Annexe A : Installation du poste stagiaire

Introduction

Il faut compter moins d'une heure pour installer un poste stagiaire, à condition que celui-ci possède déjà le système d'exploitation installé.

Nous prenons en compte, dans ce guide l'installation, les plateformes Windows 32 bits (Windows 9x, Windows N.T., Windows 2000, Windows XP).

Les éléments à installer sont les suivants:

- Le JDK.
- La documentation du JDK.
- NetBeans.
- MySQL.
- Le BDK.
- Le BeanBuilder.

Prérequis du système d'exploitation

On doit disposer d'un certain nombre déléments du système Windows :

- L'installation de logiciels nécessite d'avoir le mot de passe administrateur de Windows.
- ODBC doit être installé avec le driver Access. La présence du logiciel Access (Suite Office de Microsoft) est souhaitable, mais pas nécessaire.
- La réseau doit être configuré avec le protocole TCP/IP ; notamment pour les exercices de la partie réseau si on souhaite tester l'API d'un poste à l'autre.

Installation du JDK

Il existe un JDK par plateforme. Nous nous intéressons à la plateforme Windows.

Le programme d'installation peut être téléchargé à l'url :

http://java.sun.com/downloads/

On choisira le pragraphe « Java 2 Platform, Standard Edition (J2SE) » et on téléchargera la version Windows.

Attention à bien télécharger le SDK (Software Development Kit) et non pas le JRE (Java Runtime Environment).

Le programme d'installation se déroule automatiquement, on prendra les valeurs par défaut.

A ce niveau, on peut faire tous les exercices du livre. On utilisera le bloc-note de Windows (Notepad) pour éditer les sources, et la ligne de commande DOS (invité de commandes) pour lancer le compilateur et la machine virtuelle.

Si on souhaite un éditeur un peu plus puissant, l'installation de NetBeans peut être faite.

Installation de la documentation du JDK

C'est un fichier ZIP qui devra être décompressé directement sur le répertoire dans lequel est installé le JDK.

Ce fichier peut être téléchargé sur le site de Sun à l'url suivante :

http://java.sun.com/docs/index.html

Si l'on dispose d'une connexion à Internet, on peut aussi accéder à la documentation en ligne sur le site de Sun. L'avantage est que l'on bénéficiera alors de la documentation la plus à jour de la planète.

Elle se trouve à l'url:

http://java.sun.com/apis.html

Installation de NetBeans

L'installation de NetBeans est facultative, mais vivement conseillée pour le confort de travail des stagiaires.

Ce produit, proposé par la société Sun, est gratuit et open source. Il est téléchargeable sur le site :

http://www.netbeans.org

Pour Windows, on utilisera le programme d'installation en prenant les valeurs par défaut.

Lors de l'installation, il est demandé de choisir une version du JDK (parmi les différentes versions disponibles sur le poste). Choisir au moins la version 1.5.

Installation de MySQL

Nous pouvons télécharger la version communauté sur le site www.mysql.org.

Installation « typical », puis le « configuration wizard » lancé automatiquement .

Lorsque le configuration wizard se lance, spécifier la « Detailed configuration », puis « Developer machine ».

On acceptera les paramètres par défaut, soit :

- Multifunctionnal Database.
- Decision Support (DSS)/OLAP.
- Enable TCP/IP Networking (Port number=3306).
- Enable Strict Mode.
- Standard Character Set (Latin 1).
- Install as Windows Service.
- Modify Security Settings et spécifier un mot de passe pour root (« java » par exemple) et checker « Enable root access from remote machines).

Il est alors possible d'effectuer des requêtes à l'aide d'un outil de commande en ligne très sommaire :

Démarrer / Programmes / MySQL / MySQL Server 5.0 / MySQL Command Line Client

Mais il est plus confortable d'utiliser l'interface graphique MySQL GUI Tools.

MySQL GUI Tools

Le fichier MSI d'installation sous Windows peut aussi être téléchargé depuis le site mysql.com.

C'est un ensemble de trois outils :

- MySQL Administrator 1.2 qui permet de faciliter l'administration.
- MySQL Query Browser 1.2 qui permet d'effectuer des requêtes.
- MySQL Migration Toolkit 1.1 qui permet de faire des migrations de données entre les versions.

Nous travaillerons avec la version 5.0 révision 12.

L'installation se fait sans que l'on ait à choisir d'option particulère.

Installation du BeanBuilder

Ce programme est actuellement disponible en version 1.0 beta. On pardonnera donc à Sun d'éventuels bugs de jeunesse.

Il se télécharge sur le site de Sun à l'url :

http://java.sun.com/products/javabeans/beanbuilder/index.html

C'est un fichier zip qu'il suffit de décompresser sur la racine d'une unité de disque:

`beanbuilder-1_0-beta.zip`.

Un répertoire sera alors créé, il suffira de lancer dans ce répertoire le programme **`run.bat`**.

Liste desfichiers pour l'installation du poste stagiaire

On peut faire un CD-ROM dans lequel mettre tous les programmes. On peut aussi faire une image que l'on copiera simplement sur le poste.

Liste des fichiers :

`j2sdk-1_4_0-win.exe`	Programme d'installation du SDK Java de Sun.
`j2sdk-1_4_0-doc.zip`	Documentation du SDK.
`eclipse-SDK-2.0.2-win32.ZIP`	Application Eclipse d'IBM.
`eclipse-nls-SDK-2.0.x.ZIP`	Complément d'Eclipse pour la localisation en Français.
`mysql-max-3.23.49-win.zip`	Moteur de base de données MySQL.
`docMySQL3.23_v4.pdf.zip`	Documentation PDF de MySQL.
`mysql-connector-java-3.0.6-stable.zip`	Driver JDBC de MySQL.
`bdk1_1.zip`	Bean Development Kit de Sun.
`beanbuilder-1_0-beta.zip`	Version beta du BeanBuilder de Sun.

Annexe B : Description des outils du JDK

Le JDK est un kit de développement minimaliste mais suffisant. On trouve tous les outils dont on peut avoir besoin pour développer des programmes Java.

Ces programmes, qui se trouvent dans le répertoire bin du répertoire d'installation de java sont les suivants :

javac	Le compilateur Java.
java	La machine virtuelle Java, c'est à dire l'interpréteur.
javadoc	Le générateur de documentation.
appletviewer	Un outil pour visualiser les applets sans l'usage d'un navigateur.
jar	Pour créer et lire les fichiers JAR.
jarsigner	Générateur et vérificateur de signatures de fichiers JAR.
extcheck	Détecteur de conflits de fichiers JAR.
jdb	Le debugger Java (mode caractère).
javah	Générateur de fichiers .H pour l'interface native avec le C.
javap	Désassembleur Java. Il ne permet pas de faire du reverse engineering, c'est à dire de décompiler tout le code, mais uniquement l'interface des classes (propriétés et signatures des méthodes).
rmic	C'est le compilateur de RMI. C'est cet utilitaire qui génère les fichiers stubs et skeletons des objets distants.
rmiregistry	Programme de registry pour les objets distants.
serialver	Programme de génération de numéros de versions des classes.
native2ascii	Outil de conversion de textes entre l'ACII et l'Unicode Latin-1.
keytool	Gestionnaire de clés et de certificats.
policytool	Outil graphique pour gérer des polices de sécurité.

Annexe C :
Documenter ses programmes avec Javadoc

Introduction

Bien documenter ses programmes est très important. Sun propose dans le JDK un outil particulièrement intéressant, puisqu'il permet de générer une documentation des programmes à partir des commentaires.

Cela permet de faciliter la maintenance des programmes et de vérifier qu'ils sont correctement commentés.

Le programme javadoc prend en argument les fichiers à partir desquels générer la documentation:

javadoc [options] [noms de packages] [fichiers sources] [classes] [@fichierListe]

On peut donc spécifier des noms de packages, de classes, de fichiers source java, ainsi qu'un fichier dans lequel sont spécifiés tous les fichiers java à inclure dans la documentation (@fichierListe).

