if (entier1 > entier2) {
33              System.out.println("L'entier 1 est plus grand.");
34          } else {
35              System.out.println("L'entier 2 est plus grand.");
36          }
```

```
switch(choix) {
        case 1:
                code1;
                break;
        case2:
                code2;
                break;
        .  ..   ..  ..   ..   ..
        default:
                code_par_defaut;
                break;
}
```

```
35
36          int choix = 0;
37          switch (choix) {
38              case 1:
39                  code1();
40                  break;
41              case 2:
42                  code2();
43                  break;
44              case 3:
45                  code3();
46                  break;
47              default:
48                  codeParDefaut();
49                  break;
50          }
51
```

IX) Passage entre les types primitifs

1) Type quelconque vers chaîne de caractères

Soit un type quelconque pour convertir ce type en chaîne de caractères, il suffit d'utiliser «toString()»

2) Type chaîne de caractères vers entier

Soit une chaîne de caractères « chaine » pour convertir « chaine » en entier il suffit de faire :

Integer.parseInt (chaine) ou Integer.valueOf (chaine)

3) Type chaîne de caractères vers long

Soit une chaîne de caractères « chaine » pour convertir « chaine » en long, il suffit de faire :

Long.parseLong (chaine) ou Long.valueOf (chaine)

4) Type chaîne de caractères vers double

Soit une chaîne de caractères « chaine » pour convertir « chaine » en double, il suffit de faire :

Double.parseDouble (chaine) ou Double.valueOf (chaine)

5) Type chaîne de caractères vers float

Soit une chaîne de caractères « chaine » pour convertir « chaine » en float, il suffit de faire :

Float.parseFloat (chaine) ou Float.valueOf (chaine)

Les opérations de «Parse» peuvent déclencher des erreurs si elles échouent, d'où la nécessité de les entourer par la clause *try/catch* .

```
41
42          int entier = 0;
43          try {
44              entier = Integer.parseInt("10");
45          } catch (Exception e) {
46              System.out.println("Erreur lors du Parse, Exception: " + e.getMessage());
47          } finally {
48              System.out.println("Finalement");
49          }
```

X) Les tableaux

Un tableau est un objet permettant de stocker plusieurs données de même type.
Sa déclaration est la suivante :

Type [] tableau =new Type[n] ; où Type est le type d'objet stocke dans le tableau et n la taille du tableau.

Exemple : pour un tableau de 10 entiers on aura :

int[] entiers=new int[10] ou int entiers[]=new int[10]

On peut déclarer et initialiser directement un tableau.

int entiers [] = {0,1,2,3,4,5,6,7,8,9};

Type [][] tableau=new Type [n] [m]; n étant le nombre de ligne et m le nombre de colonne où tableau [i, j] désigne l'élément se trouvant à la ligne i et colonne j.

XI) Affichage sur écran

L'affichage sur écran en java est géré par la classe «java.io.PrintStream». Il existe différentes façons d'exploiter les flux de sortie. On distingue deux flux : le flux Out et le flux Err.
Exemple :
System.out.println("Mon Message") pour le flux Out.

System.err.println("Mon erreur") pour le flux Err.

Pour les variables de type quelconque (en particulier String mais aussi un objet quelconque Objet):

```
String maVariable = "test";
System.out.println("maVariable: " + maVariable);
System.out.println(String.format("maVariable: %s", maVariable));
```

Pour les variables de type «int » :

```
int maVariable = 0;
System.out.println("maVariable: " + maVariable);
System.out.println(String.format("maVariable: %d", maVariable));
```

XII) Lire des données avec le clavier

Il existe plusieurs façons de gérer les transferts avec le clavier. On peut utiliser les classes «java.util.Scanner» et «java.io.InputStreamReader».

Exemple avec la classe Scanner :

```
37
38              Scanner scan = new Scanner(System.in);
39              System.out.println("Entrée la chaine:");
40              String chaine=scan.next();
41
```

Exemple avec la classe InputStreamReader :

```
42
43          try {
44              InputStreamReader lecteur = new InputStreamReader(System.in);
45              System.out.println("Entrée la chaine:");
46              BufferedReader entree = new BufferedReader(lecteur);
47              String chaine = entree.readLine();
48          } catch (IOException ex) {
49
50          }
```

XIII) Les opérations mathématiques.

On peut énumérer les opérateurs suivants en Java :

L'addition représentée par le signe « + ».
La soustraction représentée par le signe « - ».
La multiplication représentée par le signe « * ».
La division représentée par le signe « / ».
Le modulo représentée par le signe « % ».

Il existe un certain nombre d'autres opérations contenues dans la classe «java.lang.Math».

On peut citer parmi les méthodes de cette classe :

- double sqrt (double x) : racine carrée

- double cos (double x) : cosinus

- double sin (double x) : sinus

- double tan (double x) : tangente

- double pow (double x, double y) : x à la puissance y

- double exp (double x) : exponentielle

- double log (double x) : logarithme népérien

- double abs (double x) : valeur absolue

XIV) Les chaînes de caractères

L'objet « String » permet de manipuler les chaînes de caractères. En programmation orientée objet on ne peut faire des tests d'égalité (==) et de non égalité (!=) avec les objets en particulier l'objet « String » en utilisant les opérateurs habituels (égalité == et non égalité !=). En effet, on doit obligatoirement utiliser la méthode «equals».
Pour faire des comparaisons de supériorité, on peut utiliser la méthode «compareTo».

Exemple :

```
51          String chaine1="toto";
52          String chaine2="tata";
53          int result=chaine1.compareTo(chaine2);
54          System.out.println("result: "+result);
55      }
```

Si « result » est égale 0 alors les deux chaînes sont égales.
Si « result » est supérieur à zéro alors les chaine1 est supérieur à chaine2.
Si « result » est inférieur à zéro alors les chaine1 est inférieur à chaine2.

XV) Les boucles

Il existe plusieurs façons de traiter des boucles en Java :

Structure for

```
for (instructions_debut;condition;instructions_fin){
        code_a_executer();
}
```

Exemple:

```
56
57          int entiers[] = {11, 12, 42, 35, 46, 56, 69, 70, 88, 96};
58
59          for (int i = 0; i < entiers.length; i++) {
60              System.out.println("entiers[" + i + "]=" + entiers[i]);
61          }
62
63          //ou bien
64
65          for (int entier : entiers) {
66              System.out.println("entier=" + entier);
67          }
```

Structure tant que (while)

```
while (condition){
        code_a_executer();
}
```

Exemple:

```
69
70          int i = 0;
71          while (i < entiers.length) {
72              System.out.println("entiers[" + i + "]=" + entiers[i]);
73              i++;
74          }
```

Structure répétée jusqu'à ce que (do while)

```
do {
        code_a_executer();
} while (condition);
```

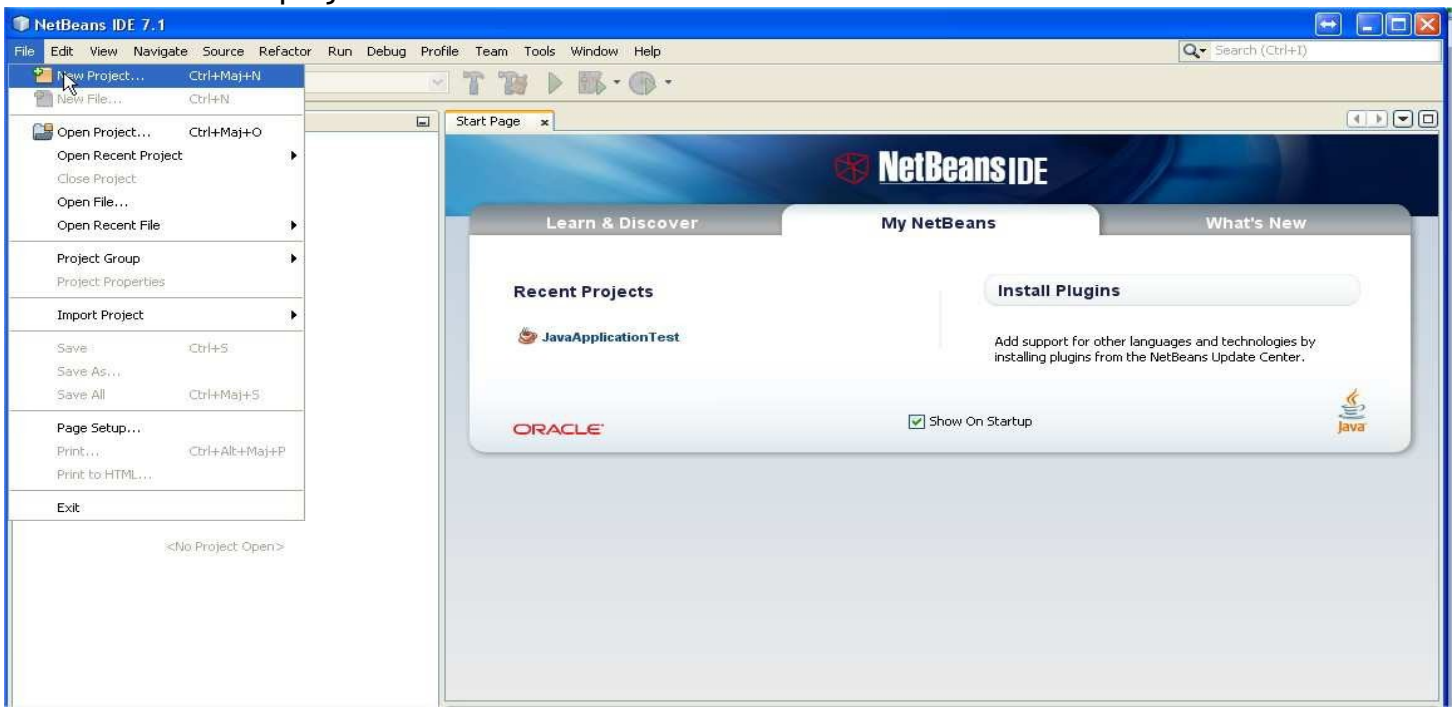
Exemple :