Les options les plus utilisées sont les suivantes:

`-overview` ***`fichier`***	Permet de spécifier un fichier HTML qui servira de page accueil à la documentation.
`-verbose`	Affiche plus d'informations sur la console.
`-d` ***`répertoire`***	Permet de spécifier un répertoire dans lequel.
`-version`	Indique dans la documentation le paragraphe de la balise @version.
`-author`	Indique dans la documentation le paragraphe de la balise @author.
`-public`	Ne prend dans la documentation que les classes et les membres publics.
`-protected`	Ne prend dans la documentation que les classes et membres public et protected.
`-package`	Ne prend dans la documentation que les classes et membres public et pretected et du même package.
`-private`	Prend toutes les classes dans la documentation.

Le programme va générer un ensemble de fichiers HTML :

- Un fichier par classe.
- La liste de toutes les classes.
- La liste de tous les packages.
- L'arborescence des classes.

- L'index des classes, des méthodes et des propriétés.

Comment bien documenter

Les commentaires JavaDoc se trouvent obligatoirement avant toute déclaration de classe, interface, méthode, constructeur ou propriété. Tous les autres commentaires seront ignorés.

De plus, seuls les commentaires commençant par **/**** (et non pas **/*** pour des commentaires normaux) seront pris en compte.

Enfin, chaque ligne de commentaire JavaDoc doit débutter par une étoile.

Exemple :

```
/**
* Cette fenêtre ressemble à ceci:<BR>
* <IMG SRC="images/FenAccueil.gif">
*/
```

Ces commentaires seront integrés dans une documentation formatée en HTML. On peut donc y mettre des balises HTML pour améliorer la mise en page.

Dans l'exemple ci-dessus, la balise **IMG** permet d'insérer une image dans la documentation.

Un commentaire peut se terminer par une section dans laquelle on trouve des balises JavaDoc. Elles n'ont rien à voir avec des balises HTML.

Les balises Javadoc

Elles commencent toute par le caractère '@'. Elles peuvent être suivies par un argument.

Exemple :

```
/** Cette classe est un outil*
* @author Jérôme BOUGEAULT
*/
```

Cette balise permet de spécifier l'auteur du programme.

Si on souhaite afficher le caractère @ dans un commentaire JavaDoc, sans qu'il ne soit interprété comme la première lettre d'une balise, on peut utiliser l'entité HTML :

```
&#064;
```

Les balises doivent être en début de ligne de commentaire, sauf certaines qui peuvent être insérés n'importe où dans le commentaire. On les appelle des "balises en ligne", et sont encadrées par des accollades.

Exemple :

```
/** Cette méthode prend les mêmes argument que
* la méthode {@link #afficher( String)afficher une String}
*/
```

Liste des balises :

@author	Permet d'indiquer l'auteur du programme. Exemple : **@author Jérôme BOUGEAULT**
@version	Permet de spécifier une version. Exemple : **@version 2.3**
@see	Invite le lecteur de la documentation à aller voir la référence indiquée à la suite de la balise.Exemple : **@see java.lang.String.compareTo(String)**
@param	Permet de documenter un argument d'une méthode. La balise est suivie par le nom de l'argument et sa description. Exemple : **@param nom Nom de l'utilisateur**
@return	Permet de décrire ce que renvoie la méthode. Exemple : **@return Un entier: le nombre d'éléments**
@deprecated	Permet de signaler que la classe ou le membre est déprécié. La balise peut être suivie d'un texte d'explications. Exemple : **@deprecated Utiliser la méthode convertEuro**
@exception	Synonyme de **@throws**.
@throws	Permet de spécifier les exceptions que la méthode peut envoyer.Exemple: **@throws IOException Si prole fichier n'existe pas**
@since	Permet de spécifier depuis quand existe cette classe ou ce membre. Exemple : **@since 1.0**
{@docRoot}	Représente le répertoire racine de la documentation. C'est utile pour faire une référence relative à un fichier que l'on a ajouté à la documentation. Par exemple un lien HTML sur une image: **<img src="(@docRoot)/images/Plan1.gif">**
{@link}	Insert un lien hypertexte sur une classe ou un membre. Exemple : **Il faut utiliser la méthode: {@link java.lang.String#toString() toString}**
{@linkplain}	C'est la même chose que {@link}, mais le texte affiché utilisera la police par défaut du navigateur (au lien de la police courrier).
{@value}	Permet, dans un commentaire de propriété statique, d'afficher la valeur d'une constante Java.

Exemple :

```
package outils.divers;
import java.awt.event.*;
/** Cette classe permet de gérer les actions de
* l'utilisateur de façon générique.
* @author Bibi
* @version 1.6
*/
public class GestionActions extends MouseAdapter
    implements ActionListener {
  /** Cette méthode doit être implémentée
  * @see java.awt.event.ActionListener
  */
  public void actionPerformed( ActionEvent l) {
  }
  /** Cette méthode vérifie que l'événement est autorisé
  * @since 1.2
  * @throws IOException Si le fichier n'existe pas
  * @throws SQLException Si la base de données n'existe pas
  * @param nom Le nom de l'utilisateur à valider
  * @return true si l'utilisateur a le droit
  */
  public boolean aDroit( String nom) {
    return true;
  }
  /** Méthode polymorphe {@link Object.toString() toString}.
  * Elle renovie simplement le texte:
  * <pre>Gestionnaire d'actions</pre>
  * @deprecated Finalement, ça ne sert à rien!
  * @return La chaine ci-dessus
  */
  public String toString() {
    return( "Gestionnaire d'actions");
  }
}
```

Ce qui donne:

outils.divers

Class GestionActions

```
java.lang.Object
  |
  +--java.awt.event.MouseAdapter
        |
        +--outils.divers.GestionActions
```

All Implemented Interfaces:

java.awt.event.ActionListener, java.util.EventListener, java.awt.event.MouseListener

---

public class **GestionActions**
extends java.awt.event.MouseAdapter
implements java.awt.event.ActionListener

Cette classe permet de gérer les actions de l'utilisateur de façon générique.

Constructor Summary
`GestionActions()`

Method Summary	
void	`actionPerformed(java.awt.event.ActionEvent l)` Cette méthode doit être implémentée
boolean	`aDroit(java.lang.String nom)` Cette méthode vérifie que l'événement est autorisé
java.lang.String	`toString()` **Deprecated.** *Finalement, ça ne sert à rien!*

Methods inherited from class java.awt.event.MouseAdapter
mouseClicked, mouseEntered, mouseExited, mousePressed, mouseReleased

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

GestionActions

```
public GestionActions()
```

Method Detail

actionPerformed

```
public void actionPerformed(java.awt.event.ActionEvent 1)
```

Cette méthode doit être implémentée

Specified by:
`actionPerformed` in interface `java.awt.event.ActionListener`
See Also:
`ActionListener`

---

aDroit

```
public boolean aDroit(java.lang.String nom)
```

Cette méthode vérifie que l'événement est autorisé

Parameters:
`nom` - Le nom de l'utilisateur à valider
Returns:
true si l'utilisateur a le droit
Throws:
`IOException` - Si le fichier n'existe pas
`SQLException` - Si la base de données n'existe pas
Since:
1.2

---

toString

```
public java.lang.String toString()
```

Deprecated. *Finalement, ça ne sert à rien!*

Méthode polymorphe `toString`. Elle renovie simplement le texte:

```
Gestionnaire d'actions
```

Overrides:
`toString` in class `java.lang.Object`
Returns:
La chaine ci-dessus

Package Class Tree Deprecated Index Help
PREV CLASS NEXT CLASS
SUMMARY: NESTED | FIELD | CONSTR | METHOD
FRAMES NO FRAMES All Classes
DETAIL: FIELD | CONSTR | METHOD

Annexe D : Prise en mains de NetBeans

Introduction

NetBeans est un environnement de développement intégré (IDE Integrated Development Environment) créé par Sun et proposé gratuitement en Open Source.