```
75
76          int i = 0;
77          do {
78              System.out.println("entiers[" + i + "]=" + entiers[i]);
79              i++;
80          } while (i < entiers.length);
```

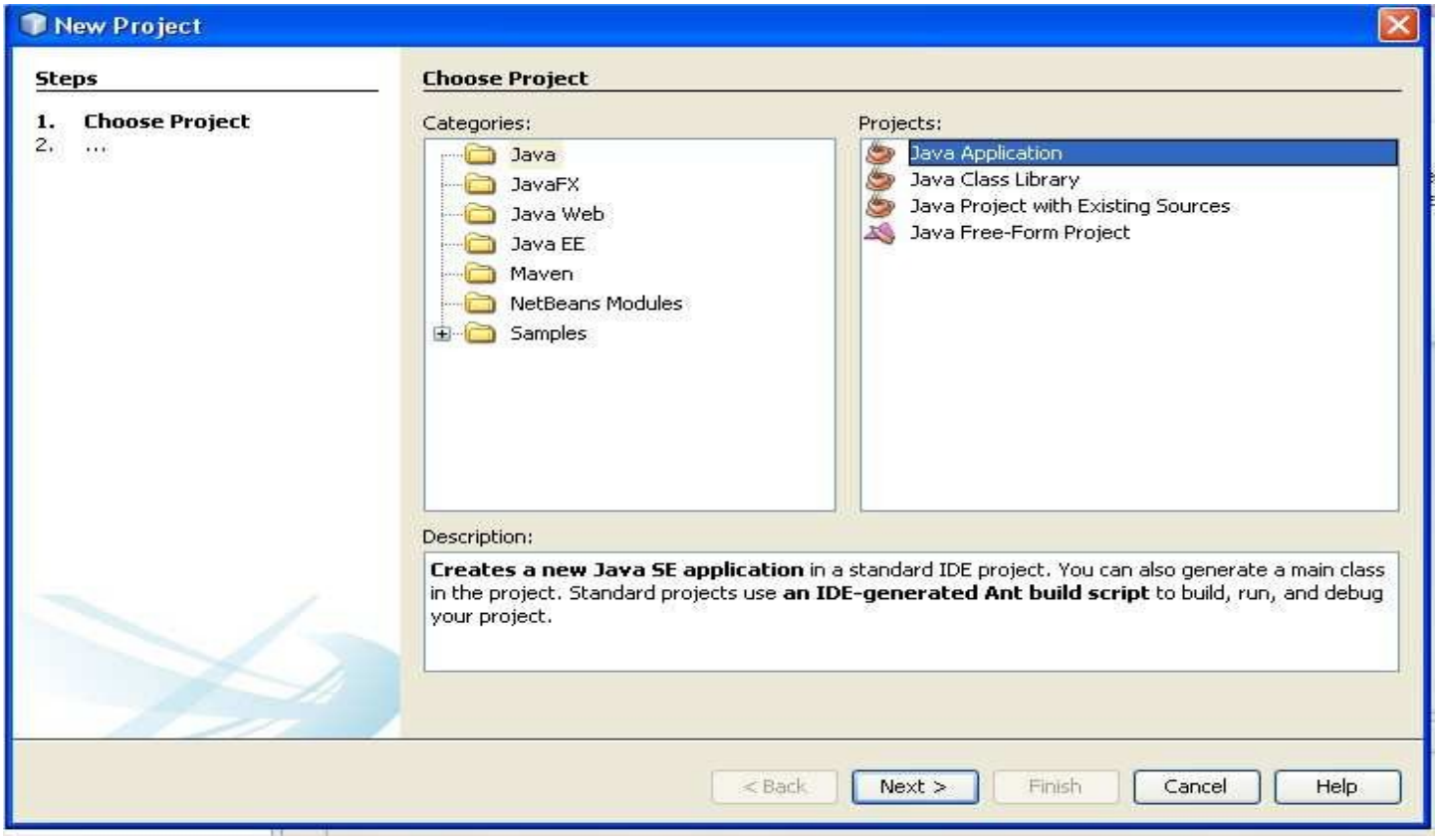
XVI) Les Objets en Java

Nous allons maintenant voir comment créer des classes en Java.

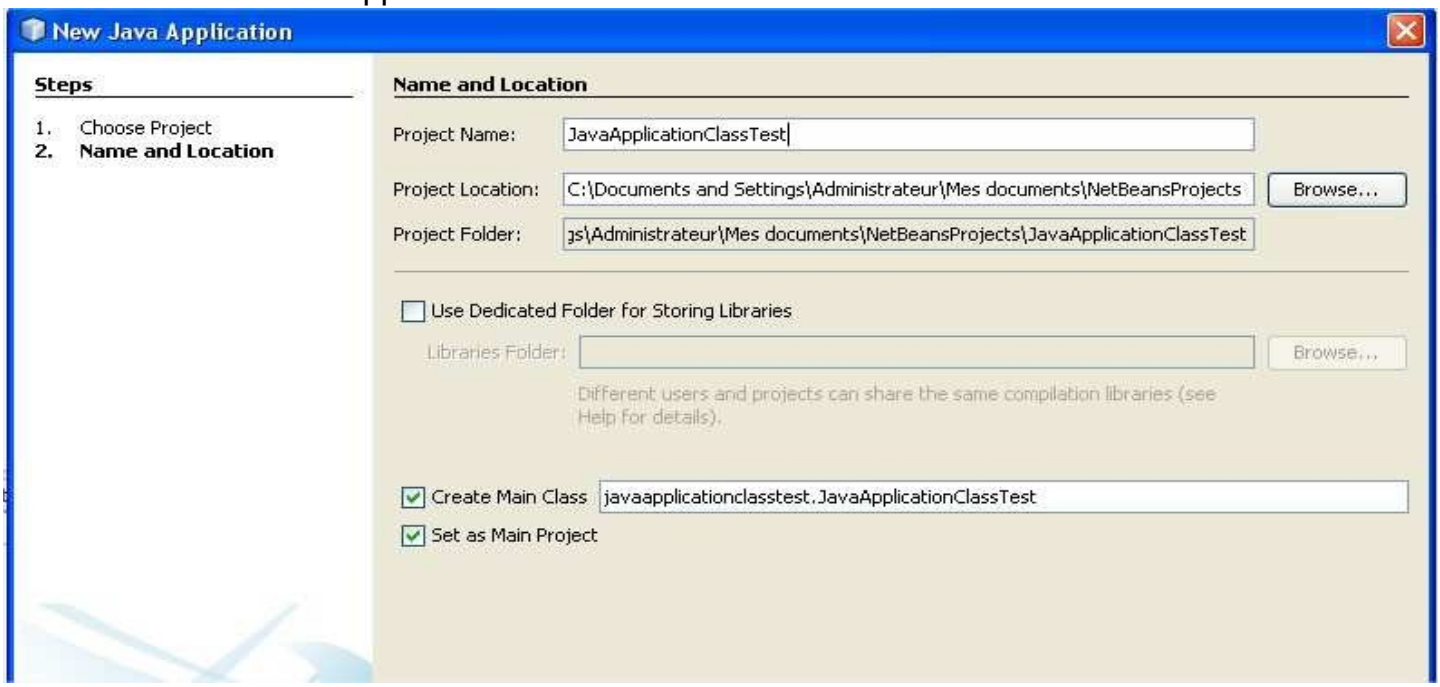
Créons un nouveau projet



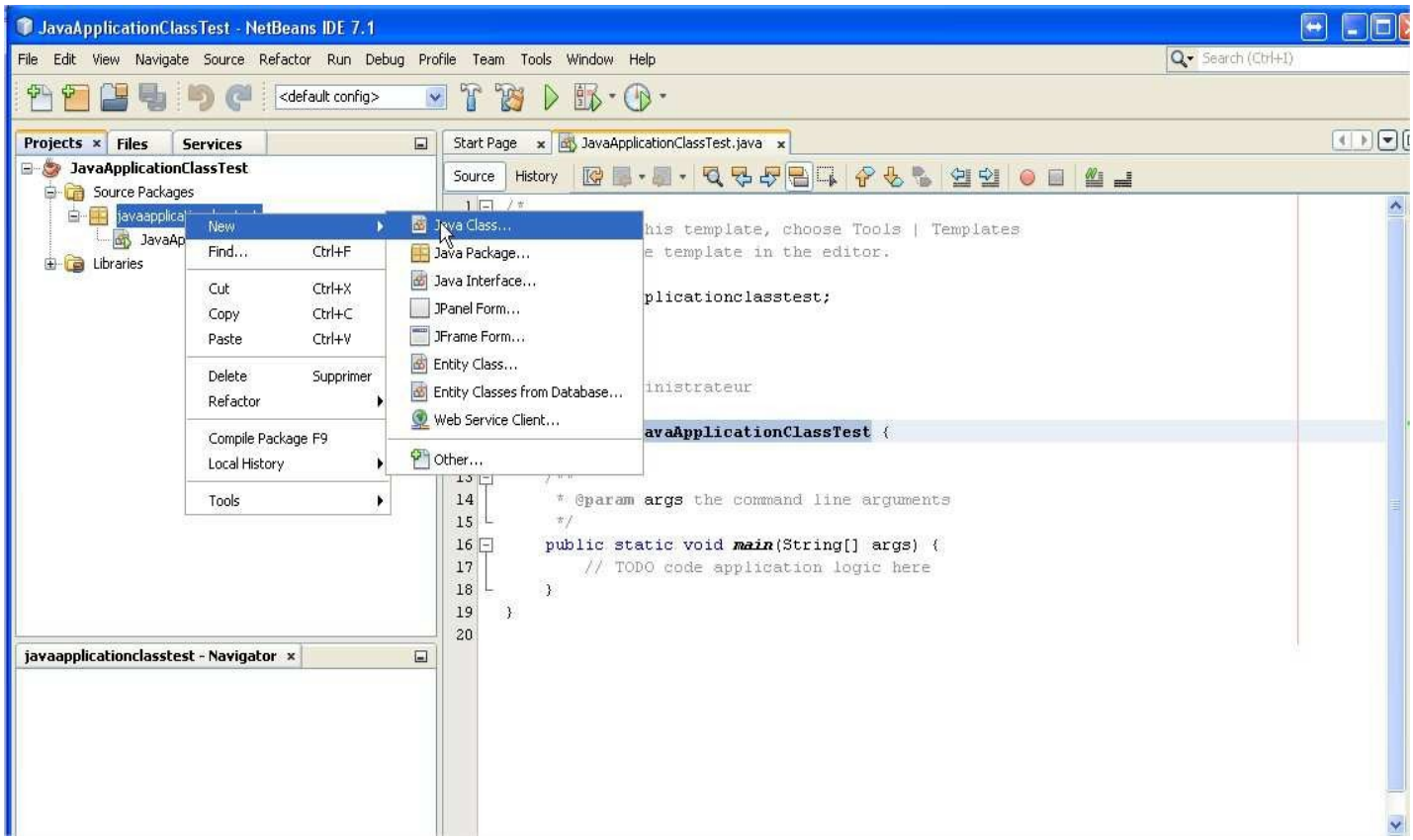
Choisir «Java Application»



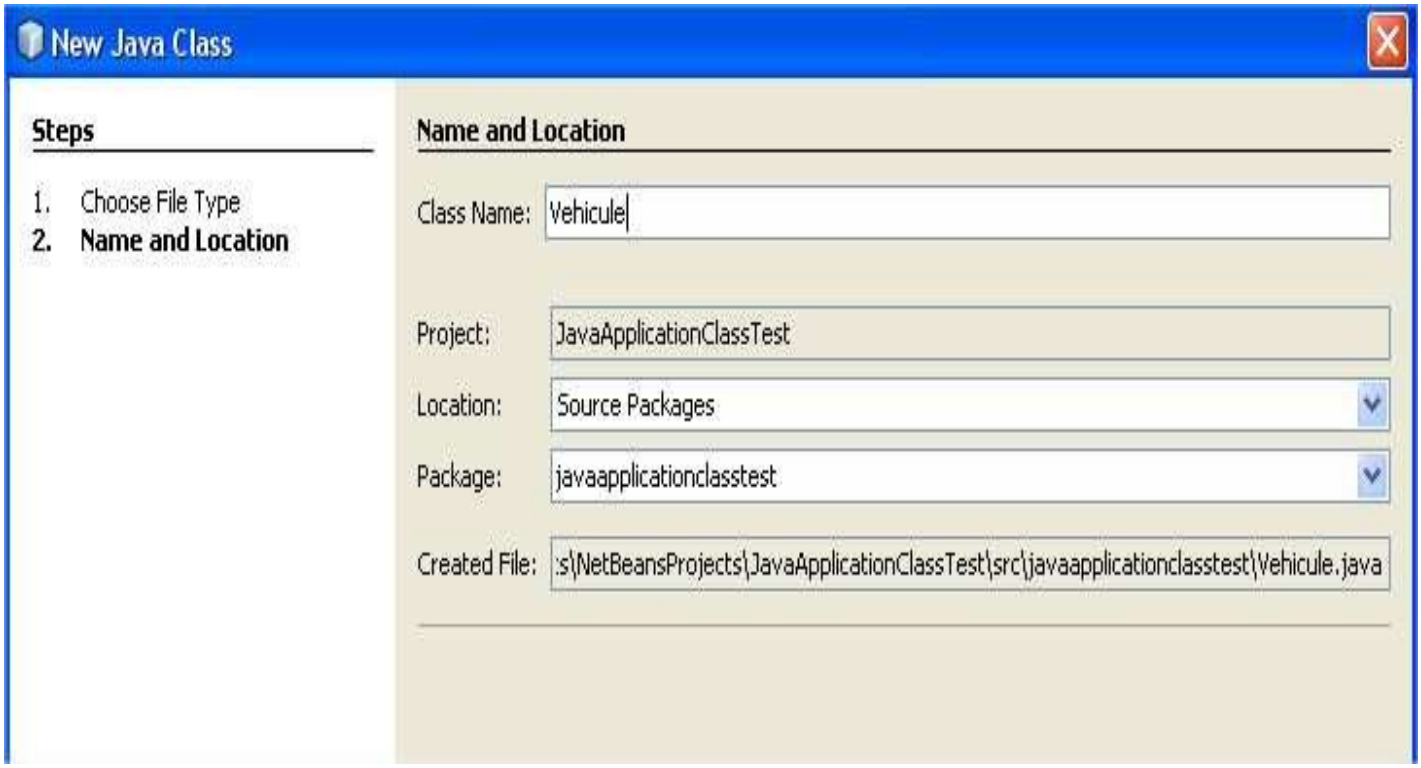
Donnez un nom à votre Application un nom.



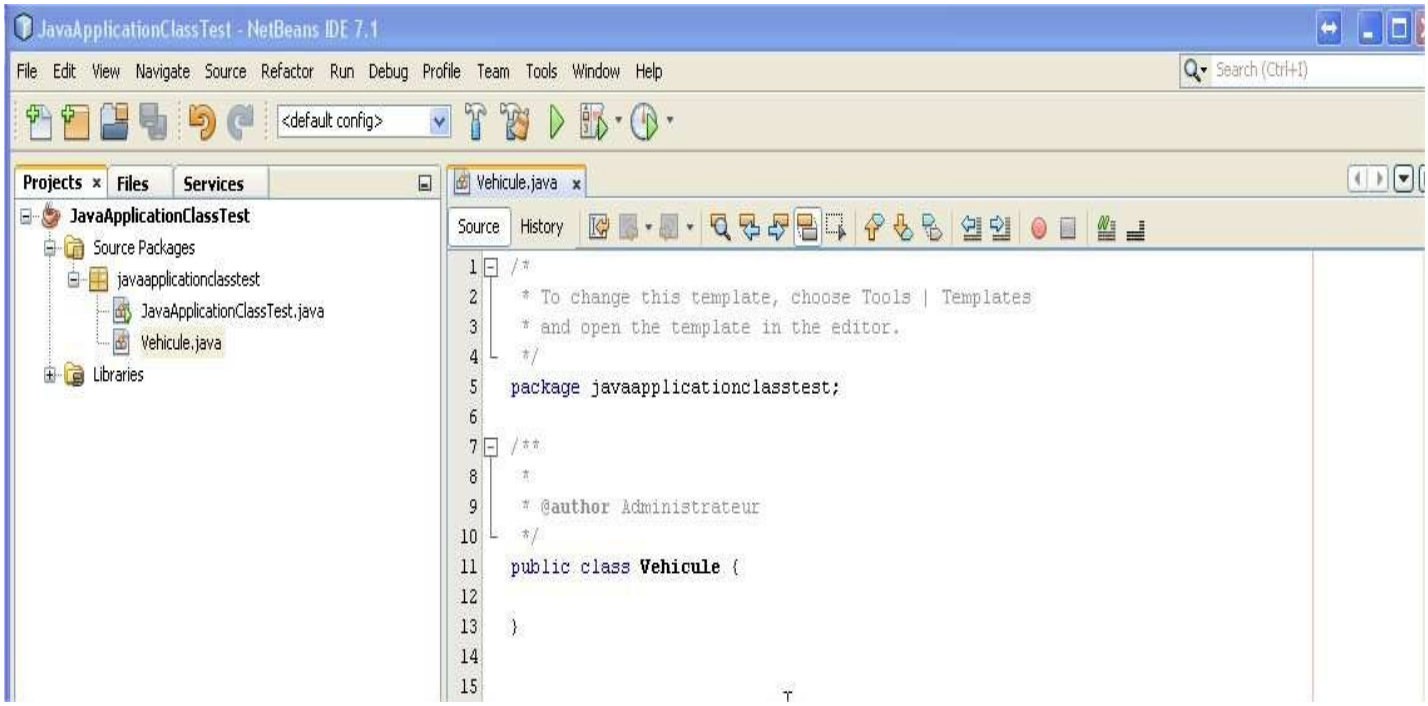
Faites un clic droit sur «javaapplicationclasstest» et Ajoutez une nouvelle classe.



Mettre le nom de votre classe, c'est-à-dire «Vehicule», et appuyer sur «Finish».



Ce qui donne le résultat suivant :



Ligne 5 : le package de notre nouvelle classe. Le package est une sorte de dossier qui va contenir notre classe. Les classes qui appartiennent au même package pourront se voir mutuellement, c'est-à-dire pas besoin d'utiliser le mot clef «import» pour voir les autres classes du même package. Complétons maintenant l'implémentation de la classe « Vehicule ».

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package javaapplicationclasstest;

/**
 *
 * @author Administrateur
 */
public class Vehicule {
    // attributs

    protected String marque;
    protected String couleur;
    protected int annee;
    protected double taille;
    protected boolean estAssure;

    // constructeurs
    public Vehicule() {
        System.out.println("Contructeur Vehicule sans argument");
    }

    public Vehicule(String marque, String couleur) {
        this.marque = marque;
        this.couleur = couleur;
        System.out.println("Contructeur Vehicule(string,string)");
    }

    public Vehicule(String marque, String couleur, int annee, double taille, boolean estAssure) {
        this.marque = marque;
        this.couleur = couleur;
        this.annee = annee;
        this.taille = taille;
        this.estAssure = estAssure;
        System.out.println("Contructeur Vehicule(string,string,int,double,bool)");
    }
    // methode descriptionVehicule

    public void descriptionVehicule() {
        System.out.println("Votre vehicule peut être décrit de la façon suivante : " + this.toString());
    }
    // Les accesseurs

    public String getMarque() {
        return marque;
    }

    public String getCouleur() {
        return couleur;
    }

    public int getAnnee() {
        return annee;
    }

    public double getTaille() {
        return taille;
    }

    public boolean getEstAssure() {
        return estAssure;
    }
    // les modificateurs

    public void setMarque(String marque) {
        this.marque = marque;
    }

    public void setCouleur(String couleur) {
        this.couleur = couleur;
    }

    public void setAnnee(int annee) {
        this.annee = annee;
    }

    public void setTaille(double taille) {
        this.taille = taille;
    }

    public void setEstAssure(boolean estAssure) {
        this.estAssure = estAssure;
    }

    @Override
    public String toString() {
        return String.format("[marque=%s,couleur=%s,annee=%s,taille=%s,estAssure=%s]", marque,couleur,annee,taille,estAssure);
    }
}
```

Ligne 14 à ligne 18 : déclaration des attributs de la classe. Ces attributs permettent de définir la nature de l'objet, exemple une voiture de marque Renault de couleur verte et fabriquée en 2012. Les attributs d'une classe peuvent être :
-Privé (private) : accessible que par les seules méthodes internes de la classe.
-Public (public): accessible par toute méthode définie ou non au sein de la classe.
-Protégé (protected) : accessible que par les seules méthodes internes de la classe ou d'un objet dérivé.
Ligne 21 à ligne 23, Ligne 25 à ligne 29 et Ligne 31 à ligne 38 : déclaration des constructeurs. Comme le nom l'indique un constructeur permet de fabrique des objets à travers l'initialisation des attributs de l'objet. Une classe peut avoir plusieurs constructeurs.
Ligne 21 à ligne 23 : constructeur sans argument de la classe.

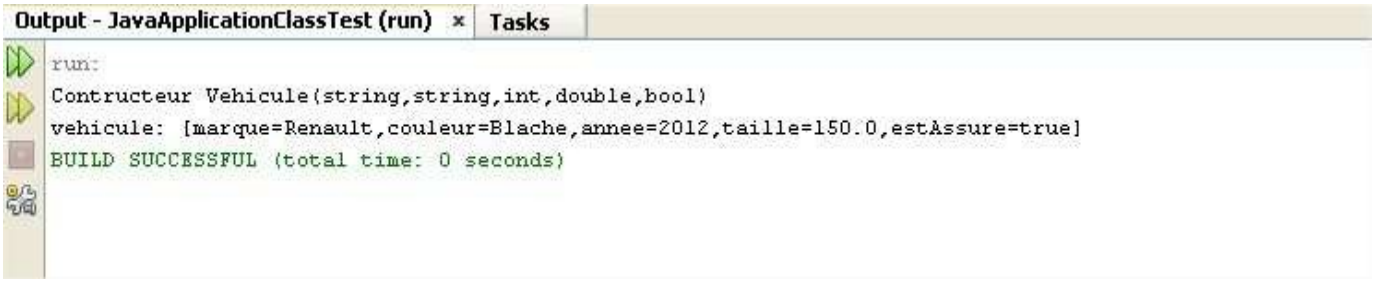
Ligne 41 à ligne 59 : déclaration des accesseurs. Les accesseurs sont des méthodes qui permettent d'accéder aux valeurs des attributs de l'objet.
Ligne 62 à ligne 80 : déclaration des modificateurs. Les modificateurs sont des méthodes qui permettent de modifier des attributs de l'objet.

Ligne 83 à ligne 85 : déclaration de la méthode « toString », cette méthode est très pratique car elle permet de savoir ce qui se trouve dans notre objet. Déclarer la méthode « toString » dans chacune de ses classes fait parti des bonnes pratiques pour un développeur.
Chaque Classe a, par défaut, une méthode « toString » qui lui est associée. Dans notre exemple, nous redéfinissons la méthode « toString » avec le mot clef « override » ce qui signifie qu'on écrase le contenu de la méthode par défaut par celle définie dans la classe « Vehicule ».
La méthode «toString » est définie par défaut dans la classe «Object » et toutes les classes héritent de la classe « Object ».

Exemple : Considérons le programme suivant :

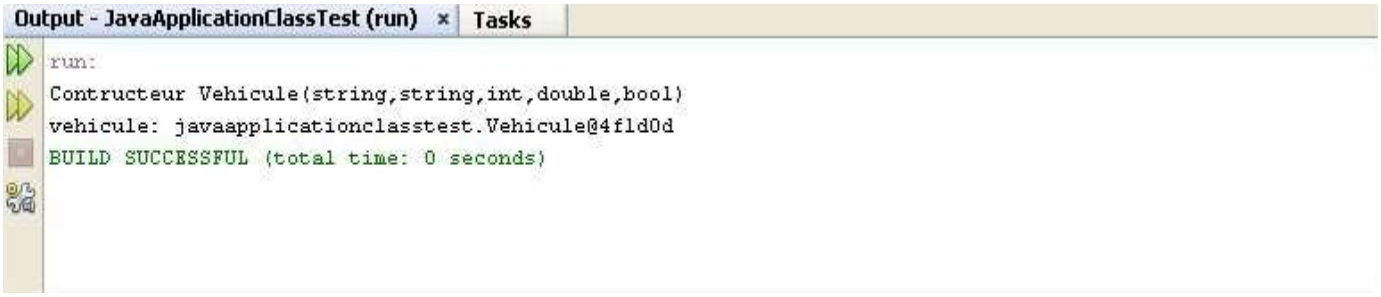
```
1  /*
2   * To change this template, choose Tools | Templates
3   * and open the template in the editor.
4   */
5  package javaapplicationclasstest;
6
7  /**
8   *
9   * @author Administrateur
10  */
11 public class JavaApplicationClassTest {
12
13     /**
14      * @param args the command line arguments
15      */
16     public static void main(String[] args) {
17         Vehicule vehicule = new Vehicule("Renault", "Blache", 2012, 150, true);
18         System.out.println("vehicule: " + vehicule);
19     }
20 }
```

Le programme affiche le résultat suivant :



L'écriture «System.out.println ("vehicule: " + vehicule.toString()) » est équivalent «System.out.println ("vehicule: " + vehicule) ». En effet System.out.println ("vehicule: " + vehicule) fait appelle implicitement à la méthode « toString » de la classe «Vehicule».

Remarque : Si la méthode « toString » n'était pas redéfinie, le programme aurait affiché :



Ce résultat ne donne aucune information sur le contenu de l'objet « vehicule » d'où la nécessité de redéfinir la méthode « toString ».

XVII) Notion d'héritage

L'héritage est une notion très pratique en programmation orientée objet. Il a pour but de spécialiser une classe fille par rapport à une classe mère. Par exemple, une voiture est un véhicule donc une voiture est une spécialisation de véhicule. De la même manière un train est une spécialisation de véhicule.

La dérivation se fait avec le mot clés «extends» de la façon suivante : class Fille extends Mère {}

Voici ci-dessous la signature de la classe « Voiture » qui hérite de la classe « Vehicule » ou bien on peut dire que la classe « Voiture » dérive de la classe « Vehicule ».

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package javaapplicationclasstest;

/**
 *
 * @author Administrateur
 */
public class Voiture extends Vehicule {

    private int longueurParchoque;

    public Voiture(String marque, String couleur, int annee, double taille, boolean estAssure, int longueurParchoque) {
        this.marque = marque;
        this.couleur = couleur;
        this.annee = annee;
        this.taille = taille;
        this.estAssure = estAssure;
        this.longueurParchoque = longueurParchoque;
        System.out.println("Contructeur Voiture(string,string,int,double,bool,int)");
    }
    // methode descriptionVehicule

    @Override
    public void descriptionVehicule() {
        System.out.println(String.format("Votre voiture peut être décrit de la façon suivante : [%s,longueurParchoque=%s]", toSt
    }

    public int getLongueurParchoque() {
        return longueurParchoque;
    }

    public void setLongueurParchoque(int longueurParchoque) {
        this.longueurParchoque = longueurParchoque;
    }
}
```

Une voiture a certaines particularités qui lui sont propres en tant que véhicule par exemple une voiture a un par choque ce qui veut dire qu'une voiture est un véhicule avec un par choque. En programmation orientée objet ceci se traduit par le fait qu'un objet « voiture » en plus d'avoir les mêmes attributs et méthodes qu'un objet «vehicule » a aussi un attribut pare choque en plus.

La méthode « descriptionVehicule () » est redéfinie dans cette nouvelle classe « Voiture », pour redéfinir une nouvelle méthode il suffit de déclarer:

@Override
public Type NomDeLaMethode {code à exécuter}

C'est le mot clef « Override» qui permet de concrétiser la redéfinition.

Le constructeur de la classe « Voiture » (voir ligne 15 à ligne 23de la classe « Voiture ») peut être réécrit d'une autre manière:

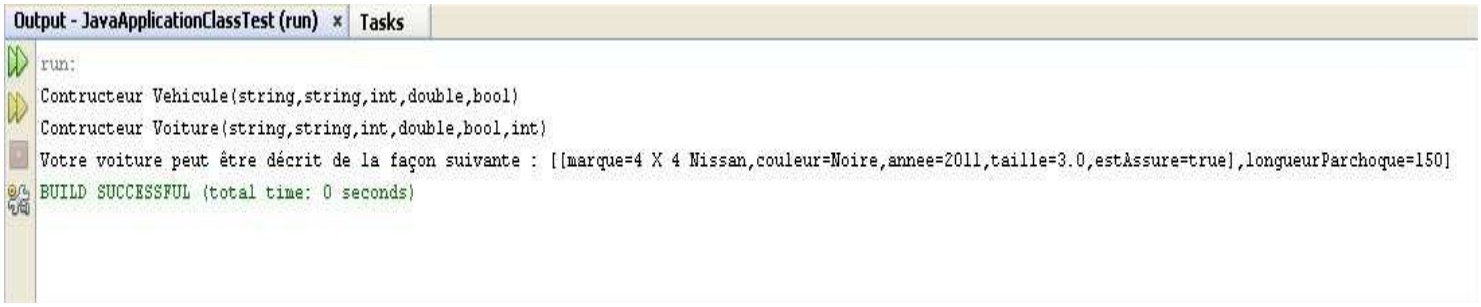
```
24
25      public Voiture(String marque, String couleur, int annee, double taille, boolean estAssure, int longueurParchoque) {
26          super( marque,  couleur,  annee,  taille,  estAssure);
27          this.longueurParchoque = longueurParchoque;
28          System.out.println("Contructeur Voiture(string,string,int,double,bool,int)");
29      }
```

Ligne 26 (super (marque, couleur, année, taille, estAssure)) : le mot clef « super » nous permet d'utiliser le constructeur de la classe mère « Vehicule (string, string, int, double, bool) » de « Vehicule ».

Regardons maintenant le programme ci-dessous :

```
1   /*
2    * To change this template, choose Tools | Templates
3    * and open the template in the editor.
4    */
5   package javaapplicationclasstest;
6
7   /**
8    *
9    * @author Administrateur
10   */
11  public class JavaApplicationClassTest {
12
13      /**
14       * @param args the command line arguments
15       */
16      public static void main(String[] args) {
17          Voiture voiture = new Voiture("4 X 4 Nissan", "Noire", 2011, 3, true, 150);
18          voiture.descriptionVehicule();
19      }
20  }
```

Voici le résultat lors de l'exécution du programme :



Nous pouvons pousser la dérivation (héritage) plus loin en spécialisant encore plus la classe en une voiture 4X4 à travers la classe « Voiture4X4 ».

```
12  public class Voiture4X4 extends Voiture {
13
14      private double puissanceMoteurArriere;
15
16      public Voiture4X4(String marque, String couleur, int annee, double taille, boolean estAssure, int longueurParchoque, double puissanceMoteurArriere) {
17          super(marque, couleur, annee, taille, estAssure, longueurParchoque);
18          this.puissanceMoteurArriere = puissanceMoteurArriere;
19          System.out.println("Contructeur Voiture4X4(string,string,int,double,bool,int,double)");
20      }
21
22      @Override
24      public void descriptionVehicule() {
25          System.out.println(String.format("Votre voiture 4X4 peut être décrit de la façon suivante : [%s,puissanceMoteurArriere=%s]", super.toString(), puissanceMoteurArriere));
26      }
27
28      public double getPuissanceMoteurArriere() {
29          return puissanceMoteurArriere;
30      }
31
32      public void setPuissanceMoteurArriere(double puissanceMoteurArriere) {
33          this.puissanceMoteurArriere = puissanceMoteurArriere;
34      }
35  }
```

XVIII) Le polymorphisme

Lorsqu'on regarde dans les dictionnaires le mot polymorphe on y trouve :
- Qui peut prendre des formes diverses.
- Qui offre des apparences, des formes diverses.

En programmation orientée objet les choses sont très similaires.

Considérons maintenant les classes suivantes:
Dans cet exemple, nous allons expliquer la notion de polymorphisme avec ces quatre classes : « Vehicule », « Voiture », « VoitureToutTerrain » et « VoitureToutTerrainCitadine ».

Considérons aussi :
La classe « VoitureToutTerrainCitadine » hérite de la classe « VoitureToutTerrain »
La classe « VoitureToutTerrain » hérite de la classe « Voiture »
La classe « Voiture » hérite de la classe « Vehicule »
« VoitureToutTerrainCitadine » □ « VoitureToutTerrain » □ « Voiture » □ « Vehicule » Dire que la classe « VoitureToutTerrainCitadine » hérite de la classe « VoitureToutTerrain » implique automatiquement que « VoitureToutTerrainCitadine » hérite de toutes les propriétés et méthode de « VoitureToutTerrain », donc un objet de type « VoitureToutTerrainCitadine » a les mêmes capacités qu'un objet de type « VoitureToutTerrain », d'une certaine manière un objet de type « VoitureToutTerrainCitadine » peut être objet de type « VoitureToutTerrain ».

On peut faire le même raisonnement entre « VoitureToutTerrain » et « Voiture » aussi entre « Voiture » et « Vehicule ». Ainsi un objet de type « VoitureToutTerrainCitadine » peut être un objet de type « VoitureToutTerrain », il peut aussi être un objet de type « Voiture » et enfin il peut aussi être un objet de type « Vehicule ». Cette capacité de l'objet à passer du type « VoitureToutTerrainCitadine » au type « VoitureToutTerrain », « Voiture » et « Vehicule » est appelée polymorphisme en référence au polymorphe.

Conséquence du polymorphisme

Considérons le programme suivant :

```
1  /*
2   * To change this template, choose Tools | Templates
3   * and open the template in the editor.
4   */
5  package javaapplicationclasstest;
6
7  /**
8   *
9   * @author Administrateur
10  */
11 public class JavaApplicationClassTest {
12
13     /**
14      * @param args the command line arguments
15      */
16     public static void main(String[] args) {
17         Vehicule vehicule = new Vehicule("Vehicule de type train", "gris", 2000, 300, true);
18         description(vehicule);
19
20         Voiture voiture = new Voiture("4 X 4 Nissan", "Noire", 2011, 3, true, 150);
21         description(voiture);
22     }
23
24     public static void description(Vehicule vehicule) {
25         vehicule.descriptionVehicule();
26     }
27 }
```

Ligne 24 « public static void description (Vehicule vehicule) » : on voit la première conséquence du polymorphisme, en effet la méthode « description » peut prendre comme paramètre un objet de type « Vehicule » mais aussi tous les objets des classes dérivées de « Vehicule » (les objets de classe « Voiture »).

Le programme affiche le résultat suivant :

```
Output - JavaApplicationClassTest (run)
run:
Contructeur Vehicule(string,string,int,double,bool)
Votre vehicule peut être décrit de la façon suivante : [marque=Vehicule de type train,couleur=gris,annee=2000,taille=300.0,estAssure=true]
Contructeur Vehicule(string,string,int,double,bool)
Contructeur Voiture(string,string,int,double,bool,int)
Votre voiture peut être décrit de la façon suivante : [[marque=4 X 4 Nissan,couleur=Noire,annee=2011,taille=3.0,estAssure=true],longueurParchoque=150]
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ce résultat nous montre que le programme a exécuté la méthode « descriptionVehicule() » de la classe « Vehicule ».En effet, ligne 18 de la classe «JavaApplicationClassTest» (description(vehicule)) exécute la méthode «Vehicule.descritionVehicule» et ligne 21 (description(voiture)) exécute la méthode «Voiture.descritionVehicule».

XIX) Classe abstraite

Une classe abstraite est une classe dont l'implémentation n'est pas complète et qu'on ne peut instancier. Par contre, les classes dérivées d'une classe abstraite peuvent être instanciées.

Considérons l'exemple de la classe abstraite suivante :

```
1  /*
2   * To change this template, choose Tools | Templates
3   * and open the template in the editor.
4   */
5  package javaapplicationclasstest;
6
7  /**
8   *
9   * @author Administrateur
10  */
11 public abstract class Felin {
12     // attributs
13
14     protected String nom;
15     protected String couleur;
16
17     public Felin(String nom, String couleur) {
18         this.nom = nom;
19         this.couleur = couleur;
20     }
21
22     abstract public void crie();
23 }
```

Voici un exemple d'une classe « Tigre » qui dérive de la classe « Felin » :

```
1  /*
2   * To change this template, choose Tools | Templates
3   * and open the template in the editor.
4   */
5  package javaapplicationclasstest;
6
7  /**
8   *
9   * @author Administrateur
10  */
11 public class Tigre extends Felin {
12     // constructeur
13
14     public Tigre(String nom, String couleur) {
15         super(nom, couleur);
16     }
17
18     @Override
19     public void crie() {
20         System.out.println("Le Tigre grogne.");
21     }
22 }
```

XX) Les interfaces

Une interface est un ensemble de prototypes de méthodes ou de propriétés formant une entité. Une classe implémentant une interface doit obligatoirement fournir une implémentation de toutes les méthodes de l'interface. Une interface est très semblable à une classe abstraite.

Voici l'exemple d'une interface « IAnimal » :

```
1  /*
2   * To change this template, choose Tools | Templates
3   * and open the template in the editor.
4   */
5  package javaapplicationclasstest;
6
7  /**
8   *
9   * @author Administrateur
10  */
11 public interface IAnimal {
12
13     void crie();
14
15     void marcher();
16 }
```

Voici la classe « Animal » qui implémente l'interface « IAnimal »

```
1  /*
2   * To change this template, choose Tools | Templates
3   * and open the template in the editor.
4   */
5  package javaapplicationclasstest;
6
7  /**
8   *
9   * @author Administrateur
10  */
11 public class Animal implements IAnimal {
12
13     @Override
14     public void crie() {
15         System.out.println("L'animal crie");
16     }
17
18     @Override
19     public void marcher() {
20         System.out.println("L'animal marche");
21     }
22 }
```

Ligne 11 : « class Animal implements IAnimal » implique que la classe « Animal » implémente l'interface « IAnimal ».

XXI) Objets très utiles en Java

Lecture des fichiers texte

La classe « BufferedReader» permet de lire le contenu d'un fichier texte. Cette classe appartient à l'espace de nom «java.io.BufferedReader».

Cette classe dispose d'un constructeur qui prend en paramètre le chemin du fichier texte à lire :
public BufferedReader(Reader in)

«BufferedReader» dispose d'un certain nombre de méthode, parmi elle on peut citer :

public int read() : Cette méthode renvoie le caractère suivant du flux et avance d'un caractère dans le flux.

public void close () : Cette méthode ferme le flux de texte et libère toutes les ressources allouées pour la lecture du fichier. Il est indispensable de fermer une ressource après son utilisation sinon on risque d'avoir des fuites mémoires dans l'application.

public override int read (char[] buffer, int index, int count) : Cette méthode lit dans le flux count caractères à partir de la position index et les met dans un buffer. Cette méthode revoit le nombre de caractères lus qui peut prendre dans certains cas la valeur 0.

public String readLine() : Cette méthode renvoie la ligne suivante du flux. Quand on arrive à la fin flux alors « ReadLine() » revoie la valeur null.

Ecriture des fichiers texte

La classe « BufferedWriter» permet d'écrire dans un fichier texte. Cette classe appartient à l'espace de nom « java.io..BufferedWriter».

Cette classe dispose d'un constructeur qui prend en paramètre le chemin du fichier texte à lire :
public BufferedWriter(Writer out)

«BufferedWriter» dispose de propriétés incontournables comme:

private Writer out

Parmi les méthodes « BufferedWriter» dispose d'un certain nombre de méthodes, parmi-elles on peut citer :

public String newLine()

Fixe les caractères de fin de ligne. La fin de ligne est marquée par un "\r\n" par défaut ("\r\n" pour un system Windows et "\n" pour un système Unix).

public void close() : Cette méthode ferme le flux d'écriture et libère toutes les ressources allouées pour l'écriture du fichier. Il est indispensable de fermer une ressource après son utilisation sinon on risque d'avoir des fuites mémoires dans l'application.

public void flush () : Cette méthode force le vidage de la mémoire tampon sur le fichier à écrire.

public void write (String value) : Cette méthode écrit value dans le fichier associé.

Les chaînes de caractères.

La classe « String » appartient au package «java.lang». Cette classe présente de nombreuses propriétés et méthodes :

public string trim(): cette méthode enlève les espaces de début et de fin de la chaine courante.

public int length () : cette méthode retourne nombre de caractères de la chaîne.

public bool endsWith (String value) : cette méthode retourne « true » si la chaîne se termine par value.
public bool startsWith(String value) : cette méthode retourne « true » si la chaîne commence par value.

public bool equals(object ob): cette méthode retourne « true » si la chaîne courante est égale à ob.

public int indexOf (String value, int startIndex) : cette méthode retourne la première position dans la chaîne « value » à partir « startIndex ».

public int lastIndexOf (String value, int startIndex): cette méthode retourne la dernière position dans la chaîne « value » à partir « startIndex ».

public String toLowerCase () : cette méthode transforme la chaîne courante en minuscules.

public String toUpperCase () : cette méthode transforme la chaîne courante en majuscules.

La classe ArrayList et interface «java.util.List»

Cette classe appartient au package «java.util».

Parmi les méthodes de la classe on peut citer :

public int size() : cette méthode retourne la taille de notre liste.

public void add (Object element) : cette méthode ajoute l'élément «element» à la liste.

public void get(int index) : retourne l'élément situé à la position index.

public void remove(int index) : retire l'élément situé à la position index.

public boolean isEmpty() : renvoie «true» si l'objet est vide.

public void removeAll() : efface tous les éléments contenu dans notre liste.

public boolean contains(Object element) : retourne « true » si l'élément «element» est dans la liste.

La classe «java.util.HashMap» et interface «java.util.Map»

Parmi les méthodes on peut citer :

public int size () : cette méthode retourne la taille du Map.

public V put (K key, V value) : cette méthode permet d'ajouter le couple «key» et «value».

public V get(Object key) : retourne l'élément dont la clef vaut key.

public V remove(Object key) : retire l'élément dont la clef vaut key.

public boolean isEmpty() : renvoie «true» si l'objet est vide.

public boolean containsKey (Object key) : retourne « true » si la clef «key» existe dans le Map.

public boolean containsValue (Object value) : retourne « true » si la valeur «value» existe dans le Map.

La classe «java.util.HashSet» et interface «java.util.Set»

Parmi les méthodes de la classe on peut citer :

public int size() : cette méthode retourne la taille de notre liste.

public void add (Object element) : cette méthode ajoute l'élément «element» à la liste.

public void get(int index) : retourne l'élément situé à la position index.

public void remove(int index) : retire l'élément situé à la position index.

public boolean isEmpty() : renvoie «true» si l'objet est vide.

public void removeAll() : efface tous les éléments contenu dans notre liste.

public boolean contains(Object element) : retourne « true » si l'élément «element» est dans la liste.

XXII) Gestion des dates

Les dates sont très importantes en Java, elles permettent d'affecter un aspect temporel aux évènements. Par exemple, dans une application on peut décider de stocker dans une variable ou objet la date de naissance d'une personne.

Pour transformer une chaine de caractères en date, il suffit de faire :

Pour un format dd/MM/yyyy nous aurons :

Pour un format dd-MM-yyyy nous aurons :

On peut transformer une date en « Timestamp ». Le type « Timestamp » est très important en java, il est utilisé par tous les SGBD (MySQL, Oracle, DB2) donc il est important de savoir comment passer du type « Date » au type « Timestamp ».

```
40
41          try {
42              String dateNaissanceString = "25/08/2008";
                SimpleDateFormat formatterDMY = new SimpleDateFormat("dd/MM/yyyy");
44              Date dateNaissance = formatterDMY.parse(dateNaissanceString);
45
46              System.out.println("Format dd/MM/yyyy dateNaissance: " + formatterDMY.format(dateNaissance));
47
48              Timestamp timestampDateNaissance = new Timestamp(dateNaissance.getTime());
49
50              System.out.println("timestampDateNaissance: " + timestampDateNaissance.toString());
51
52          } catch (Exception e) {
53              System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + e.getMessage());
54          }
```

javaapplicationstatistiqueforet.JavaApplicationStatistiqueForet > main > try > formatterDMY >

Output X

Debugger Console X | JavaApplicationStatistiqueForet (run) X

```
run:
Format dd/MM/yyyy dateNaissance: 25/08/2008
timestampDateNaissance: 2008-08-25 00:00:00.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

Passage du type « Date » vers « Timestamp »

```
40
41          try {
42
43              Timestamp timestamp = new Timestamp(System.currentTimeMillis());
44              Date date = new Date(timestamp.getTime());
45              System.out.println("date: " + date);
46          } catch (Exception e) {
47              System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + e.getMessage());
48          }
```

javaapplicationstatistiqueforet.JavaApplicationStatistiqueForet >

Output X

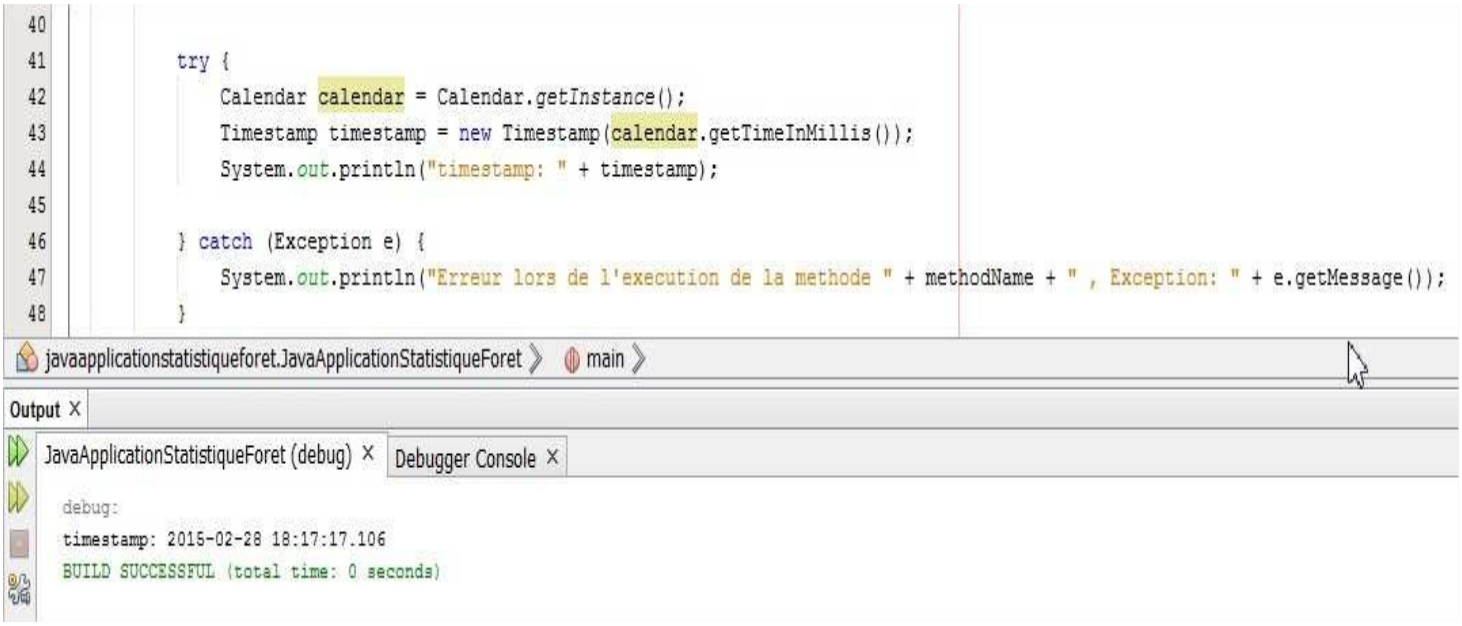
JavaApplicationStatistiqueForet (debug) X | Debugger Console X