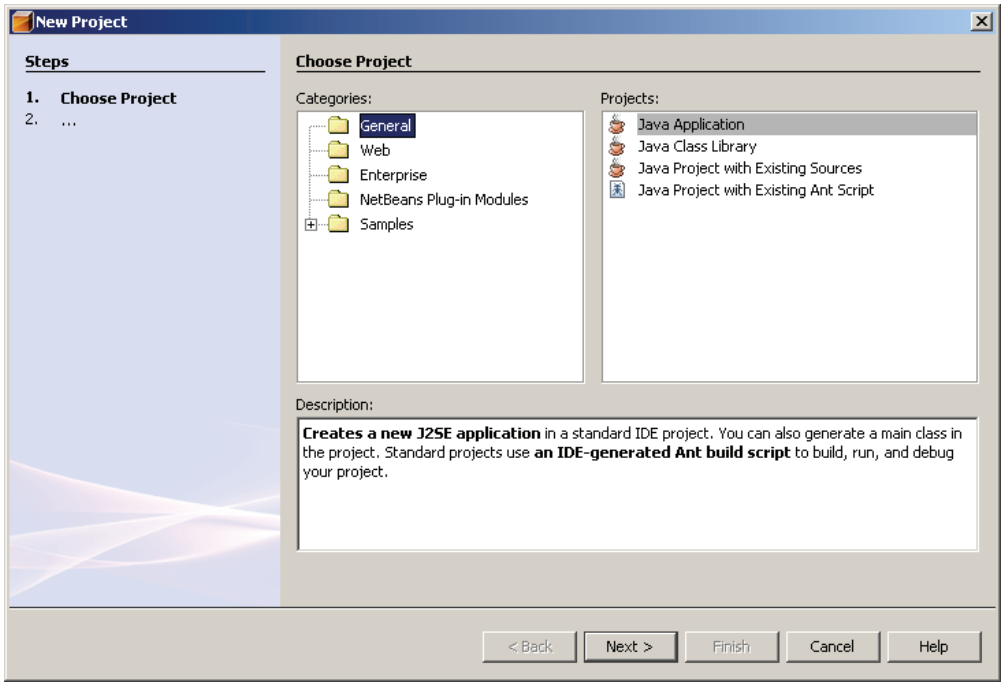
Il peut être téléchargé sur le site de Sun, à l'adresse :

http://www.netbeans.org/products/

Son installation est intuitive est simple. Reportez vous reporter à l'annexe A : Installation du poste stagiaire.

Création d'un projet

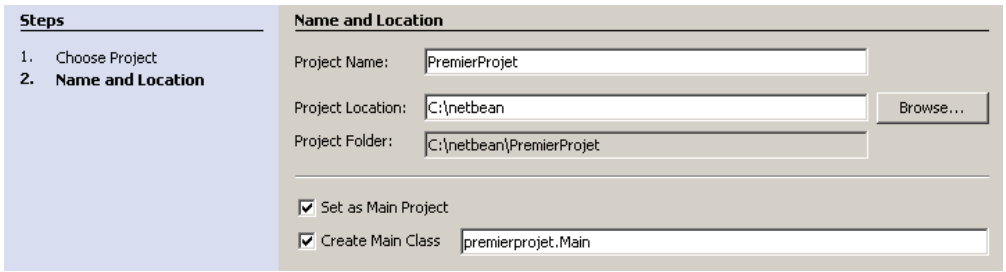
Pour créer un projet, choisir l'option menu « File / New project… ».



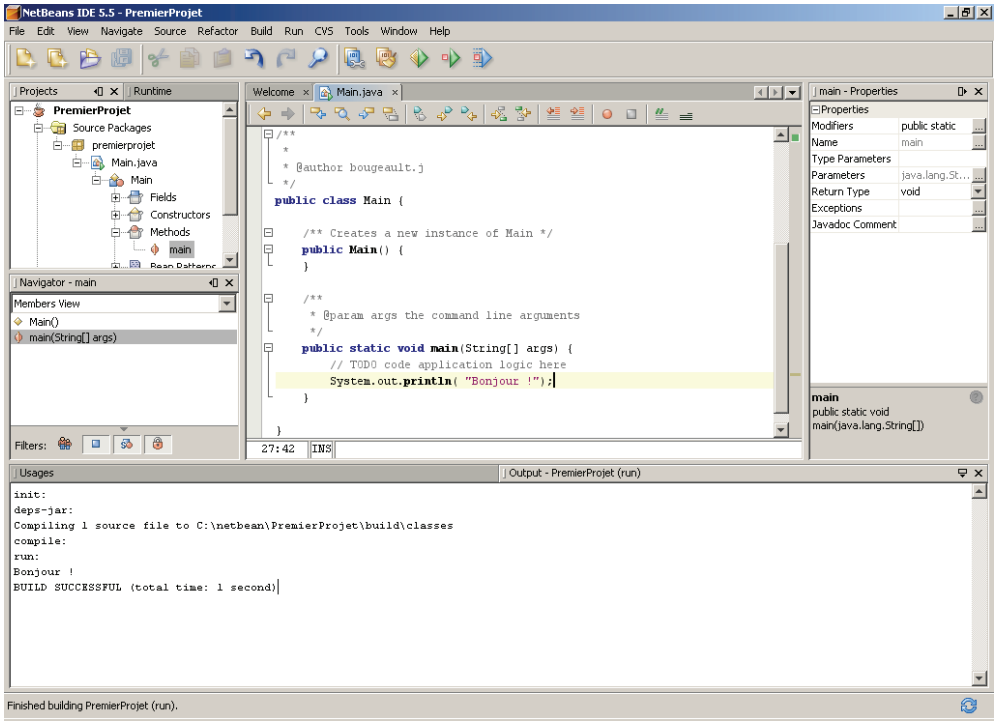
Plusieurs types d'applications peuvent être réalisés. Nous choisissons « Java Application ».

Dans l'écran suivant, il faut choisir un nom de projet. Nous entrons dans cet exemple le nom « PremierProjet ».

NetBeans nous propose par défaut un répertoire de stockage du projet, et nous propose aussi la création d'une classe principale : « premierprojet.Main ». C'est dans cette classe que se trouvera la méthode statique main.



L'ensemble du package est créé automatiquement, et on peut commencer à coder notre application. On remarquera les commentaires au format JavaDoc déjà pré-renseignés.



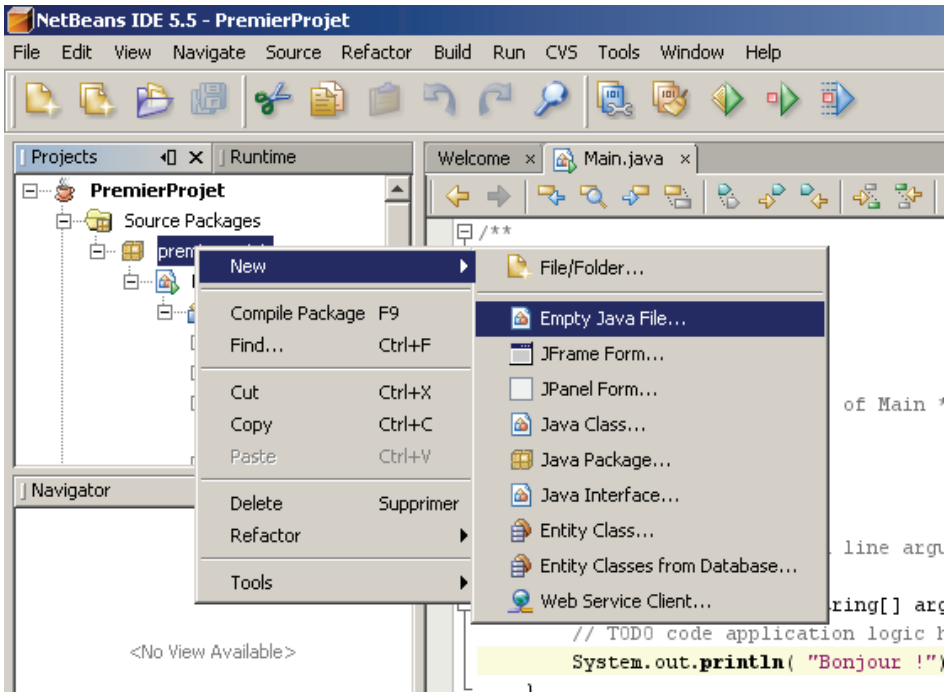
L'écran de NetBeans est composé de plusieurs parties :

- En haut à gauche, derrière l'onglet « Projects », se trouve l'ensemble des projets sur lesquels on travaille, présentés sous la forme d'une arborescence. Cette arborescence permet d'accéder à n'importe quel package, classe ou membre. En double cliquant dessus (dans notre exemple la méthode statique main), on visualise les informations la concernant dans les autres parties de l'écran.
- Au milieu à gauche, « Member View » permet de voir la liste des membres, mais on peut aussi choisi dans la liste déroulante « Inheritance view » qui permet d'avoir une vue par héritage, ce qui peut être très pratique.
- En haut au milieu, se trouve l'espace permettant d'éditer le code. Une série d'onglets, au dessus, permettent de choisir le fichier.
- En haut à droite, la fenêtre « Properties » permet de visualiser les propriétés, ce qui est particulièrement intéressant lorsque l'on travaille avec des JavaBeans.
- Enfin, en bas, s'affiche la sortie standard du programme.