```
debug:
date: Sat Feb 28 17:34:02 CET 2015
BUILD SUCCESSFUL (total time: 0 seconds)
```

On peut aussi accéder à la date actuelle :

Par l'intermédiaire de « System.currentTimeMillis() » :

Par l'intermédiaire de « java.util.Calendar » :



XXIII) Quelques designs patterns

Un Design Pattern est une solution à un problème récurrent dans la conception d'applications orientées objet. Un patron de conception décrit alors la solution éprouvée pour résoudre ce problème d'architecture de logiciel. L'utilisation des Design Patterns offre de nombreux avantages. Tout d'abord cela permet de répondre à un problème de conception grâce à une solution éprouvée et validée par des experts. Ainsi on gagne en rapidité et en qualité de conception ce qui diminue également les coûts.
De plus, les Design Patterns sont réutilisables et permettent de mettre en avant les bonnes pratiques de conception.

Les Design Patterns sont représentés par :

- **Nom** : qui permet de l'identifier clairement
- **Problématique** : description du problème auquel il répond
- **Solution** : description de la solution souvent accompagnée d'un schéma UML
- **Conséquences** : les avantages et les inconvénients de cette solution

Les patrons de conception sont classés en trois catégories :

- **Création** : ils permettent d'instancier et de configurer des classes et des objets.
- **Structure** : ils permettent d'organiser les classes d'une application.
- **Comportement** : ils permettent d'organiser les objets pour qu'ils collaborent entre eux.

1) Design pattern Fabrique (Factory)

Ce design pattern est très utile lorsqu'on veut concevoir une classe qui va instancier différents types d'objets suivant un paramètre fourni. Par exemple, une usine va fabriquer des produits en fonction du modèle qu'on lui indique.
Exemple : FabriqueAnimal, Tigre, Mouton, Animal
Soit la classe « FabriqueAnimal » dont la signature est la suivante :

```
1   package javaapplicationdesignpattern;
2
3   public class FabriqueAnimal {
4
5       public Animal getAnimal(int typeAnimal) {
6           Animal animal = null;
7           switch (typeAnimal) {
8               case 0:
9                   animal = new Tigre();
10                  break;
11              case 1:
12                  animal = new Mouton();
13                  break;
14              case 2:
15                  animal = new Chat();
16                  break;
17          }
18          return animal;
19      }
20  }
```

En version copiable :

```
package javaapplicationdesignpattern;

public class FabriqueAnimal {

  public Animal getAnimal(int typeAnimal) {
    Animal animal = null;
    switch (typeAnimal) {
      case 0:
        animal = new Tigre();
        break;
      case 1:
        animal = new Mouton();
        break;
      case 2:
        animal = new Chat();
        break;
    }
    return animal;
  }
}
```

Ligne 5 : Un entier passé en parameter permet de passer d'un animal à un autre. En effet, si typeAnimal est égale à zéro alors on retournera un animal de type Tigre, si typeAnimal est égale à un alors on retournera un animal de type Mouton et ainsi de suite. Mais il est impératif que « Tigre », « Mouton » et « Chat » hérite d'animal pour que cela marche.

La classe « Animal » aura la signature suivante :

```
package javaapplicationdesignpattern;

public class Animal {

  public void crie() {
    System.out.println("L'animal crie");
  }
}
```

La classe « Tigre » aura la signature suivante :

```
package javaapplicationdesignpattern;

public class Tigre extends Animal {

   public Tigre() {
   }

   @Override
   public void crie() {
      System.out.println("Le Tigre grogne.");
   }
}
```

L'annotation avec mot-clé « Override » permet d'écraser la définition de la méthode crie qui a été héritée de la classe mère « Animal ».

La classe « Mouton » aura la signature suivante :

```
package javaapplicationdesignpattern;

public class Mouton extends Animal {

   public Mouton() {
   }

   @Override
   public void crie() {
      System.out.println("Le Mouton bêle.");
   }
}
```

La classe « Chat » aura la signature suivante :

```
package javaapplicationdesignpattern;

public class Chat extends Animal {

   public Chat() {
   }

   @Override
   public void crie() {
      System.out.println("Le Chat miaule.");
   }
}
```

2) Design pattern Singleton

Problématique : certaines applications possèdent des classes qui doivent être instanciées une seule et unique fois. Exemple : le cas d'une classe qui implémenterait un pilote pour un périphérique.
Un Singleton garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance.

Son implémentation peut être la suivante :

```
package javaapplicationdesignpattern;

public class MySingleton {

    /**
     * Constructeur privé
     */
    private MySingleton() {
    }

    /**
     * Instance unique non préinitialisée
     */
    private static MySingleton INSTANCE = null;

    /**
     * Point d'accès pour l'instance unique du singleton
     *
     * @return
     */
    public static synchronized MySingleton getInstance() {
        if (INSTANCE == null) {
            INSTANCE = new MySingleton();
        }
        return INSTANCE;
    }
}
```

Le constructeur privé empêche de créer des instances de MySingleton à l'extérieur de la classe.
Le mot-clé synchronized permet de mutualiser les instances de MySingleton dans un environnement multi-Threads. En effet, sans ce mot-clé, il y a une possibilité d'avoir une instance de MySingleton par Thread ce qui serait catastrophique dans la mesure où on veut avoir qu'une seule et unique instance de MySingleton dans la JVM (Machine virtuelle de Java).

3) Design pattern DAO

Le pattern DAO (Data Access Object) est un patron de conception qui propose de lier les objets en mémoire vive aux données persistantes (stockées en base de données, dans des fichiers, dans des annuaires, etc.). Ainsi on centralise les mécanismes de mapping entre le système de stockage (SGBD) et les objets images de Java. Ce pattern a pour intérêt de prévenir un changement éventuel de système de stockage de données (de fichiers vers une base de données, de la SGBD MySQL vers la SGBD Oracle etc…..).

Voici l'implémentation générique d'une classe DAO générique pour une entité T. Cette classe sert à mutualiser les méthodes find,create,update,delete qui seront communes à toutes les entités de notre application. Par exemple, dans une application de gestion d'employés , nous pouvons avoir comme entité Adresse correspondant à l'adresse de l'employé, Role correspondant au role de l'employé dans la société et l'entité Employe qui sera elle-même composée de certains attributs spécifiques plus un attribut de type Adresse et un attribut de type Role.

```
package javaapplicationdesignpattern;

public abstract class Dao<T> {
   /**
    * Permet de récupérer un objet via son ID
    *
    * @param id
    * @return
    */
   public abstract T find(int id);

   /**
    * Permet de créer une entrée dans la datasource par rapport à un objet
    *
    * @param obj
    */
   public abstract T create(T obj);

   /**
    * Permet de mettre à jour les données d'une entrée dans la datasource
    *
    * @param obj
    */
   public abstract T update(T obj);

   /**
    * Permet la suppression d'une entrée de la datasource
    *
    * @param obj
    */
   public abstract void delete(T obj);
}
```

Si on applique la classe Dao à une entité « Personne » alors nous aurons :

```
package javaapplicationdesignpattern;

public class PersonneDao extends Dao<Personne> {

  @Override
  public Personne find(int id) {
    // Implementer la methode pour trouver la personne qui a l'identifiant id
    // et retourner la personne concernée.
  }

  @Override
  public Personne create(Personne obj) {
    // Implementer la methode pour creer une personne.
    // et retourner la personne crée.
  }

  @Override
  public Personne update(Personne obj) {
    // Implementer la methode pour mettre à jour la personne obj.
    // et retourner la personne mise à jour.
  }

  @Override
  public boolean delete(Personne obj) {
    // Implementer la methode pour supprimer la personne obj.
    // et retourner true pour dire que cela s'est bien passé.
  }
}
```

Les implementions de find,create,update,delete vont dependre de la source de données (dataSource) qui peut être un ou des objets java, un ou des fichiers textes, un ou des fichiers xml, ou un ou des fichiers Json ou une base de données.

4) Design pattern MVC

Ce patron d'architecture logicielle signifie Modèle Vue Contrôleur. Ce dernier peut être découpé en trois parties :

- Le modèle (modèle de données) : le modèle représente le cœur (algorithmique) de l'application : traitements des données, interactions avec la base de données, etc. Il décrit les données manipulées par l'application. Il regroupe la gestion de ces données et est responsable de leur intégrité. La base de données sera l'un de ses composants. Le modèle comporte des méthodes standards pour mettre à jour ces données (insertion, suppression, changement de valeur).

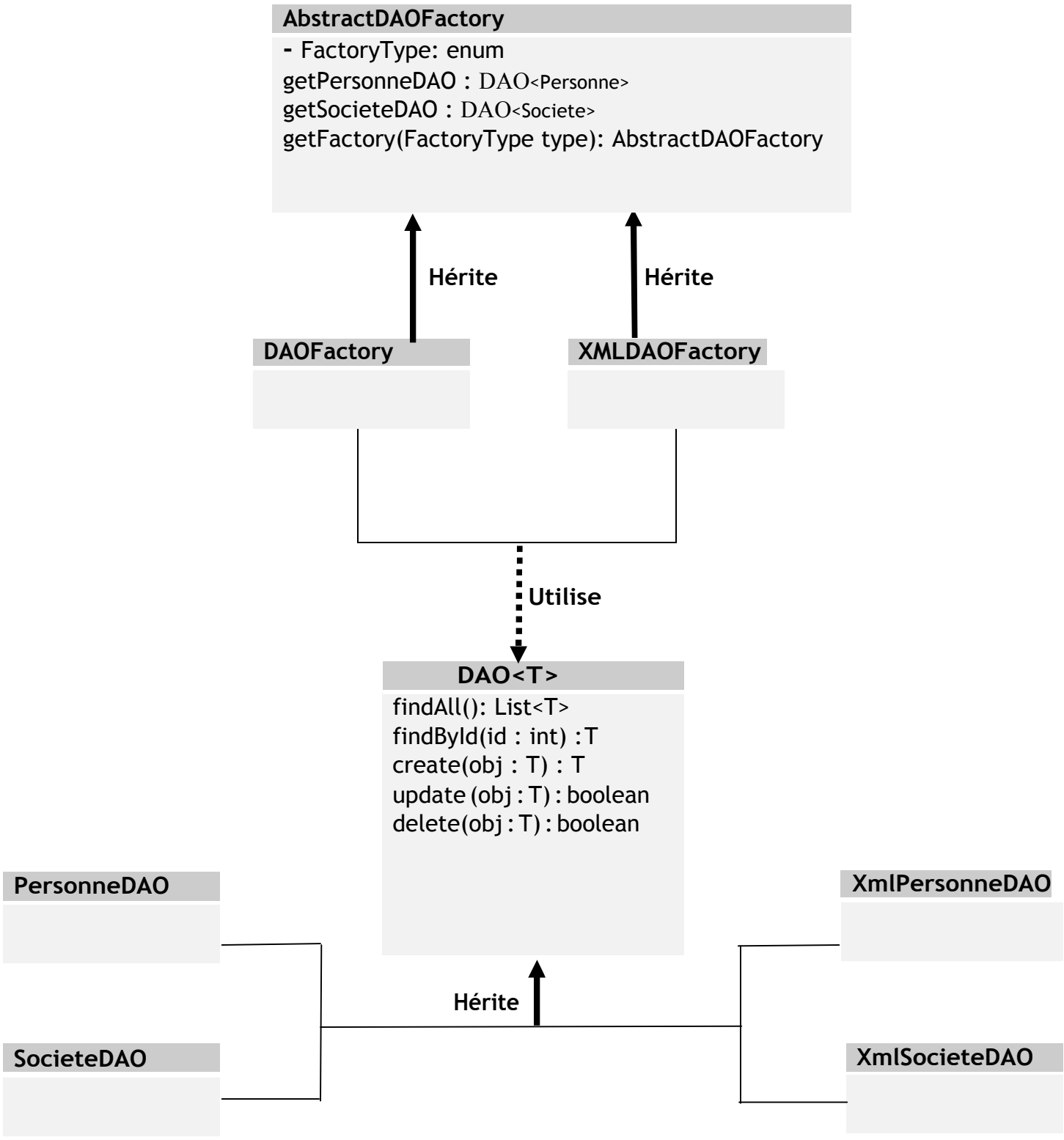
- La vue (présentation, interface utilisateur) : ce avec quoi l'utilisateur interagit se nomme précisément la vue. Sa première tâche est de présenter les résultats renvoyés par le modèle. Sa seconde tâche est de recevoir toute action de l'utilisateur (hover, clic de souris, sélection d'un bouton radio, cochage d'une case, entrée de texte, de mouvements, de voix, etc.). Ces différents événements sont envoyés au contrôleur. La vue n'effectue pas de traitement, elle se contente d'afficher les résultats des traitements effectués par le modèle et d'interagir avec l'utilisateur.

- Le contrôleur (logique de contrôle, gestion des événements, synchronisation) : le contrôleur prend en charge la gestion des événements de synchronisation pour mettre à jour la vue ou le modèle et les synchroniser. Il reçoit tous les événements de la vue et enclenche les actions à effectuer. Si une action nécessite un changement des données, le contrôleur demande la modification des données au modèle afin que les données affichées se mettent à jour. D'après le patron de conception observateur/observable, la vue est un « observateur » du modèle qui est lui « observable».

XXIV) Exemple d'application des Designs Pattern Dao et Factory

Problématique : on veut implémenter une solution qui permet de changer de source de données sans trop modifier le code. Soit une société X dans laquelle travaillent des personnes. Les entités de cette relation sont « **Personne** » et « **Societe** ».
La classe DAO<T> (où T représente la classe générique) est une classe abstraite avec les méthodes findById (int id) retournant une entité de type « T », create (T obj) retournant une entité de type « T », update (T obj) retournant un booléen, delete (T obj) retournant un booléen.
Les classes PersonneDAO, SocieteDAO, XmlPersonneDAO et XmlSocieteDAO sont des classes no abstraites (c'est-à-dire avec des implémentations complètes) qui héritent toutes de « DAO<T> ».

Les classes « DAOFactory » et « XMLDAOFactory » hérite de la classe abstraite « AbstractDAOFactory » qui elle est une Fabrique ou Factory qui va renvoyer le bon DAO (DAO normale ou XMLDAO etc. …) pour chaque entité « Personne » ou « Societe » à travers PersonneDAO, SocieteDAO, XmlPersonneDAO ou XmlSocieteDAO.


AbstractDAOFactory
- FactoryType: enum
getPersonneDAO : DAO<Personne>
getSocieteDAO : DAO<Societe>
getFactory(FactoryType type): AbstractDAOFactory
Hérite
Hérite
DAOFactory
XMLDAOFactory
Utilise
DAO<T>
findAll(): List<T>
findById(id : int) :T
create(obj : T) : T
update (obj : T) : boolean
delete(obj : T) : boolean
PersonneDAO
XmlPersonneDAO
Hérite
SocieteDAO
XmlSocieteDAO

XXV) Réflexivité en Java

La réflexivité ou encore l'introspection consiste à découvrir de manière dynamique des informations propres à une classe Java ou à un objet. Ce mécanisme est notamment utilisé au niveau de la machine virtuelle Java lors de l'exécution de votre programme.

Le paquetage de l'API qui gère la réflexivité est « java.lang.reflect ».
La réflexivité permet notamment l'introspection et rend possible l'accès aux classes, à leurs champs, méthodes, constructeurs et à toutes les informations les caractérisant, même celles qu'on pensait inaccessibles. Elle est également très utile pour instancier des classes de manière dynamique, dans le processus de sérialisation d'un Bean Java. Elle est aussi utilisée dans la génération de code (Exemple ORM tel que Hibernate).

Avec la réflexivité il est possible :

1) D'identifier les classes parents d'une classe donnée

C'est-à-dire, connaître l'ensemble des classes dont hérite une classe. Considérons l'exemple précédemment de la classe « Vehicule » cf. « Les Objets Java » nous avions la classe « Voiture » qui hérite de la classe « Vehicule » et une classe « Voiture4X4 » qui hérite de « Voiture ».

Affichons dans le programme ci-dessous tous les parents de la classe « Voiture4X4 » :

```
24
25      public static void displaySuperclass() {
26          String methodName = "displaySuperclass";
27          Class theClass = null;
28          try {
29              theClass = Class.forName("javaapplicationreflect.Voiture4X4");
30          } catch (ClassNotFoundException ex) {
31              System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + ex);
32          }
            while ((theClass = theClass.getSuperclass()) != null) {
34              System.out.println("Nom de la classe Mère: " + theClass.getSimpleName());
35              System.out.println("Nom Complet de la classe Mère: " + theClass.getName()+"\n");
36          }
37      }
```

On obtient donc à l'affichage :

```
Output - JavaApplicationReflect (run) | Search Results | Test Results
run:
Nom de la classe Mère: Voiture
Nom Complet de la classe Mère: javaapplicationreflect.Voiture

Nom de la classe Mère: Vehicule
Nom Complet de la classe Mère: javaapplicationreflect.Vehicule

Nom de la classe Mère: Object
Nom Complet de la classe Mère: java.lang.Object

BUILD SUCCESSFUL (total time: 0 seconds)
```

L'affichage nous montre bien que la classe « Voiture4X4 » hérite des classes « Voiture », « Vehicule » et bien entendu la classe « Object » qui est par définition la mère de toute les classes Java.

2) Connaitre toutes les interfaces implémentées par une classe donnée

Considérons l'exemple d'une interface « IAnimale » avec sa classe d'implémentation « Animal ».
« Animal » :

```
13  public class Animal implements IAnimal, Serializable {
14
15      public String nom;
16      public double poids;
17      public String couleur;
18
19      @Override
20      public void crie() {
21          System.out.println("L'animal crie");
22      }
23
24      @Override
25      public void marcher() {
26          System.out.println("L'animal marche");
27      }
28
29      public String getNom() {
30          return nom;
31      }
32
33      public void setNom(String nom) {
34          this.nom = nom;
35      }
36
37      public double getPoids() {
38          return poids;
39      }
40
41      public void setPoids(double poids) {
42          this.poids = poids;
43      }
44
45      public String getCouleur() {
46          return couleur;
47      }
48
49      public void setCouleur(String couleur) {
50          this.couleur = couleur;
51      }
52  }
53
```

« IAnimale » :

```
10  */
11  public interface IAnimal {
12
13      void crie();
14
15      void marcher();
16  }
17
18
```

Considérons le programme Java ci-dessous :

```
39
40      public static void displayInterfaces() {
41          String methodName = "displayInterfaces";
            Class theClass = null;
43          try {
44              theClass = Class.forName("javaapplicationreflect.Animal");
45              Class[] interfaces = theClass.getInterfaces();
46              for (Class theInterface : interfaces) {
47                  System.out.println("Nom de l'interface : " + theInterface.getSimpleName());
48                  System.out.println("Nom complet de l'interface : " + theInterface.getName() + "\n");
49              }
50          } catch (ClassNotFoundException ex) {
51              System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + ex);
52          }
53      }
54
```

On obtient à l'affichage:

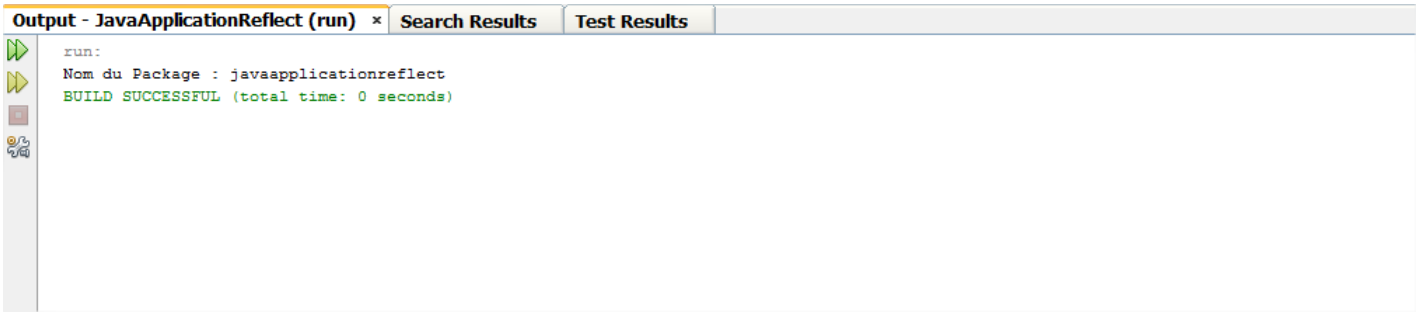
```
Output - JavaApplicationReflect (run) × | Search Results | Test Results
run:
Nom de l'interface : IAnimal
Nom complet de l'interface : javaapplicationreflect.IAnimal

Nom de l'interface : Serializable
Nom complet de l'interface : java.io.Serializable

BUILD SUCCESSFUL (total time: 0 seconds)
```

3) Connaitre le nom du paquetage d'une classe donnée

Toujours avec la classe « Animal » nous pouvons récupérer son nom de paquetage.

```
61
62      public static void displayPackage() {
63          String methodName = "displayPackage";
            Class theClass = null;
65          try {
66              theClass = Class.forName("javaapplicationreflect.Animal");
67              Package pack = theClass.getPackage();
68              System.out.println("Nom du Package : " + pack.getName());
69          } catch (ClassNotFoundException ex) {
70              System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + ex);
71          }
72      }
73
```

On obtient à l'affichage:



4) Récupérer la liste des méthodes d'une classe.

Nous pouvons récupérer la liste des méthodes de la classe la classe « Voiture4X4 ».

```
76
77      public static void displayMethods() {
78          String methodName = "displayMethods";
            Class theClass = null;
80          try {
81              theClass = Class.forName("javaapplicationreflect.Voiture4X4");
82              Method[] methodes = theClass.getMethods();
83              for (Method methode : methodes) {
84                  System.out.println("Nom de methode : " + methode.getName());
85              }
86
87              Class[] argument = new Class[]{double.class};
88
89              Method method = theClass.getMethod("setPuissanceMoteurArriere", argument);
90              System.out.println("Signature methode setPuissanceMoteurArriere : " + method);
91
92          } catch (ClassNotFoundException | SecurityException | NoSuchMethodException ex) {
93              System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + ex);
94          }
95      }
```

Ligne 83 : On récupère la liste des méthodes de la classe « Voiture4X4 ».
Ligne 89 : On récupère la signature de la méthode « setPuissanceMoteurArriere » en lui passant comme argument « new Class [] {double.class} » car, en effet, la méthode « setPuissanceMoteurArriere » a comme signature dans la classe « Voiture4X4 » :

```
public void setPuissanceMoteurArriere(double puissanceMoteurArriere) {
    this.puissanceMoteurArriere = puissanceMoteurArriere;
}
```

5) Récupérer la liste des attributs d'une classe.

Il est aussi possible de récupérer la liste des attributs d'une classe.

```java
public static void displayAllFields() {
    String methodName = "displayAllFields";
    Class theClass = null;
    try {
        theClass = Class.forName("javaapplicationreflect.Vehicule");
        java.lang.reflect.Field[] fields = theClass.getDeclaredFields();
        for (Field field : fields) {
            System.out.println("Nom du champ : " + field);
        }
    } catch (ClassNotFoundException | SecurityException ex) {
        System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + ex);
    }
}
```

```
Output - JavaApplicationReflect (run) | Search Results | Test Results
run:
Nom du champ : protected java.lang.String javaapplicationreflect.Vehicule.marque
Nom du champ : protected java.lang.String javaapplicationreflect.Vehicule.couleur
Nom du champ : protected int javaapplicationreflect.Vehicule.annee
Nom du champ : protected double javaapplicationreflect.Vehicule.taille
Nom du champ : protected boolean javaapplicationreflect.Vehicule.estAssure
BUILD SUCCESSFUL (total time: 0 seconds)
```

6) Récupérer la liste des constructeurs d'une classe.

On aussi récupérer la liste de constructeurs.

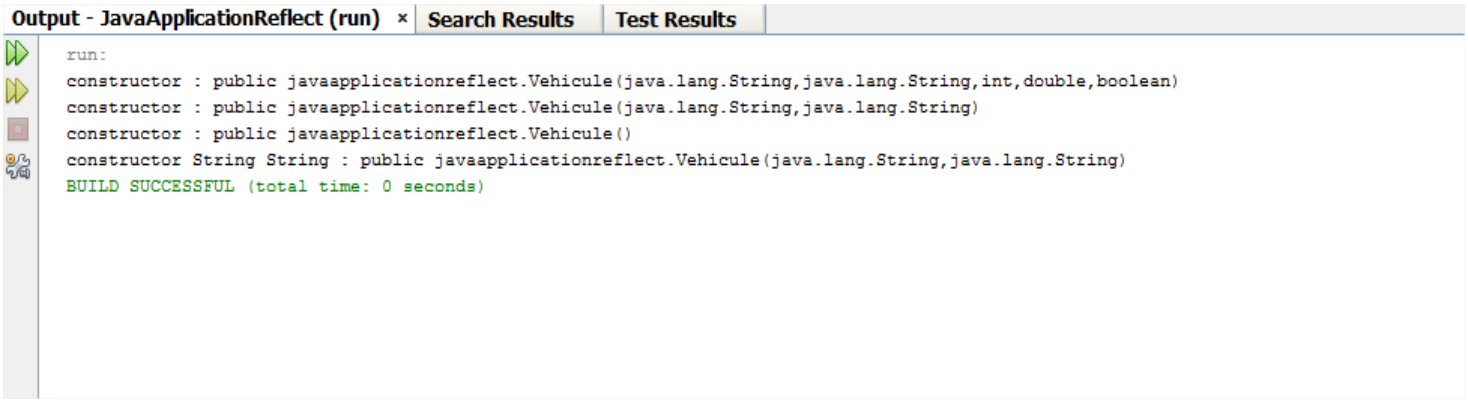
```java
public static void displayConstructors() {
    String methodName = "displayConstructors";
    Class theClass = null;
    try {
        theClass = Class.forName("javaapplicationreflect.Vehicule");
        Constructor[] constructors = theClass.getConstructors();
        for (Constructor constructor : constructors) {
            System.out.println("constructor : " + constructor);
        }
        Class[] arguments = new Class[]{String.class, String.class};
        Constructor constructor = theClass.getConstructor(arguments);
        System.out.println("constructor String String : " + constructor);
    } catch (ClassNotFoundException | SecurityException | NoSuchMethodException ex) {
        System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + ex);
    }
}
```

Ligne 137 : On récupère la liste des constructeurs de la classe « Vehicule ».
Ligne 89 : On récupère la signature de constructeur « Vehicule(String marque, String couleur) » en lui passant comme argument « new Class [] {String.class, String.class } ».

Nous rappelons que le code du constructeur est le suivant :

```
public Vehicule(String marque, String couleur) {
      this.marque = marque;
      this.couleur = couleur;
      System.out.println("Contructeur Vehicule(string,string)");
 }
```

On obtient le résultat suivant :

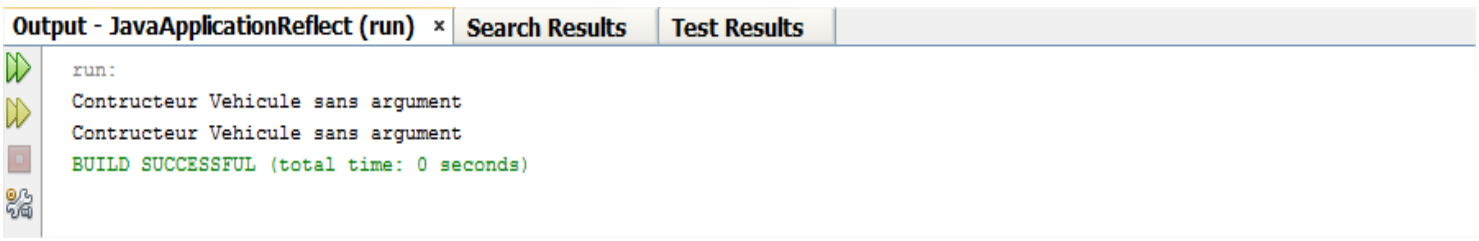


7) Instancier une classe de manière dynamique.

Dans l'exemple, ci-dessous nous allons fabriquer une instance de « Vehicule » de manière inhabituelle en utilisant la retro inspection.

```
150
151     public static void instanciateVehicule() {
152         String methodName = "instanciateVehicule";
153         try {
154             Vehicule renault = new Vehicule();
155             // equivalent à
156             Class theClass = Class.forName("javaapplicationreflect.Vehicule");
157             Object objectCitroen = theClass.newInstance();
158         } catch (ClassNotFoundException | SecurityException | IllegalAccessException | InstantiationException ex) {
159             System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + ex);
160         }
161     }
```

Ligne 154 et 157 : Ces deux lignes sont totalement équivalentes. D'ailleurs, même la console affiche :



Ce qui correspont exactement à l'instanciation de deux object de type « Vehicule ».

On peut caster « objectCitroen » en « Vehicule ».

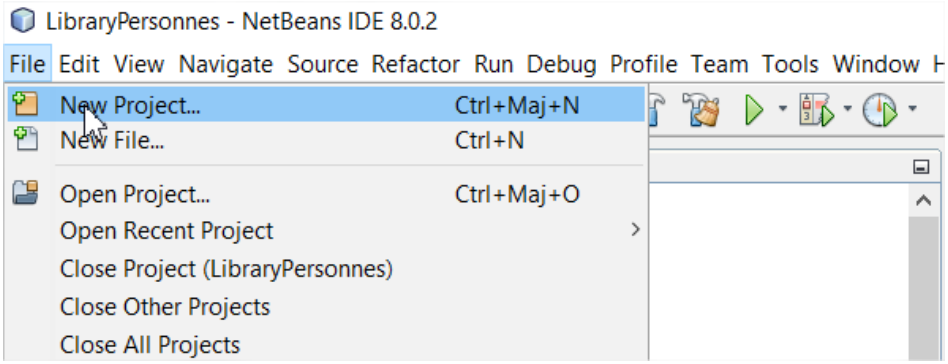
```
150
151     public static void instanciateVehicule() {
152         String methodName = "instanciateVehicule";
153         try {
154             Vehicule citroen = null;
155             Vehicule renault = new Vehicule();
156             // equivalent à
157             Class theClass = Class.forName("javaapplicationreflect.Vehicule");
158             Object objectCitroen = theClass.newInstance();
159             if (objectCitroen instanceof Vehicule) {
160                 citroen = (Vehicule) objectCitroen;
161             }
162         } catch (ClassNotFoundException | SecurityException | IllegalAccessException | InstantiationException ex) {
163             System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + ex);
164         }
165     }
```

XXVI) Maven

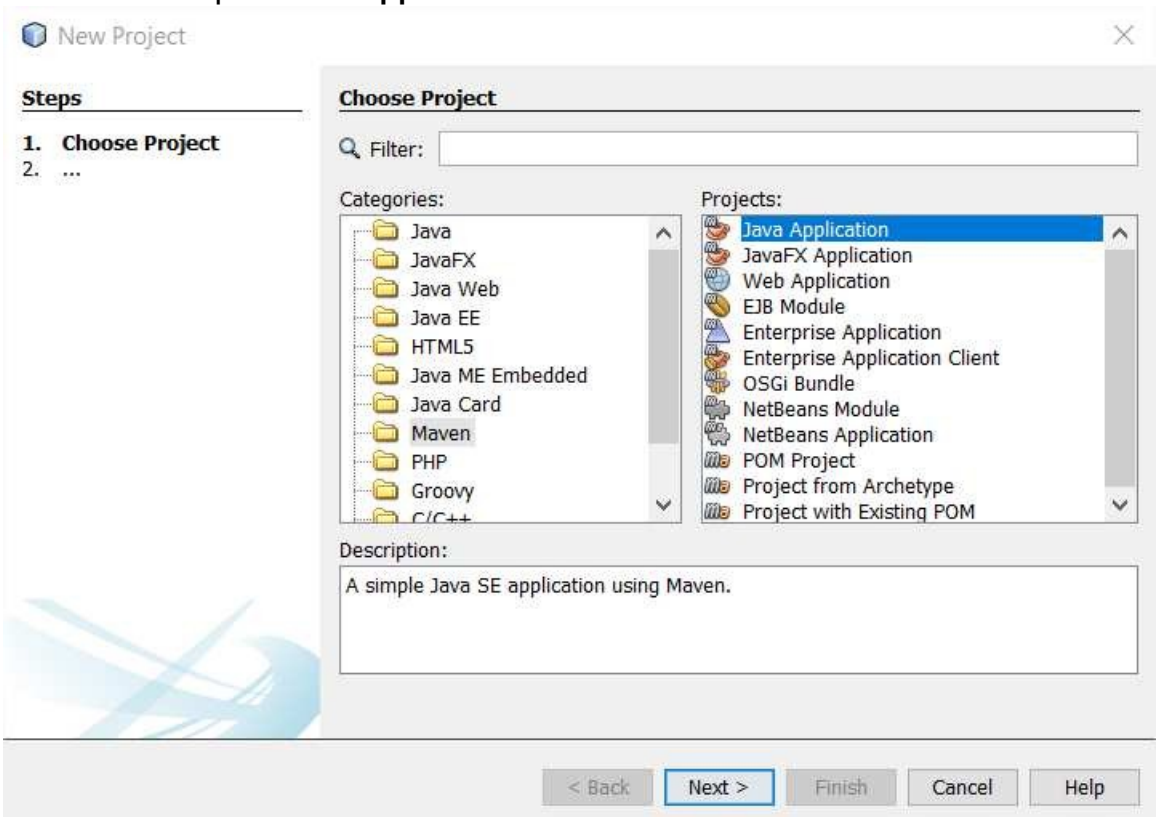
Maven est un outil de la fondation Apache pour la gestion et l'automatisation de production des projets logiciels Java.
Il permet en outre la gestion des bibliothèques d'un projet. Celles-ci sont formées de l'ensemble de archives jars qui doivent être dans le « **Classpath** » du projet.
Les archives jars d'un projet sont parfois très nombreuses. Par exemple, si nous voulons utiliser l'ORM (**Object Relational Mapping**) Hibernate. Cet ORM a des dépendances avec une dizaine d'archives jar. L'intérêt de Maven est qu'il nous affranchit de les connaître toutes. Il nous suffit d'indiquer dans notre projet que nous avons besoin d'Hibernate en donnant toutes les informations utiles pour trouver l'archive principale de cet ORM. Maven télécharge alors toutes les bibliothèques nécessaires à Hibernate. On appelle cela, les dépendances d'Hibernate. Une bibliothèque nécessaire à Hibernate peut elle-même dépendre d'autres archives. Celles-ci seront également téléchargées.
Toutes ces bibliothèques sont placées dans un dossier appelé le dépôt local de Maven.

Maven utilise un paradigme connu sous le nom de **Project Object Model** (POM) afin de décrire un projet logiciel, ses dépendances avec des modules externes. Chaque projet ou sous-projet est configuré par un POM qui contient les informations nécessaires à Maven pour traiter le projet (nom du projet, numéro de version, dépendances vers d'autres projets, bibliothèques nécessaires à la compilation, noms des contributeurs etc.). Ce POM se matérialise par un fichier « pom.xml » à la racine du projet.

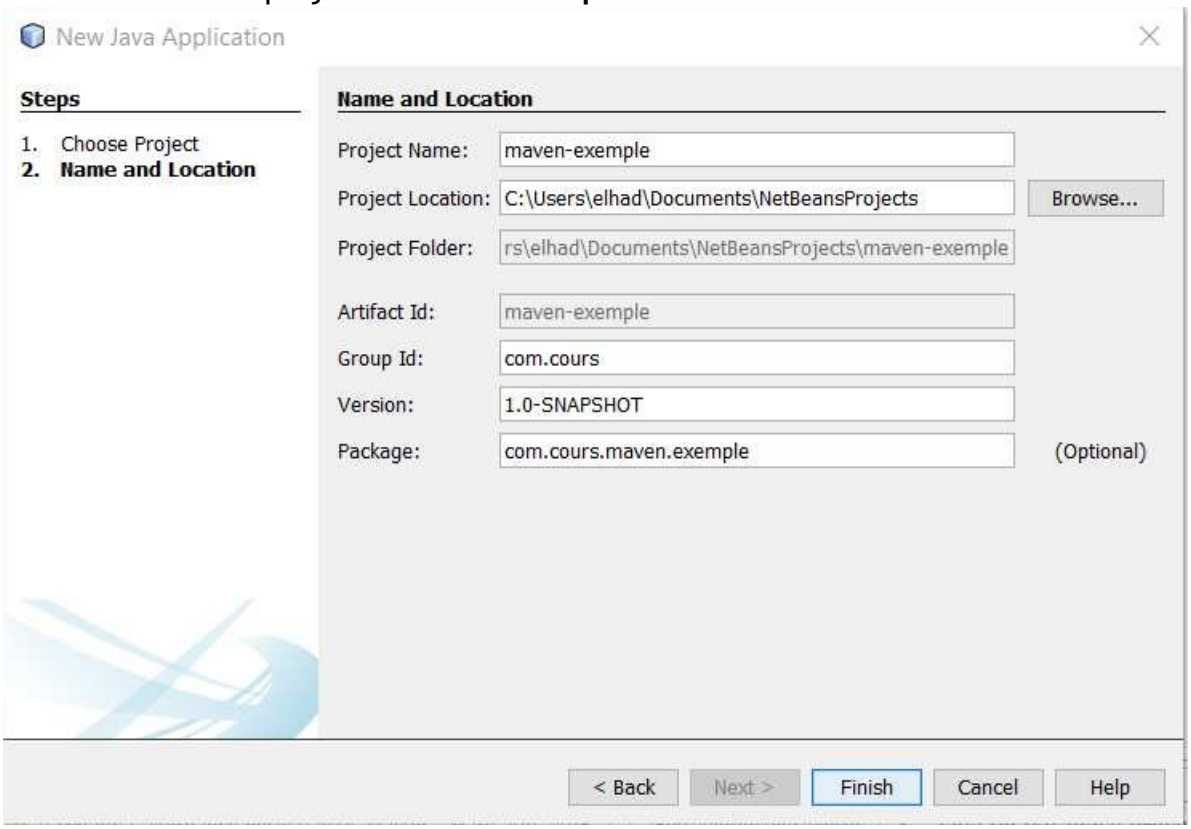
L'un des grands intérêts de Maven est qu'un projet Maven est facilement partageable. Si on le transfert d'un poste à un autre et que les dépendances du projet ne sont pas présentes dans le dépôt local du nouveau poste, elles seront téléchargées.
Créer le projet Maven **maven-exemple**.



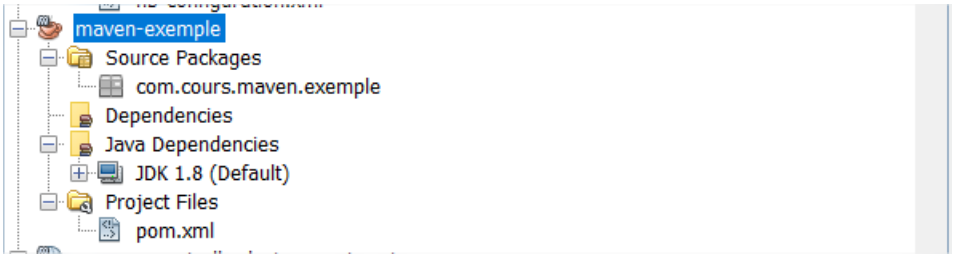
Choisir **Maven** puis **Java Application.**



On va nommer ce projet « **maven-exemple** ».



Le projet **maven-exemple** devient :



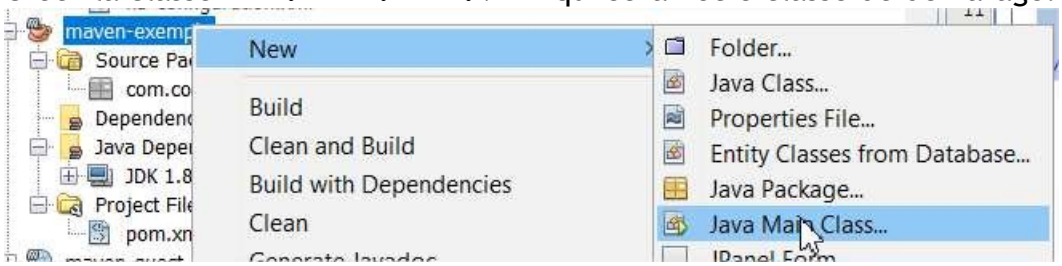
La création du projet Maven a généré un fichier **pom.xml** dont le contenu est le suivant :

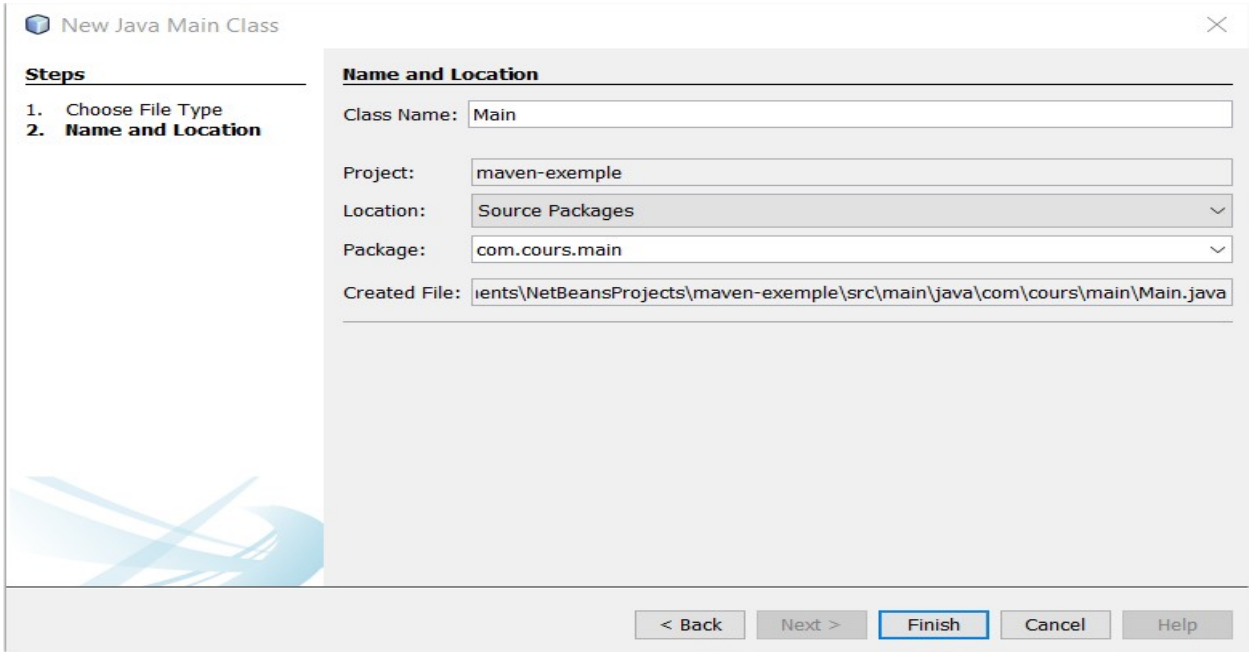
```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.cours</groupId>
    <artifactId>maven-exemple</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.7</maven.compiler.source>
        <maven.compiler.target>1.7</maven.compiler.target>
    </properties>
</project>
```

Un objet Maven est défini par quatre propriétés :

[**groupId**] : une information qui ressemble à un nom de package. Par exemple, la bibliothèque ORM d'Hibernate a **groupId=org.hibernate**.
[**artifactId**] : le nom de l'objet Maven. Par exemple, l'artifact Hibernate a pour **artifactId= hibernate-core**.
[**version**] : numéro de version de l'artifact Maven. Par exemple, la dernière version d'Hibernate a pour version « **5.0.7.Final** ».
[**packaging**] : le format prise par l'artifact (.war, .jar).

Créer la classe **com.cours.main.Main** qui sera notre classe de démarage.





Nous allons, par exemple, mettre en place log4j qui est une librairie de gestion de logs. Les dépendances pour cette librairtie seront :