Pour exécuter le programme, il faut invoquer le menu « Run / Run Main project » ou la touche de fonction F6 ou encore l'icône en forme de flèche verte sur la toolbar.

Création d'une classe

On peut créer de nouvelles classes en cliquant avec le bouton droit de la souris sur le nom d'un projet ou d'un package.



De nombreuses classes pré-remplies sont disponibles.

Dans notre cas, nous choisirons « Java class ».

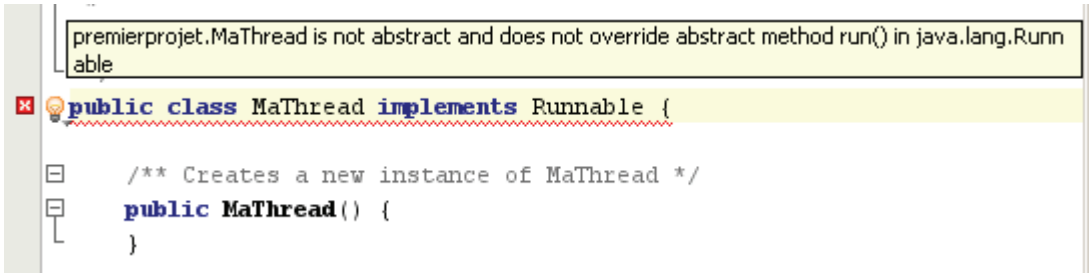
Dans l'écran qui apparaît, on peut renseigner le nom de la classe et le package dans lequel elle sera déposée.

Une fois cette classe créée, on remarque dans l'éditeur que le code de la classe, avec un constructeur sans argument par défaut, ont été prédéfinis. On peut maintenant coder.

L'éditeur Java

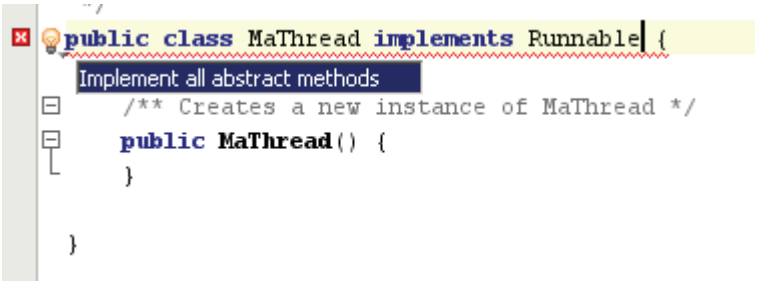
C'est un éditeur à coloration syntaxique. Il ressemble à ce que l'on a dans la plupart des IDE.

Il signale les erreurs pendant la saisie par un carré rouge à gauche de la ligne incriminée.



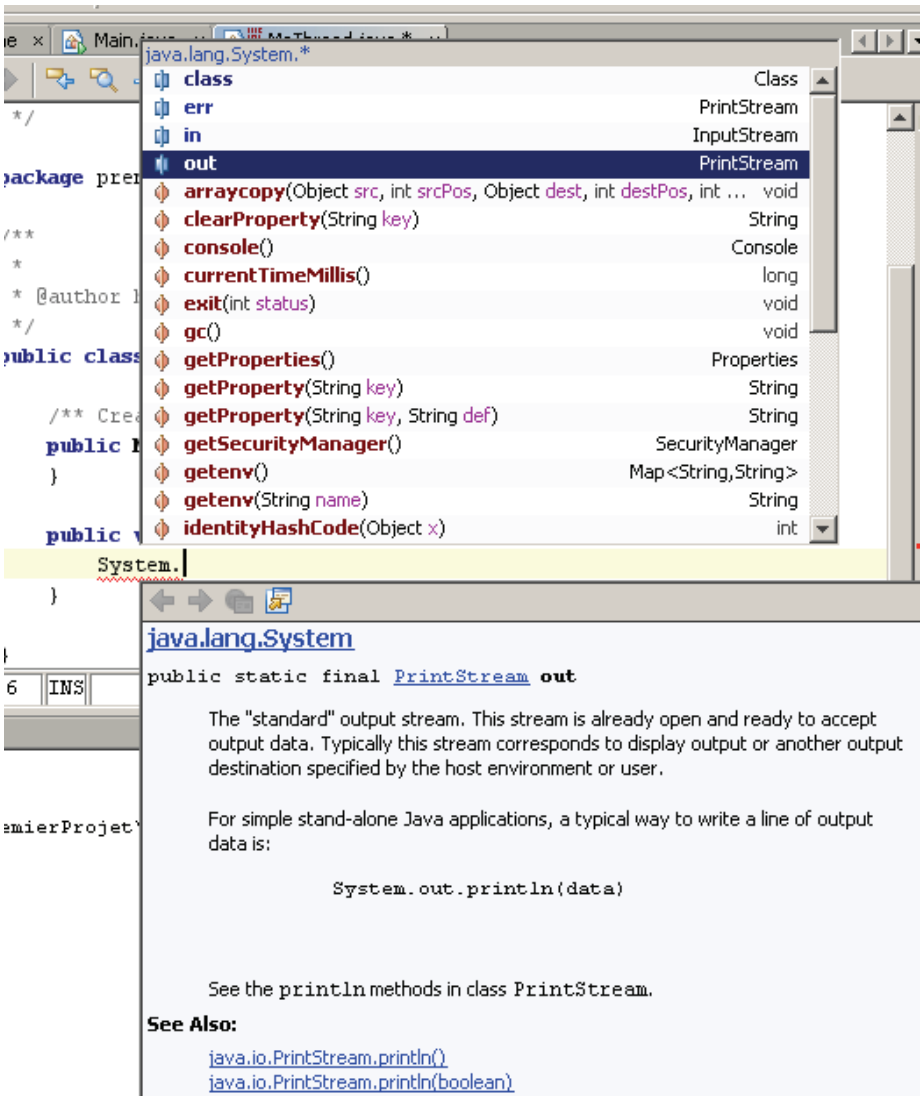
Si on passe la souris au dessus de l'erreur, alors une petite infobulle explique l'erreur. Dans notre cas, j'ai signalé que la classe MaThread implémentait Runnable, mais je n'ai pas implémenté la méthode run.

Plus pratique encore, lorsqu'il sait réparer l'erreur, l'éditeur affiche une petite ampoule jaune. Si on clique dessus, il propose alors une correction :



Dans cet exemple, il propose d'implémenter les méthodes abstraites (il n'y a que la méthode run dans le cas de l'interface Runnable). Je clique sur ce menu contextuel, alors l'implémentation d'une méthode run vide est faite automatiquement.

L'éditeur de NetBeans propose aussi une aide lors de la saisie. Dans l'exemple ci-dessous, je tape « System. » l'éditeur me propose alors la liste des membres de la classe System (dont la propriété out) en spécifiant leur type (PrintStream pour cette propriété) ainsi que la documentation Java des classes, ce qui est particulièrement intéressant lorsque l'on débute dans le développement Java, tout en étant très pratique pour les experts.



Modification des propriétés d'un projet

Le menu « File / Project properties » permet d'ouvrir la boite de dialogue des propriétés du projet.

On peut notament modifier :

- Les répertoires de stockage des sources.
- Les librairies utilisées, c'est notamment là que l'on importera d'éventuels fichiers JAR.
- Les options de compilations.
- Les options de packaging (c'est là que l'on va signaler si l'on ajoutera ou non les sources java dans le JAR de distribution).
- Les options de documentation.
- Les options d'exécution, notamment les options de la VM.

Annexe E : Access et MySQL

Introduction

Nous utilisons, pour nos travaux pratiques, deux types de bases de données: Microsoft Access avec ODBC et le moteur de bases de données MySQL.

Cette annexe permet de connaître l'installation et l'utilisation de ces moteurs.

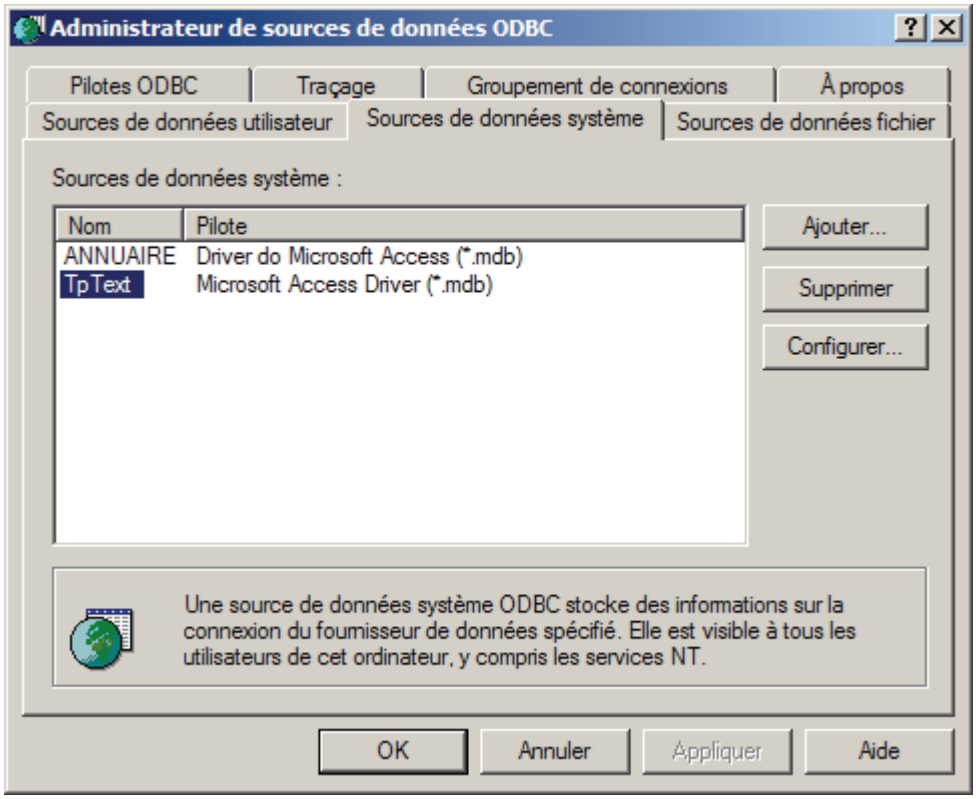
Création de bases Access avec ODBC

Access est un outil de bases de données fichier proposé par Microsoft dans sa suite bureautique: Microsoft Office.

L'application permet de créer des fichiers (dont l'extention est .mdb) et d'effectuer des requêtes pour y créer des tables, les remplir et les interroger.

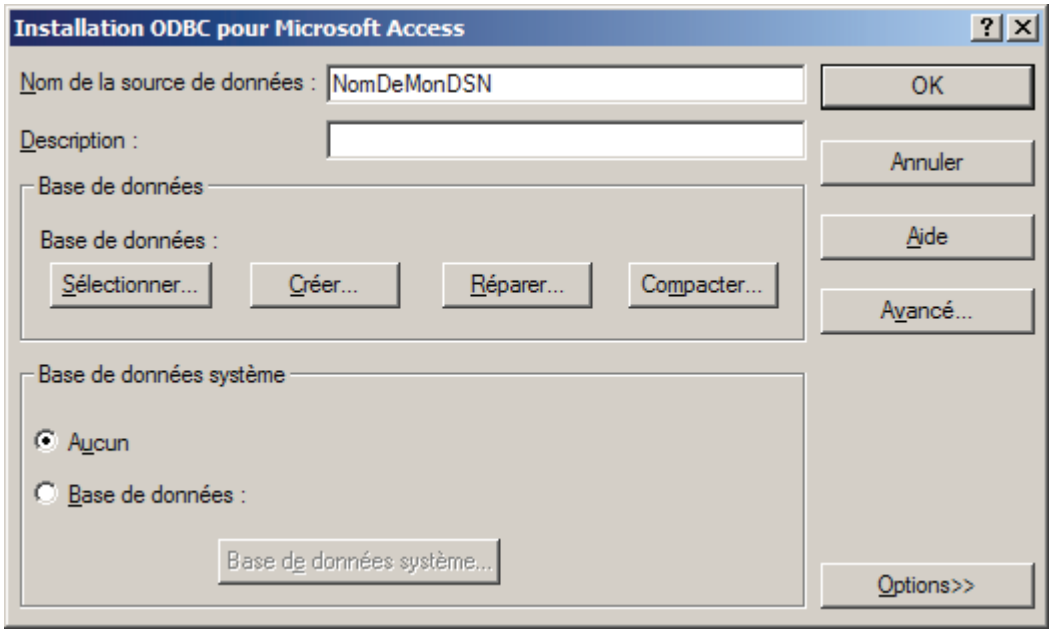
Par ailleurs, le driver Access pour ODBC permet d'accéder directement aux fichiers mdb (sans Access) pour y lancer des requêtes. Grâce à ODBC on peut donc créer des bases Access sans posséder pour autant l'application Access.

Il faut utiliser le gestionnaire de source de données ODBC, accessible dans le panneau de configuration de Window (dans certaines versions de Windows il faut aller le chercher dans les « outils d'administration »).



On se positionne sur l'onglet « Sources de données système » et on appuie sur le bouton « Ajouter ».

Dans la liste des drivers, on choisit le driver Access, et on aboutie à la fenêtre de configuration:



Il y a deux choses importantes à renseigner :

- Le nom de la source de données: Entrez le nom de votre choix.
- Le fichier de la base de données: Utiliser un fichier existant (appuyer sur le bouton « Sélectionner ») ou créer un nouveau fichier (appuyer sur le bouton « Créer… »).

Utilisation de bases Access avec JDBC

On utilise le pont JDBC/ODBC, qui est en standard dans le JDK.

Le driver se trouve dans la classe: **`sun.jdbc.odbc.JdbcOdbcDriver`**.

L'URL d'accès à une base avec ODBC nécessitera d'entrer le nom de la source de données (DSN = Data Source Name). Elle aura la forme suivante:

`jdbc:odbc:NomDuDSN`.

Exemple :

```
Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
c= DriverManager.getConnection( "jdbc:odbc:NomDuDSN");
```

Particularités de Access en Java

Le driver est assez limité :

- Pas de requêtes préparées (Attention: Le driver ne renvoie pas d'erreur lorsque l'on effectue une requête préparée!).
- Le driver est de type JDBC 1 (pas de navigation dans le ResultSet).
- Les types de données sont particuliers (**`TEXTE`**, **`NUMERIC`**, **`NUMERIC AUTO`**…).

MySQL

Nous travaillons avec la version 5 de ce moteur de bases de données. Il existe deux versions disponibles :

- La version entreprise, payante mais apportant une très grande qualité de support technique et une grande réactivité des corrections.
- La version communauté, gratuite mais sans support technique.

Nous travaillons sur la version communauté 5.0.41(win32).

Par ailleurs, nous utiliserons aussi un outil permettant de faciliter l'administration et les tests : MySQL GUI Tools.

Tous les détails pour l'installation de ces produits sont dans l'annexe A : Installation du poste stagiaire.