```
<dependencies>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
  </dependency>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>apache-log4j-extras</artifactId>
    <version>1.2.17</version>
  </dependency>
  <dependency>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging</artifactId>
    <version>1.2</version>
    <type>jar</type>
  </dependency>
</dependencies>
```

Le fichier **pom.xml** devient :

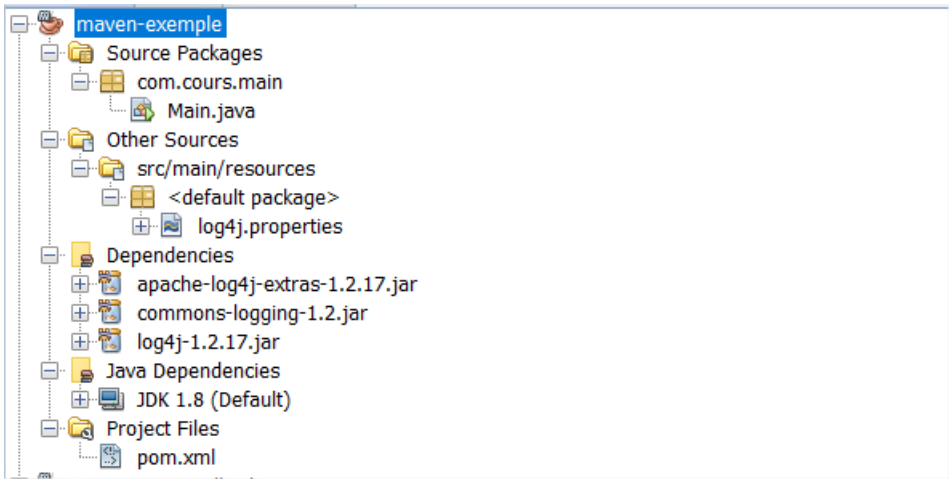
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.cours</groupId>
  <artifactId>maven-exemple</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.17</version>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>apache-log4j-extras</artifactId>
      <version>1.2.17</version>
    </dependency>
    <dependency>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
      <version>1.2</version>
      <type>jar</type>
    </dependency>
  </dependencies>
</project>
```

Créez le fichier **log4j.properties** qui nous servira à configurer les logs de l'application . Vous mettrez le fichier **log4j.properties** dans le dossier **maven-exemple/src/main/resources**. Bien entendu, si le dossier **resources** n'existe pas il faudra le créer.

Le fichier **log4j.properties** aura pour contenu :

```
# Set root logger level to DEBUG and its only appender to CONSOLE.
log4j.rootLogger=DEBUG, CONSOLE

# A1 is set to be a ConsoleAppender.
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.encoding=UTF-8

# A1 uses PatternLayout.
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=[%d{dd/MM/yyyy HH:mm:ss} %p %C{1}.%M] %m%n

# Change the level of messages for various packages.
log4j.logger.com.cours*=DEBUG
```

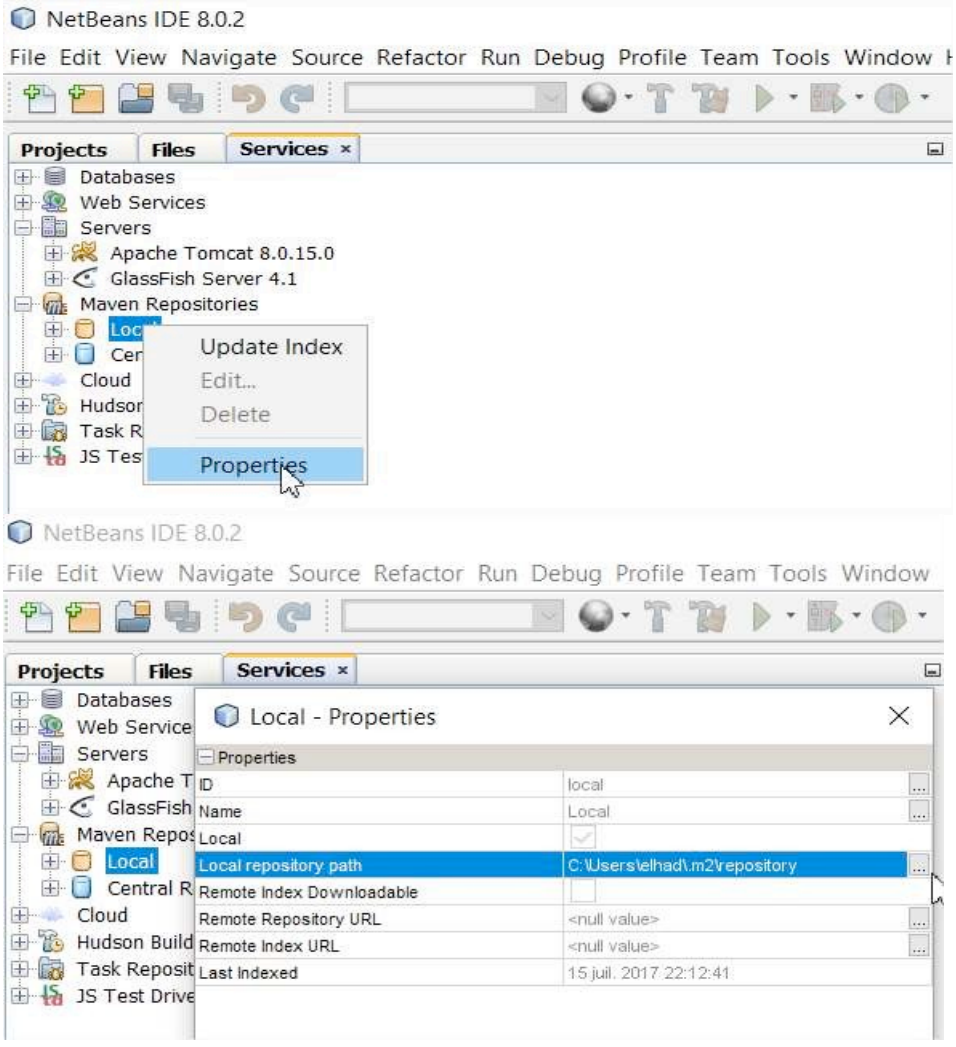
Le projet **maven-exemple** devient alors :



[Source packages] : contient les classes Java du projet.
[Test packages] : contient les classes de test du projet.
[Dependencies] : contient les archives .jar nécessaires au projet et gérées par Maven.
[Test Dependencies] : contient les archives .jar nécessaires aux tests du projet et gérées par Maven.
[Java Dependencies] : contient les archives .jar nécessaires au projet et non gérées par Maven.
[Project Files] : contient les fichiers de configuration de Maven et NetBeans.
[Other Sources] : contient les fichiers de configuration de l'application.

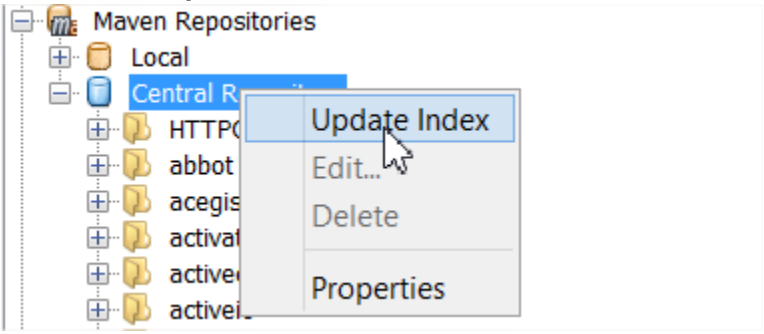
Regardons maintenant le système de fichier de Maven, pour cela vous allez dans l'onglet **Services 👁 Maven Repositories.**

- Le dossier **local** dans lequel on retrouvera tous nos librairies local telechargées par Maven. Pour connaitre le répertoire local de depot, il suffit de cliquer sur « Local » 👁 « Properties ».



Notre répertoire de depot local de Maven par defaut est « C:\Users**MonUserWindows**\.m2\repository ».

- Le dossier **Central Repository** dans lequel on retrouvera les librairies disponibles sur Maven. On peut être amener à mettre à jour la liste des librairies de Maven en faisant **Clic** ⦿**Update Index**.



Revenons sur la classe de teste **Main** dont le contenu est par défaut :

```
1   package com.cours.main;
2
3
4   public class Main {
5
6       public static void main(String[] args) {
7
8       }
9   }
10
```

La classe **Main** peut devenir :

```
1   package com.cours.main;
2
3   import org.apache.commons.logging.Log;
4   import org.apache.commons.logging.LogFactory;
5
6   public class Main {
7
8       private static final Log log = LogFactory.getLog(Main.class);
9
10      public static void main(String[] args) {
11          System.out.println("Bonjour le Monde");
12          log.debug("Bonjour le Monde");
13      }
14  }
```

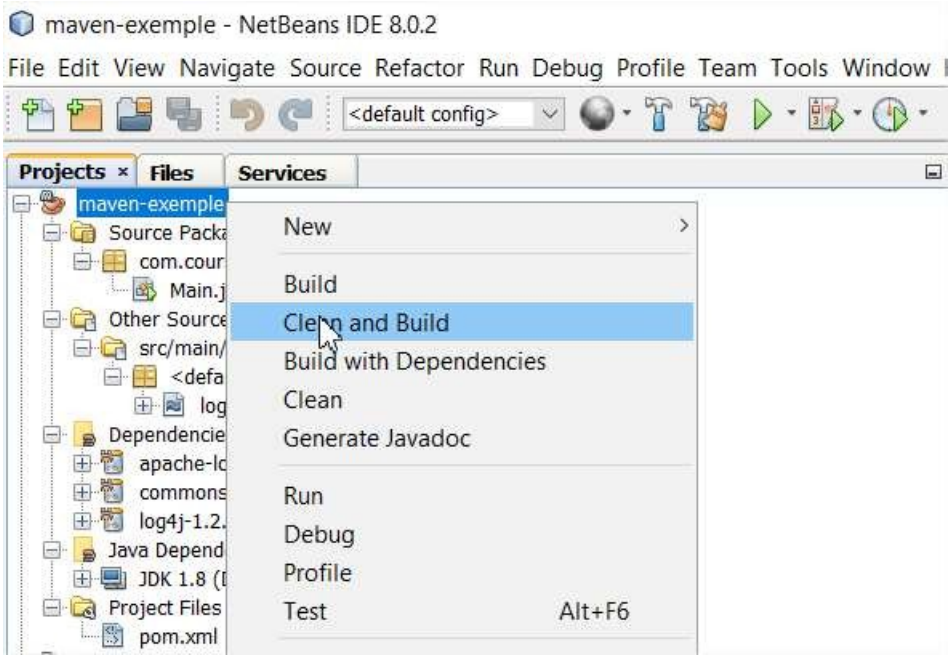
En version copiable :

```
package com.cours.main;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class Main {

   private static final Log log = LogFactory.getLog(Main.class);

   public static void main(String[] args) {
      System.out.println("Bonjour le Monde");
      log.debug("Bonjour le Monde");
   }
}
```

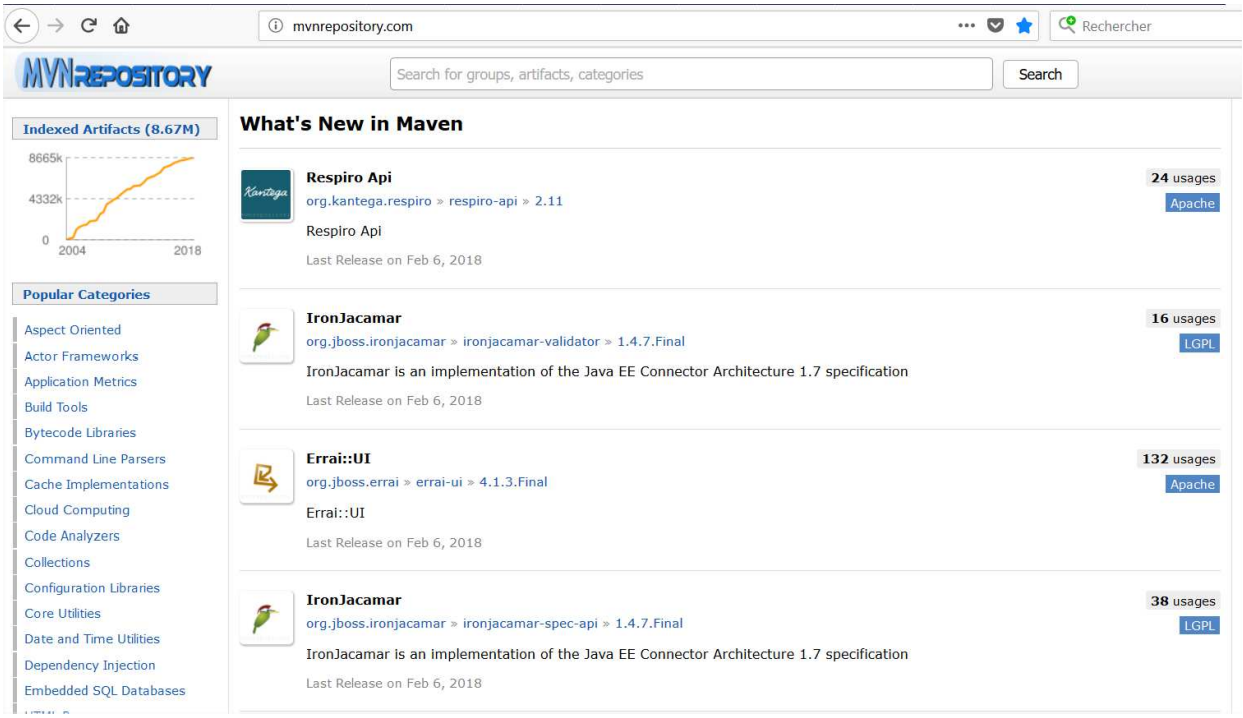
Ligne 3 et 4 : l'interface **Log** et la classe **LogFactory** appartiennent à la librairie log4j.il ne sont pas native de la JVM.
Ligne 11 et 12 : ces lignes affichent la même chose mais le log est conditionnel, on peut choisir de l'afficher ou pas (par l'intermediaire du fichier **log4j.properties**) alors que le System.out s'affiche tout temps.
On obtient à l'exécution:

```
Debug (maven-exemple) × Debugger Console ×
NetBeans: JPDA Listening Start...
JPDA Address: DESKTOP-PNFR4O2:51761
Port:51761
cd C:\Users\elhad\Documents\NetBeansProjects\maven-exemple; "JAVA_HOME=C:\\Program Files\\Java\\j
Running NetBeans Compile On Save execution. Phase execution is skipped and output directories of
Scanning for projects...

------------------------------------------------------------------------
Building maven-exemple 1.0-SNAPSHOT
------------------------------------------------------------------------

--- exec-maven-plugin:1.2.1:exec (default-cli) @ maven-exemple ---
Bonjour le Monde
[08/02/2018 13:45:39 DEBUG Main.main] Bonjour le Monde
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time: 0.857s
Finished at: Thu Feb 08 13:45:39 CET 2018
Final Memory: 5M/123M
------------------------------------------------------------------------
```