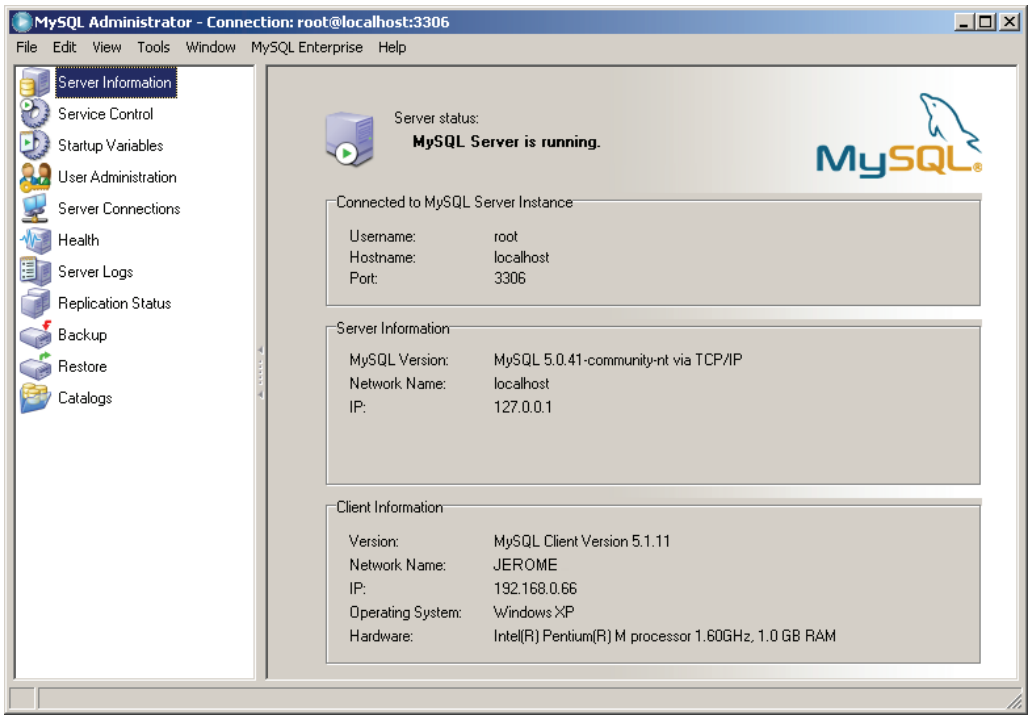
Le programme d'administration

Pour démarrer le programme d'administration, cliquer dans le menu démarrer / Programmes / MySQL / MySQL Administrator

A la boite de dialogue de connexion, entrer les informations suivantes :

- Server Host : localhost.
- Port : 3306.
- Username : root.
- Password : java (le mot de passe de root choisi lors de l'installation de MySQL).

Puis, l'interface suivante permet d'effectuer de nombreuses opérations intuitives :



En cliquant sur l'icône « Catalogs » à gauche, on peut facilement créer des tables.

La partie « server logs » permet de suivre l'utilisation des ressources système, ce qui est très utile lors de la montée en charge d'une application.

On a avec cette interface, un outil très convivial et professionnel.

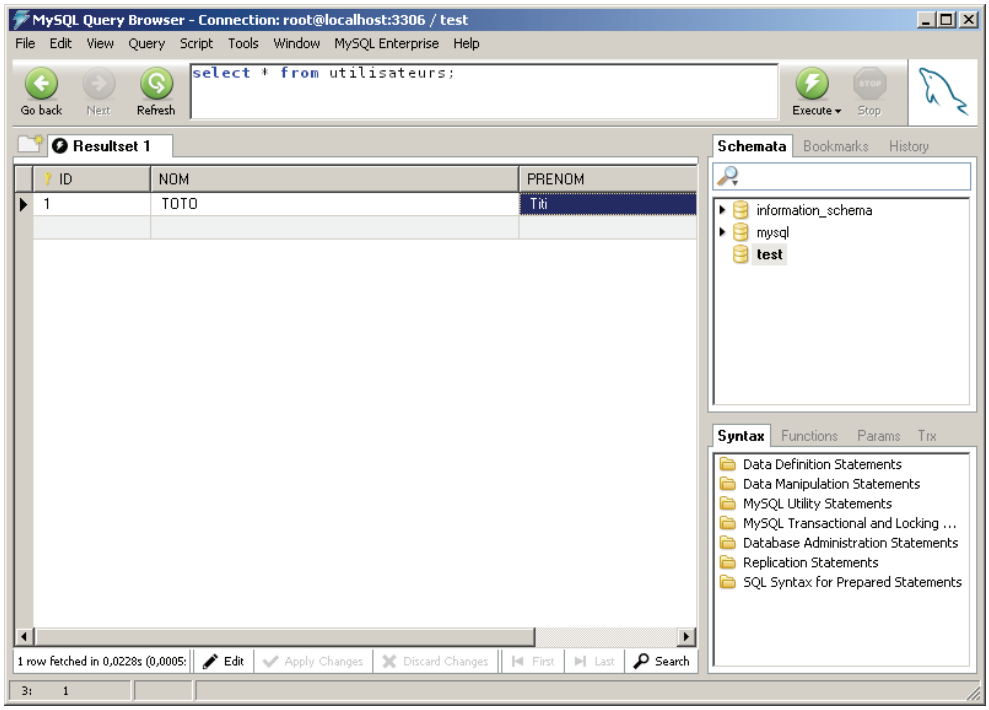
Le Query Browser

Pour démarrer le programme de gestion de requêtes, il faut choisir le programme du menu : Démarrer / Programmes / MySQL / MySQL Query Browser.

Une boite de dialogue semblable à celle de programme d'administration s'ouvre, mais dans laquelle un nouveau champ apparaît : « Default Schema ». Il s'agit de la base de données qui sera utilisée par défaut. On peut entrer par exemple le nom de la base système : « mysql ».

On notera en bas à droite une aide en ligne, très utile pour nous rappeler la syntaxe des commandes SQL.

C'est à partir de cet outil que l'on peut tester notre schéma de base de données et nos requêtes.



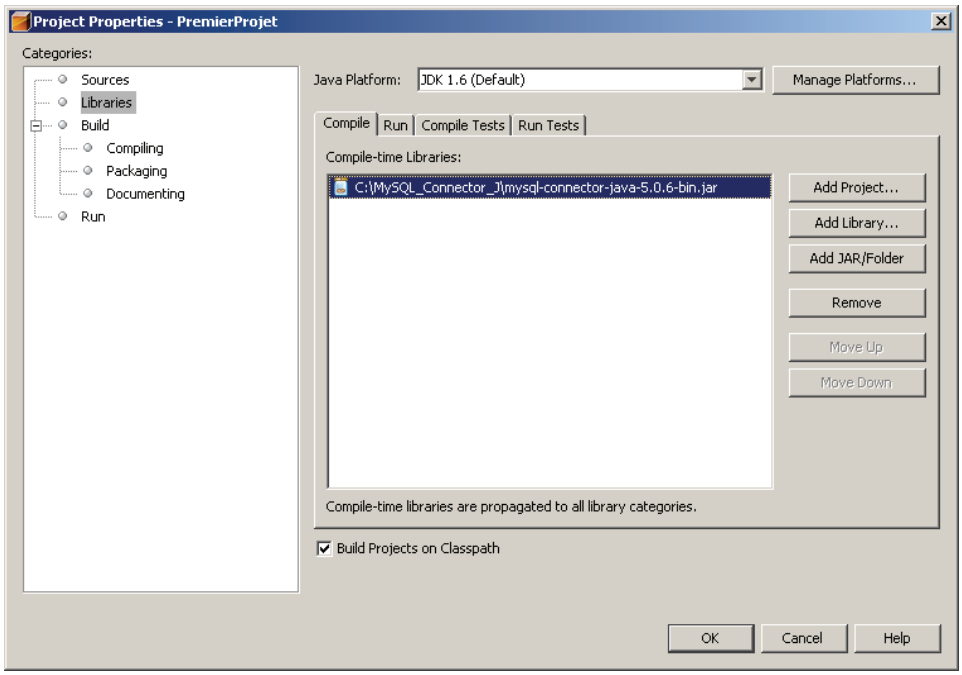
Driver JDBC

Dernier point à regler, l'installation du driver JDBC 3 de MySQL.