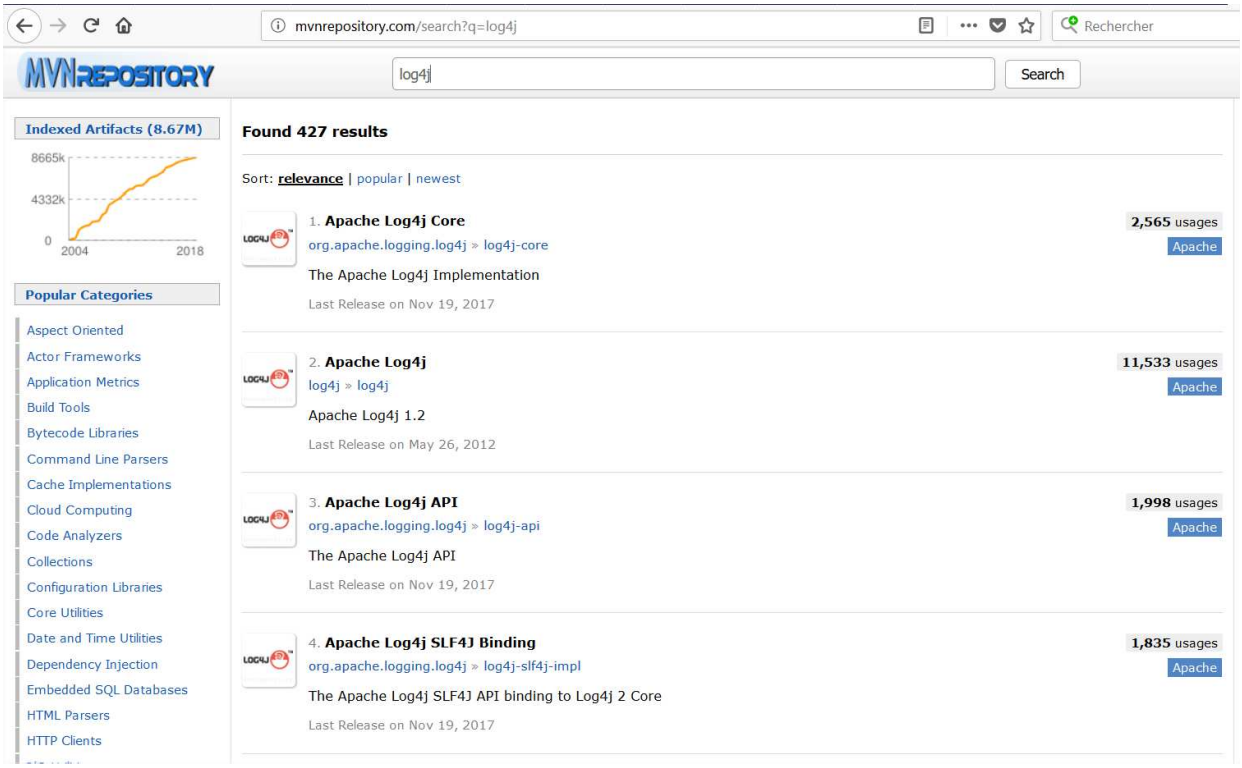
Pour nettoyer notre projet **Maven,** on peut faire un **Clean and Build**. En effet, il faut faire un **Clean and Build** à chaque fois qu'on modifie le fichier « **pom.xml** » pour permettre à Maven de télécharger toutes les archives jar necessaires.



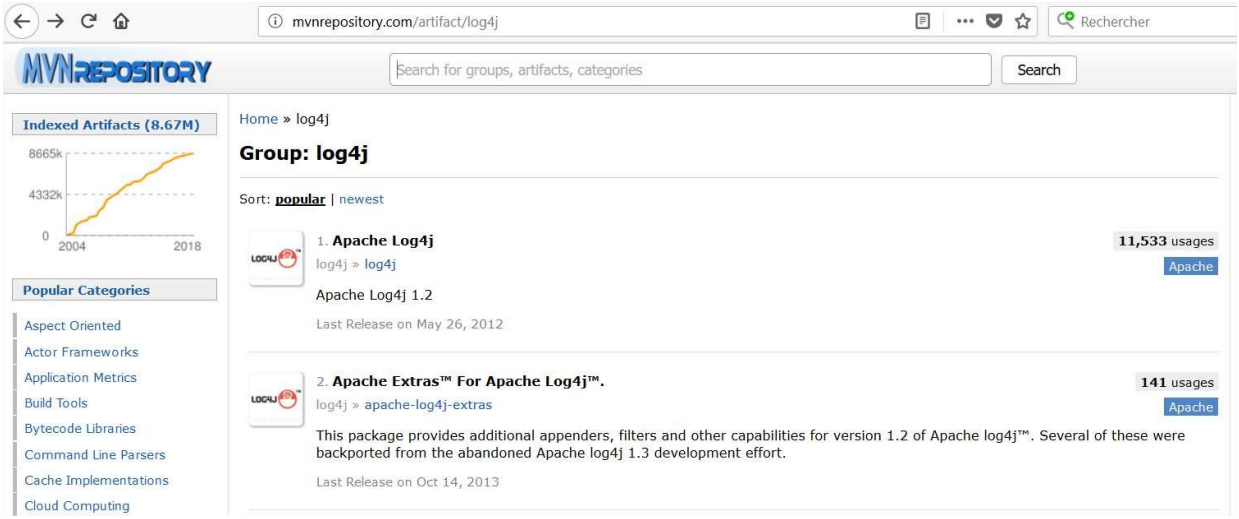
Si on ne connait pas les informations telles que [**groupId**] et [**artifactId**] alors on peut les retrouver grâce au site http://mvnrepository.com.

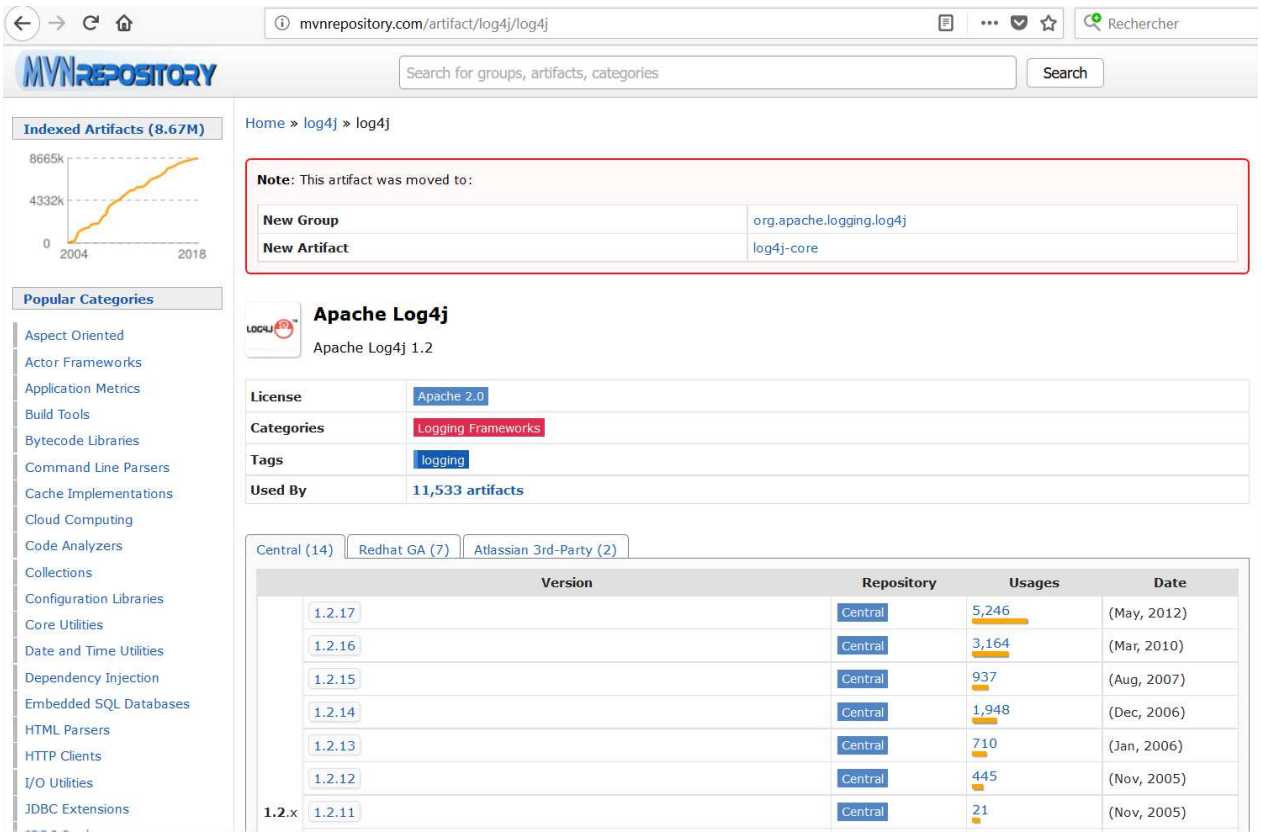


Vous pouvez, par exemple, taper sur la barre de recherche **log4j**.

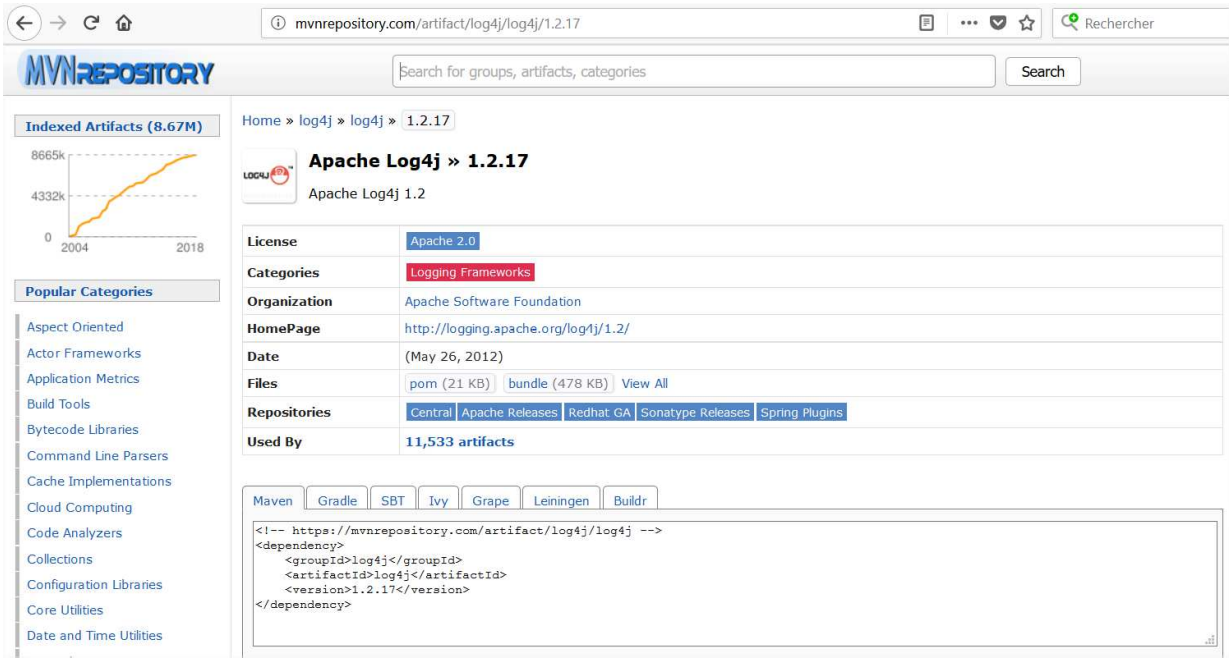


Cliquez sur la version **Apache Log4j**.

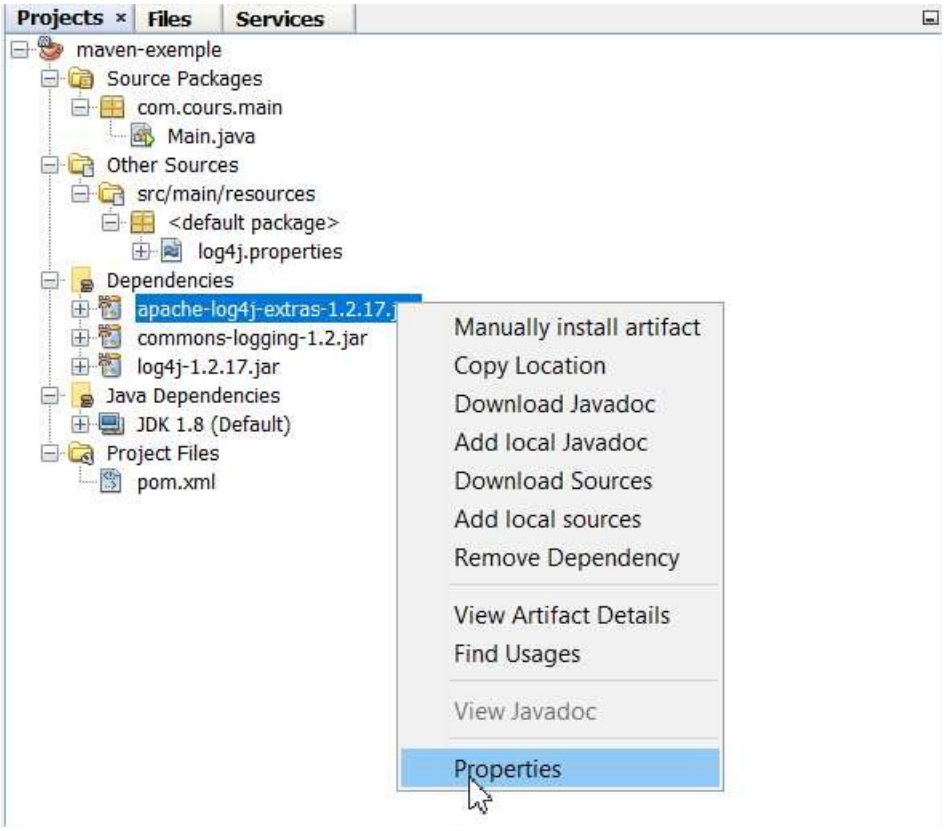


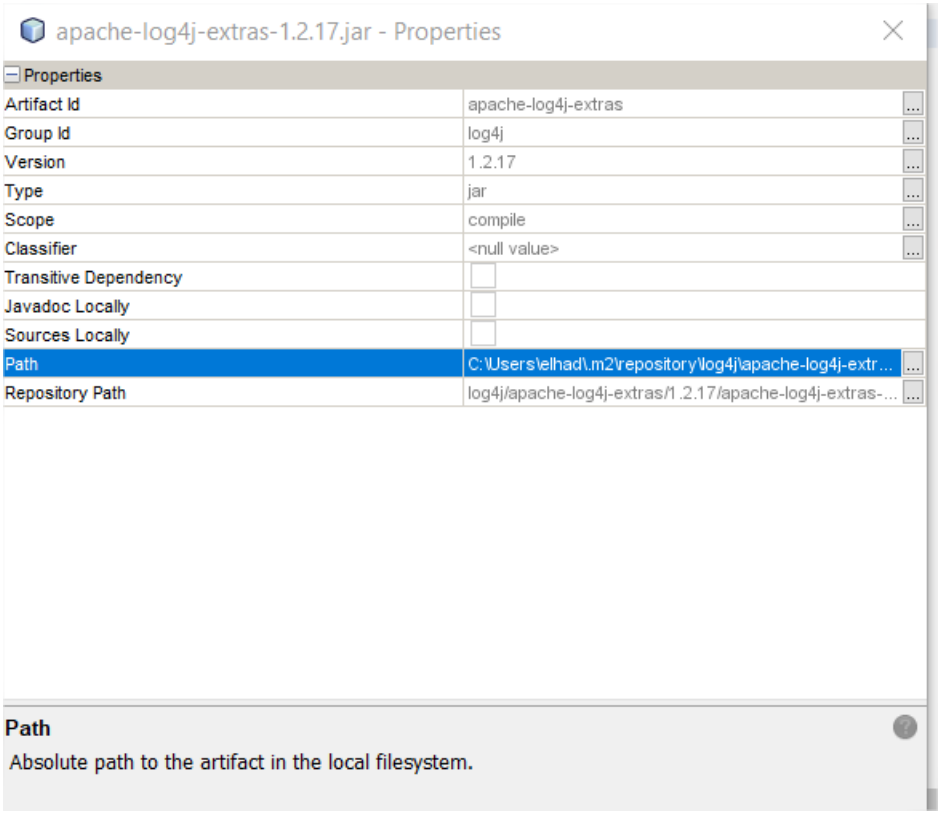


Cliquez sur la version **1.2.17 de log4j**.