Il peut être téléchargé sur le site mysql.com, il s'appelle le « MySQL Connector/J », nous utiliserons la version 5.0.6. C'est un driver JDBC 3 de type 4, c'est à dire qu'il a été entièrement implémenté en Java.

Le fichier mysql-connector-java-5.0.6-bin.jar contient l'ensemble des packages du driver. Il faut simplement l'ajouter au CLASSPATH de la JVM, soit en modifiant la variable d'environnement du même nom, soit en l'ajoutant dans l'envirionnement de tests, par exemple avec NetBeans :

La fenêtre « Project properties » est accessible par le menu « File / Nom_du_projet Properties ». On aura compris dans cet exemple que le fichier JAR est stocké dans le répertoire : **`c:\MySQL_Connector_J`**.



L'utilisation de ce driver se fera simplement, comme le montre l'exemple ci-dessous :

```
   import java.sql.Connection;
   import java.sql.DriverManager;
   import java.sql.ResultSet;
   import java.sql.SQLException;
   import java.sql.Statement;
...
        try {
        Class.forName("com.mysql.jdbc.Driver").newInstance();
        } catch (IllegalAccessException ex) {
            ex.printStackTrace();
        } catch (InstantiationException ex) {
            ex.printStackTrace();
        } catch (ClassNotFoundException ex) {
            ex.printStackTrace();
        }
        try {
            Connection conn =  DriverManager.getConnection(
"jdbc:mysql://localhost/test?user=root&password=java");
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(
"SELECT * FROM utilisateurs");
            while( rs.next()) {
                System.out.println( "Nom: "
                    + rs.getString( "NOM")
```

```
                    +" Prénom: "+rs.getString( "PRENOM"));
            }
        } catch (SQLException ex) {
            System.out.println("SQLException: "
                + ex.getMessage());
            System.out.println("SQLState: "
                + ex.getSQLState());
            System.out.println("VendorError: "
                + ex.getErrorCode());
        }
```

On retiendra donc que la classe du driver MySQL est :

```
com.mysql.jdbc.Driver
```

et que l'URL de connexion a la syntaxe suivante :

```
jdbc:mysql://adresse_serveur/nom_base?user=nom&password=passe
```

Annexe F : Corrigé des exercices

Atelier 1 : Introduction

Pas de corrigé, suivre simplement les instructions données dans l'énoncé. Voir aussi la page des questions / réponses.

Atelier 2 : Éléments du langage

Exercice n° 1

Le tri à bulles :

```
/**
 * Titre : Mon premier projet<p>
 * Description : Projet bidon pour tester<p>
 * Copyright : Copyright (c) JEB<p>
 * @author JEB
 * @version 1.0
 */
public class TriBulle {
  public static void main(String[] args) {
    // Tableau à trier:
    int [] t= {34, 2, 44, 109, 12, 5, 2};

    // Tri du tableau:
    boolean permut= true;
    // Tant que l'on a permuté
    while( permut) {
      // On considère que l'on a pas permuté
      permut= false;
      for( int n=0; n < t.length -1; n++)
        // Si élément suivant plus petit on permute
        if( t[n] > t[n+1]) {
          int x= t[n];
          t[n]= t[n+1];
          t[n+1]= x;
          // Et on signale que l'on a permuté
          permut= true;
        }
    }

    // Affichage du tableau
    for( int n=0; n < t.length; n++)
      System.out.println( t[n]);
  }
}
```

Atelier 3 : Concepts objet avec Java

Exercice n° 1

Création de la classe **Date**:

```
/**
 * Cette classe représente une date
 * @author Jérôme BOUGEAULT
 * @version 1.0
 */
public class Date {
  int jour;
  int mois;
  int an;
}
```

Compiler cette classe.

Il est inutile d'essayer de l'exécuter car elle ne possède pas de méthode main.

Exercice n° 2

Création d'objets à partir de la classe **Date**:

```
/**
 * Cette classe représente une date
 * @author Jérôme BOUGEAULT
 * @version 1.0
 */
public class Date {
  int jour;
  int mois;
  int an;

  public static void main( String [] args) {
    // Déclaration de trois variables
    // pour y stocker les références
    // des trois dates
    Date d1, d2, d3;

    // Création des trois objets
    d1= new Date();
    d2= new Date();
    d3= new Date();

    // Initialisation des objets à des valeurs
    d1.jour= 24;
    d1.mois= 12;
    d1.an= 1963;
    d2.jour= 6;
    d2.mois= 7;
    d2.an= 1969;
```

```
    d3.jour= 31;
    d3.mois= 8;
    d3.an= 2001;
  }
}
```

On utilise le constructeur par défaut sans argument.

Rappel: Si aucun constructeur n'est défini dans une classe, alors un constructeur par défaut est créé automatiquement par Java. Il appelle le constructeur par défaut du père (ici la classe **Object**).

Exercice n° 3

Définition d'une variable de classe :

```
public class Date {
  int jour;
  int mois;
  int an;
  // Date dans une variable statique qui
  // représente le nouvel an
  static Date jourDeLAn;
  // Bloc static qui permet d'initialiser cette variable
  static {
    jourDeLAn= new Date();
    jourDeLAn.jour= 1;
    jourDeLAn.mois= 1;
  }
  public static void main( String [] args) {
etc...
```

On définit une variable objet de type **Date** avec le modificateur static. Sa valeur sera partagée par tous les objets de type **Date**.

Pour initialiser cette variable, il faut créer l'instance d'un objet **Date** et lui attribuer les attributs du jour de l'an (1er Janvier). On utilise ici un bloc static qui sera exécuté une seule fois lors du chargement de la classe Date dans la machine virtuelle.

Exercice n° 4

Implémentation d'une méthode **public**:

```
...
// Initialisation des objets à des valeurs
    d1.jour= 24;
    d1.mois= 12;
    d1.an= 1963;
    d2.jour= 6;
    d2.mois= 7;
    d2.an= 1969;
    d3.jour= 31;
    d3.mois= 8;
    d3.an= 2001;
```

```
    // Affichage des trois objets date:
    d1.affiche();
    d2.affiche();
    d3.affiche();
  }

  /** Cette méthode affiche la date au format français
  * sur la console Java.
  */
  public void affiche() {
    System.out.println( "Date: "+jour+" / "+mois+" / "+an);
  }
}
```

Ce qui donne à l'exécution:

```
Date: 24 / 12 / 1963
Date: 6 / 7 / 1969
Date: 31 / 8 / 2001
```

Exercice n° 5

Méthode privée de vérification de la validité de la date:

```
...
/** Cette méthode affiche la date au format français
  * sur la console Java.
  */
  public void affiche() {
    System.out.println( "Date: "+jour+" / "+mois+" / "+an);
  }
  /** Cette méthode privée vérifie si la date est bonne.
  * @return true si le contenu de l'objet est une date valide.
  */
  private boolean verifDate() {
    if( (mois < 0) || (mois > 12) )
      return false;
    if( jour < 0)
      return false;
    switch( mois) {
      case 1:
      case 3:
      case 5:
      case 7:
      case 8:
      case 10:
      case 12:
        // Ces mois ont 31 jours
        if( jour > 31)
          return false;
        break;
      case 2:
```

```
        // On considère que fevrier a 28 jours
        if( jour > 28)
          return false;
        break;
      case 4:
      case 6:
      case 9:
      case 11:
        // Ces mois ont 30 jours
        if( jour > 30)
          return false;
        break;
    }
    // Arrivé là, on a testé tous les cas d'erreurs,
    // donc la date est juste.
    return true;
  }
}
```

Exercice n° 6

Implémentation d'un constructeur :

```
...
  /** Constructeur sans argument.
  */
  public Date() {
  }

  /** Constructeur prenant en argument le jour, le mois
  * et l'année.
  * @param jour Le jour
  * @param mois Le mois
  * @param an L'année
  */
  public Date( int jour, int mois, int an) {
    this.jour= jour;
    this.mois= mois;
    this.an= an;
  }

  public static void main( String [] args) {
    // Déclaration de trois variables
    // pour y stocker les références
    // des trois dates
    Date d1, d2, d3;

    // Création des trois objets
    d1= new Date( 24, 12, 1963);
    d2= new Date( 6, 7, 1969);
    d3= new Date( 31, 8, 2001);
...
```

On remarque qu'il est nécessaire de définir un constructeur par défaut sans argument, car il y a maintenant un constructeur avec argument (celui sans argument n'est plus fait automatiquement par Java).