Pour télécharger ces differentes bibliotheques, il suffit de faire :



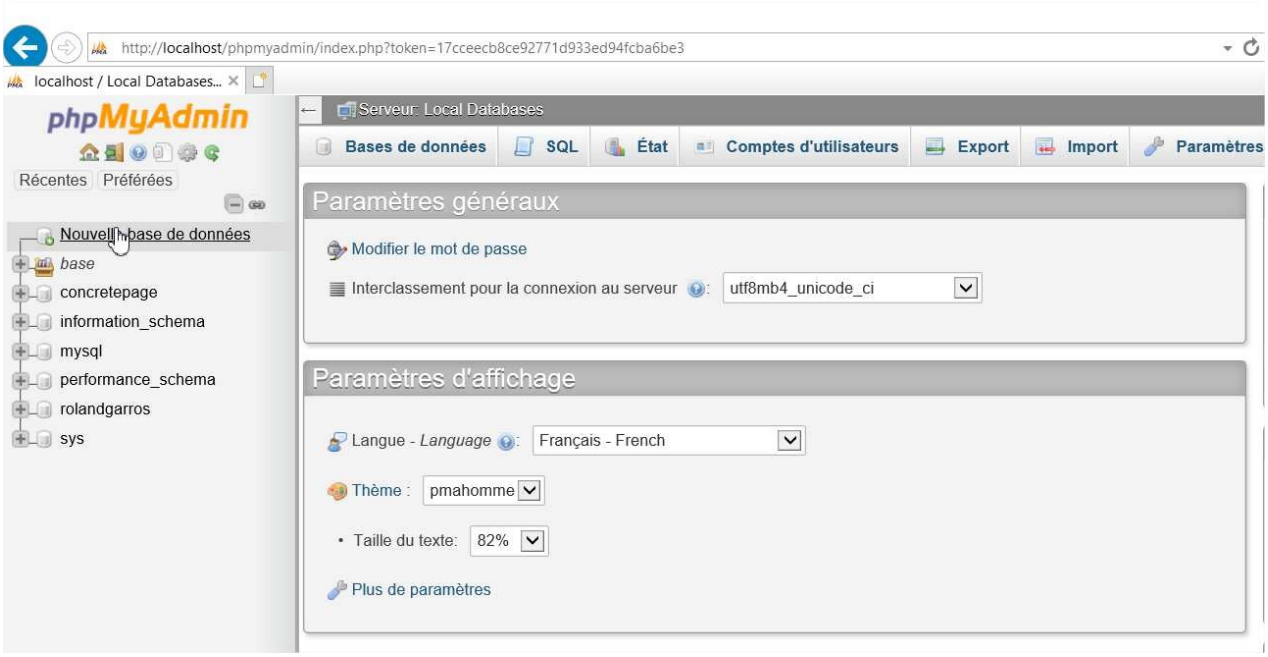


Le champ **Path** permet d'obtenir le chemin complet vers la librairie log4j qui sera C:\Users**MonUserWindows**\.m2\repository\log4j\apache-log4j-extras\1.2.17\apache-log4j-extras-1.2.17.jar

XXVII) Gestion base de données MySQL avec Java

1) Initialisation base de données base_personnes

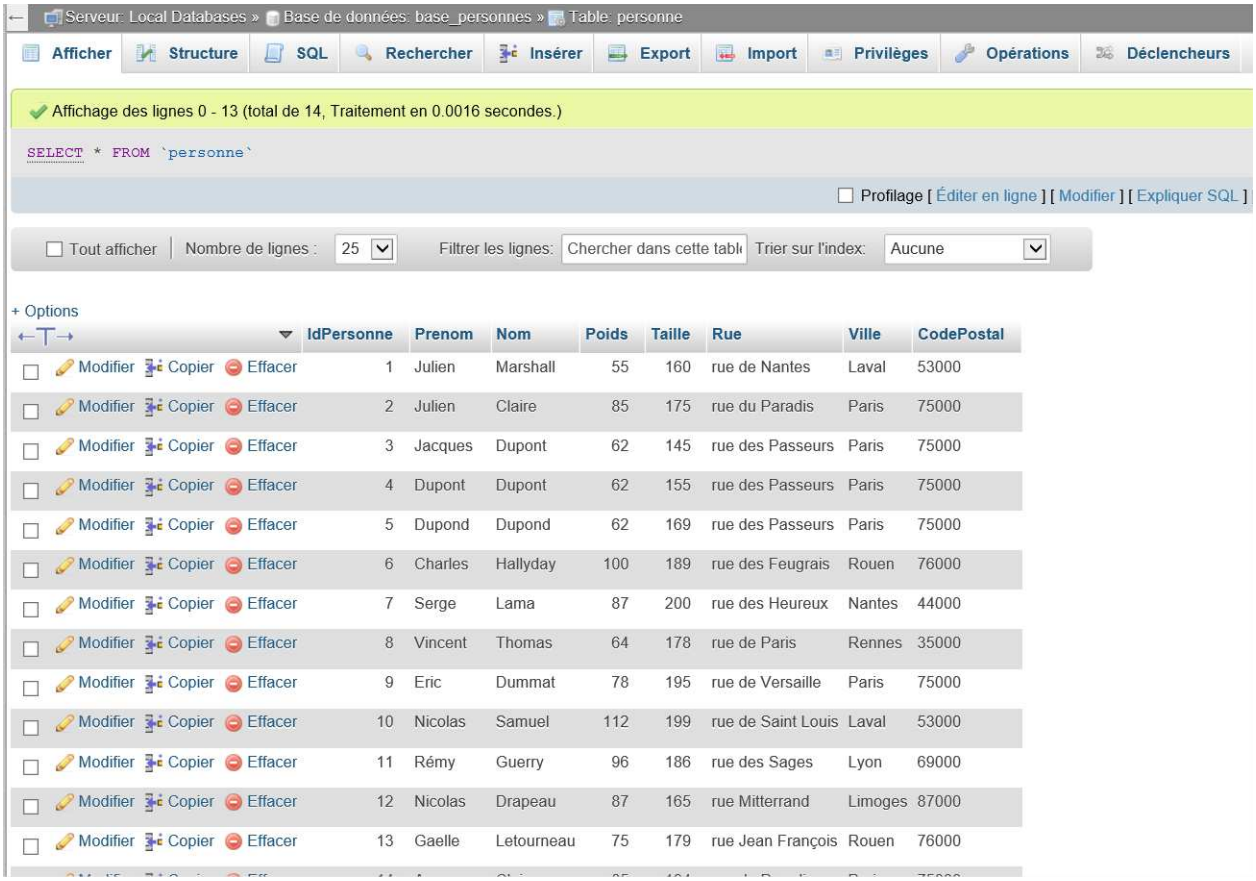
Nous allons voir, dans cette partie du cours, la communication entre une base de données MYSQL avec Java. Nous nommerons cette base de données **base_personnes**.



```
SET FOREIGN_KEY_CHECKS = 0;
DROP TABLE IF EXISTS Personne;

 CREATE TABLE Personne(
        IdPersonne INTEGER PRIMARY KEY AUTO_INCREMENT,
        Prenom VARCHAR(100),
        Nom VARCHAR(100),
        Poids double,
        Taille double,
        Rue VARCHAR(100),
        Ville VARCHAR(100),
        CodePostal VARCHAR(100)
 )ENGINE=InnoDB  DEFAULT CHARSET=utf8;

 INSERT INTO Personne(Prenom,Nom,Poids,Taille,Rue,Ville,CodePostal) VALUES ('Julien',
'Marshall',55,160,'rue de Nantes','Laval','53000');
 INSERT INTO Personne(Prenom,Nom,Poids,Taille,Rue,Ville,CodePostal) VALUES ('Julien',
'Claire',85,175,'rue du Paradis','Paris','75000');
 INSERT INTO Personne(Prenom,Nom,Poids,Taille,Rue,Ville,CodePostal) VALUES ('Jacques',
'Dupont',62,145,'rue des Passeurs','Paris','75000');
 INSERT INTO Personne(Prenom,Nom,Poids,Taille,Rue,Ville,CodePostal) VALUES ('Dupont',
'Dupont',62,155,'rue des Passeurs','Paris','75000');
 INSERT INTO Personne(Prenom,Nom,Poids,Taille,Rue,Ville,CodePostal) VALUES ('Dupond',
'Dupond',62,169,'rue des Passeurs','Paris','75000');
 INSERT INTO Personne(Prenom,Nom,Poids,Taille,Rue,Ville,CodePostal) VALUES ('Charles',
'Hallyday',100,189,'rue des Feugrais','Rouen','76000');
 INSERT INTO Personne(Prenom,Nom,Poids,Taille,Rue,Ville,CodePostal) VALUES ('Serge',
'Lama',87,200,'rue des Heureux','Nantes','44000');
 INSERT INTO Personne(Prenom,Nom,Poids,Taille,Rue,Ville,CodePostal) VALUES ('Vincent',
'Thomas',64,178,'rue de Paris','Rennes','35000');
 INSERT INTO Personne(Prenom,Nom,Poids,Taille,Rue,Ville,CodePostal) VALUES ('Eric',
'Dummat',78,195,'rue de Versaille','Paris','75000');
 INSERT INTO Personne(Prenom,Nom,Poids,Taille,Rue,Ville,CodePostal) VALUES ('Nicolas',
'Samuel',112,199,'rue de Saint Louis','Laval','53000');
 INSERT INTO Personne(Prenom,Nom,Poids,Taille,Rue,Ville,CodePostal) VALUES ('Rémy',
'Guerry',96,186,'rue des Sages','Lyon','69000');
 INSERT INTO Personne(Prenom,Nom,Poids,Taille,Rue,Ville,CodePostal) VALUES ('Nicolas',
'Drapeau',87,165,'rue Mitterrand','Limoges','87000');
 INSERT INTO Personne(Prenom,Nom,Poids,Taille,Rue,Ville,CodePostal) VALUES ('Gaelle',
'Letourneau',75,179,'rue Jean François','Rouen','76000');
 INSERT INTO Personne(Prenom,Nom,Poids,Taille,Rue,Ville,CodePostal) VALUES ('Anne',
'Claire',85,194,'rue du Paradis','Paris','75000');
```

Copiez le code SQL ci-dessus dans l'onglet SQL.

Cliquzr ensuite sur la table **personne** pour voir les données de cette table.



	IdPersonne	Prenom	Nom	Poids	Taille	Rue	Ville	CodePostal
Modifier Copier Effacer	1	Julien	Marshall	55	160	rue de Nantes	Laval	53000
Modifier Copier Effacer	2	Julien	Claire	85	175	rue du Paradis	Paris	75000
Modifier Copier Effacer	3	Jacques	Dupont	62	145	rue des Passeurs	Paris	75000
Modifier Copier Effacer	4	Dupont	Dupont	62	155	rue des Passeurs	Paris	75000
Modifier Copier Effacer	5	Dupond	Dupond	62	169	rue des Passeurs	Paris	75000
Modifier Copier Effacer	6	Charles	Hallyday	100	189	rue des Feugrais	Rouen	76000
Modifier Copier Effacer	7	Serge	Lama	87	200	rue des Heureux	Nantes	44000
Modifier Copier Effacer	8	Vincent	Thomas	64	178	rue de Paris	Rennes	35000
Modifier Copier Effacer	9	Eric	Dummat	78	195	rue de Versaille	Paris	75000
Modifier Copier Effacer	10	Nicolas	Samuel	112	199	rue de Saint Louis	Laval	53000
Modifier Copier Effacer	11	Rémy	Guerry	96	186	rue des Sages	Lyon	69000
Modifier Copier Effacer	12	Nicolas	Drapeau	87	165	rue Mitterrand	Limoges	87000
Modifier Copier Effacer	13	Gaelle	Letourneau	75	179	rue Jean François	Rouen	76000

Créez, pour cette base de données, l'utilisateur **application** avec le mot de passe **passw0rd** pour la suite du cours.

2) Connecteur JDBC MySQL

JDBC veut dire **Java DataBase Connectivity**. Il s'agit de librairies Java permettant de se connecter e
d'interagir avec des bases de données (MySQL, Oracle, DB2 etc.…).
Nous allons voir, dans cette partie du cours, comment récupérer des données de la base de données
via la librairie JDBC.

Créez le projet Maven **maven-data-base** et la classe **com.cours.main.Main**.

Le fichier **pom.xml** sera:

```
<?xml version="1.0" encoding="UTF-8"?>
<project  xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.cours</groupId>
  <artifactId>maven-data-base</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>
   <dependencies>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version> 5.1.41</version>
    </dependency>
  </dependencies>
</project>
```

La classe **com.cours.main.Main** :
Soit la méthode **displayAllPersonnes**:

```java
package com.cours.main;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class Main {

    public static void main(String[] args) {
        displayAllPersonnes();
    }

    public static void displayAllPersonnes() {
        String methodName = "displayAllPersonnes";
        Connection connection;
        try {
            // Url de connexion en base de donnée
            String url = "jdbc:mysql://localhost:3306/base_personnes?useSSL=false";
            // Utilisateur de la base de données
            String user = "application";
            // Mot de passe de la base de données
            String password = "passw0rd";
            // instentiation du driver JDBC
            Class.forName("com.mysql.jdbc.Driver");
            // Objet Connection
            connection = DriverManager.getConnection(url, user, password);
            String sqlCommand = "select * from personne";
            ResultSet result = null;
            PreparedStatement preparedStatement = null;
            try {
                preparedStatement = connection.prepareStatement(sqlCommand);
                result = preparedStatement.executeQuery();
                while (result.next()) {
                    System.out.print("idPersonne: " + result.getInt("idPersonne") + " , prenom: " + result.getString("prenom"));
                    System.out.print(" , nom: " + result.getString("nom") + " , poids: " + result.getString("poids") + " , rue: " + result.getString("rue"));
                    System.out.println(" , ville: " + result.getString("ville") + " , codePostal: " + result.getString("codePostal"));
                }
            } finally {
                if (result != null) {
                    result.close();
                }
                if (preparedStatement != null) {
                    preparedStatement.close();
                }
                if (connection != null) {
                    connection.close();
                }
            }
        } catch (ClassNotFoundException | SQLException e) {
            System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + e.getMessage());
        }
    }

}
```

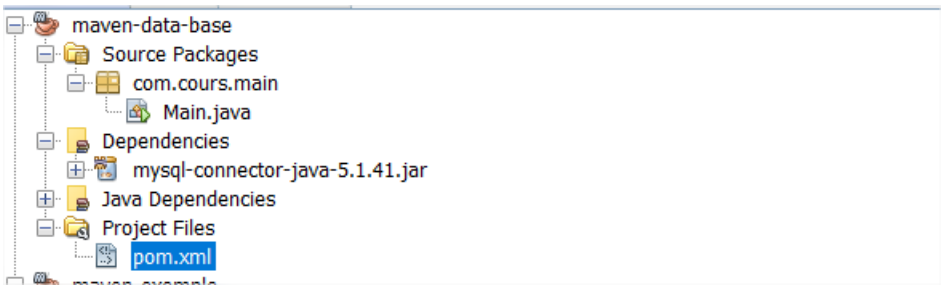
En version copiable :

```java
package com.cours.main;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class Main {

  public static void main(String[] args) {
    displayAllPersonnes();
  }

  public static void displayAllPersonnes() {
```

```
    String methodName = "displayAllPersonnes";
    Connection connection;
    try {
      // Url de connexion en base de donnée
      String url = "jdbc:mysql://localhost:3306/base_personnes?useSSL=false";
      // Utilisateur de la base de données
      String user = "application";
      // Mot de passe de la base de données
      String password ="passw0rd";
      // instentiation du driver JDBC
      Class.forName("com.mysql.jdbc.Driver");
      // Objet Connection
      connection = DriverManager.getConnection(url, user, password);
      String sqlCommand = "select * from personne";
      ResultSet result = null;
      PreparedStatement preparedStatement = null;
      try {
        preparedStatement = connection.prepareStatement(sqlCommand);
        result = preparedStatement.executeQuery();
        while (result.next()) {
          System.out.print("idPersonne: " + result.getInt("idPersonne") + " , prenom: " +
result.getString("prenom"));
          System.out.print(" , nom: " + result.getString("nom") + " , poids: " + result.getString("poids") + " , rue: "
+ result.getString("rue"));
          System.out.println(" , ville: " + result.getString("ville") + " , codePostal: " +
result.getString("codePostal"));
        }
      } finally {
        if (result != null) {
          result.close();
        }
        if (preparedStatement != null) {
          preparedStatement.close();
        }
        if (connection != null) {
          connection.close();
        }
      }
    } catch (ClassNotFoundException | SQLException e) {
      System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " +
e.getMessage());
    }
  }
}
```

Ligne 20 : url de la base de données MySQL.
Ligne 22 : représente l'utilisateur de la base de données.
Ligne 24 : représente le mot de la base de données.
Ligne 26 : instancie le driver JDBC pour l'accès à la source de donnée en utilisant le principe de réflexion.
Ligne 29 : requête SQL à exécuter.
Ligne 33 : prépare la requête SQL à exécuter avec « PreparedStatement », il est aussi possible d'utiliser « Statement ».
Ligne 34 : lance l'exécution de la requête.
Ligne 35-39 : récupération du résultat de la requête, c'est l'interface « ResultSet » qui permet d'accéder aux données soit avec le nom du champ en base de données (result.getInt("idPersonne")) ou avec l'index du champ dans la table(result.getString(1) pour un champ « String » et result. getInt(1) pour un champ « Integer » (1) représentant le premier champ sélectionner par la requête SQL, pour le deuxième champ il suffit de mettre (2) , puis (3) etc. …).
Ligne 42,45 et 48 : fermeture des ressources liées à l'accès à la source de données. Il est indispensable de toujours fermer ces ressources.

Le projet **maven-data-base** devient donc :



On obtient le résultat suivant suite à l'exécution displayAllPersonnes :

```
idPersonne: 1 , prenom: Julien , nom: Marshall , poids: 55 , rue: rue de Nantes , ville: Laval , codePostal: 53000
idPersonne: 2 , prenom: Julien , nom: Claire , poids: 85 , rue: rue du Paradis , ville: Paris , codePostal: 75000
idPersonne: 3 , prenom: Jacques , nom: Dupont , poids: 62 , rue: rue des Passeurs , ville: Paris , codePostal: 75000
idPersonne: 4 , prenom: Dupont , nom: Dupont , poids: 62 , rue: rue des Passeurs , ville: Paris , codePostal: 75000
idPersonne: 5 , prenom: Dupond , nom: Dupond , poids: 62 , rue: rue des Passeurs , ville: Paris , codePostal: 75000
idPersonne: 6 , prenom: Charles , nom: Hallyday , poids: 100 , rue: rue des Feugrais , ville: Rouen , codePostal: 76000
idPersonne: 7 , prenom: Serge , nom: Lama , poids: 87 , rue: rue des Heureux , ville: Nantes , codePostal: 44000
idPersonne: 8 , prenom: Vincent , nom: Thomas , poids: 64 , rue: rue de Paris , ville: Rennes , codePostal: 35000
idPersonne: 9 , prenom: Eric , nom: Dummat , poids: 78 , rue: rue de Versaille , ville: Paris , codePostal: 75000
idPersonne: 10 , prenom: Nicolas , nom: Samuel , poids: 112 , rue: rue de Saint Louis , ville: Laval , codePostal: 53000
idPersonne: 11 , prenom: Rémy , nom: Guerry , poids: 96 , rue: rue des Sages , ville: Lyon , codePostal: 69000
idPersonne: 12 , prenom: Nicolas , nom: Drapeau , poids: 87 , rue: rue Mitterrand , ville: Limoges , codePostal: 87000
idPersonne: 13 , prenom: Gaelle , nom: Letourneau , poids: 75 , rue: rue Jean François , ville: Rouen , codePostal: 76000
idPersonne: 14 , prenom: Anne , nom: Claire , poids: 85 , rue: rue du Paradis , ville: Paris , codePostal: 75000
```