Exercice n° 7

Héritage. La classe **Evenement** hérite de **Date** :

```
/** La classe Evenement hérite de Date.
* Elle possède une propriété qui est la
* description de cet événement.
*/
public class Evenement extends Date {
  /** Cette propriété est le texte qui définit
  * l'événement.
  */
  public String texte;

  /** Ce constructeur crée un événement à partir des
  * propriétés d'une date et d'un texte.
  * @param jour Le jour
  * @param mois Le mois
  * @param an L'année
  * @param texte Le texte qui définit l'événement
  */
  public Evenement( int jour, int mois, int an, String texte) {
    super( jour, mois, an);
    this.texte= texte;
  }
}
```

On remarque que le constructeur fait appel au constructeur du père.

Exercice n° 8

Polymorphisme :

Implémentation des méthodes **toString** et **equals** dans **Date** :

```
...
  /** La méthode toString renvoie la date sous la forme:
  * <PRE> jour / mois / an
  * </PRE>
  */
  public String toString() {
    return jour+" / "+mois+" / "+an;
  }
  /** Cette méthode test l'égalite de l'objet avec celui passé
  * en argument.
  */
  public boolean equals( Object o) {
    // Si l'objet passé en argument n'est pas compatible
```

```
    // alors ils ne peuvent pas être égaux.
    if( !(o instanceof Date))
      return false;
    // Si la date n'est pas la même alors ils ne sont pas égaux
    Date d= (Date) o;
    if( d.jour != jour) return false;
    if( d.mois != mois) return false;
    if( d.an != an) return false;
    return true;
  }
...
```

Implémentation des méthodes **`toString`** et **`equals`** dans **`Evenement`** :

```
...
/** La méthode toString renvoie l'événement sous la forme:
  * <PRE> texte à la date du jour / mois / an
  * </PRE>
  */
  public String toString() {
    return texte+" à la date du: "+super.toString();
  }
  /** Cette méthode test l'égalite de l'objet avec celui passé
  * en argument.
  */
  public boolean equals( Object o) {
    // Si l'objet passé en argument n'est pas compatible
    // alors ils ne peuvent pas être égaux.
    if( !(o instanceof Evenement))
      return false;
    // Si la date de l'événement n'est pas la même
    // alors ils ne sont pas égaux
    if( ! super.equals( o))
      return false;
    // Si le texte de l'événement n'est pas le même
    // alors ils ne sont pas égaux
    Evenement e= (Evenement)o;
    if( ! e.texte.equals( this.texte))
      return false;
    return true;
  }
...
```

Exercice n° 9

Association :

```
/**
* Cette classe permet de trier des String
*/
public class TriBulleChaines {
```

```
  String [] ti;
  /* Le constructeur prend en argument le tableau de String
  * a trier.
  */
  public TriBulleChaines( String [] ti) {
    this.ti= ti;
  }

  public void trier() {
    boolean permut= false;
    for( int n=0; n < ti.length -1; n++)
      if( ti[n].compareToIgnoreCase( ti[n+1]) > 0) {
        String x= ti[n];
        ti[n]= ti[n+1];
        ti[n+1]= x;
        permut= true;
        }
    if( permut) trier();
  }

  public void afficher() {
    for( int n=0; n < ti.length; n++)
      System.out.println( ti[n]);
  }

  public static void main(String[] args) {
    String [] t= {"Martin", "Albert", "Jacques", "Paul",
                  "albert", "Jean", "Louis"};
    System.out.println( "Données à trier: ");
    TriBulleChaines tb= new TriBulleChaines( t);
    tb.afficher();
    tb.trier();
    System.out.println( "Données triées:");
    tb.afficher();
  }
}
```

Exercice n° 10

Les interfaces :

La classe **`TriBulleGenerique`** sait trier tout objet implémentant l'interface **`Comparable`**.

```
/**
* Tri à bulles générique<p>
*/
public class TriBulleGenerique {
  Comparable [] ti;
  // Le constructeur prend en argument le tableau de Comparable
  // a trier.
  public TriBulleGenerique( Comparable [] ti) {
```

```
    this.ti= ti;
  }
  public void trier() {
    boolean permut= false;
    for( int n=0; n < ti.length -1; n++)
      if( ti[n].compareTo( ti[n+1]) > 0) {
        Comparable x= ti[n];
        ti[n]= ti[n+1];
        ti[n+1]= x;
        permut= true;
        }
    if( permut) trier();
  }
  public void afficher() {
    for( int n=0; n < ti.length; n++)
      System.out.println( ti[n]);
  }

  public static void main(String[] args) {
    String [] t= {"Martin", "Albert", "Jacques", "Paul",
                  "albert", "Jean", "Louis"};
    System.out.println( "Données à trier: ");
    TriBulleGenerique tb= new TriBulleGenerique( t);
    tb.afficher();
    tb.trier();
    System.out.println( "Données triées:");
    tb.afficher();
  }
}
```

On récupère la classe de tri faite dans l'exercice précédent, mais on manipule maintenant des objets de type **`Comparable`** au lieu du type **`String`**.

On remarque en gras les deux seules modifications: On utilise l'interface **`Comparable`** à la place de **`String`** et la méthode **`compareTo`** à la place de **`compareToIgnoreCase`**.

L'interface **`Comparable`** de Java contient la méthode nécessaire pour comparer deux objets :

```
public interface Comparable {
  public int compareTo( Object o);
}
```

Dans le **`main`** du programme, on trie un tableau de **`String`**. **`String`** implémente **`Comparable`**, les objets de type **`String`** sont donc forcément aussi des objets de type **`Comparable`**.

Exercice n° 11

Implémentation de l'interface **`Comparable`**:

On va maintenant trier des dates. On ajoute à la classe Date l'implémentation de comparable :

```
...
public class Date implements Comparable{

...

  /** Méthode de comparaison de l'interface Comparable
  * @return 0 si l'objet passé en argument est égal, un nombre
  * négatif si elle est plus petite et un nombre positif
  * si elle est plus grande
  */
  public int compareTo( Object o) {
    if( !(o instanceof Comparable)) return -1;
    Date d= (Date)o;
    int jours= jour+(mois * 31) + (an * 366);
    int joursDeLArgument= d.jour+(d.mois * 31) + (d.an * 366);
    return joursDeLArgument - jours;
  }

...
```

Atelier 4 : Les exceptions

Exercice n° 1

Classe **Collection** :

```
/** Cette classe permet de faire des collections d'objets
* stockés dans un tableau et accessible par une interface
* à base de add et de get.
*/
public class Collection {
  /** Le tableau des objets */
  Object [] tab= new Object[100];
  private int compteur= 0;

  /** Méthode pour ajouter un élément à la collection.
  * @param element Référence de l'objet à ajouter.
  * @throws Une IndexOutOfBoundsException lorsque la
  * limite de la collection est atteinte.
  */
  public void add( Object element)
        throws IndexOutOfBoundsException {
    if( compteur >= tab.length)
      throw new IndexOutOfBoundsException(
        "Limite de la collection atteinte");
    tab[compteur]= element;
    compteur++;
```

```
  }

  /** Méthode pour récupérer un élément du tableau.
  * @param index Le numéro de l'élément indexé à partir de 0.
  * @return L'objet dont l'index est spécifié en argument.
  * @throws Une IndexOutOfBoundsException lorsque l'index
  * passé en argument est invalide (négatif ou supérieur
  * au nombre d'éléments).
  */
  public Object get( int index)
        throws IndexOutOfBoundsException {
    if( index >= compteur)
      throw new IndexOutOfBoundsException(
        "Index hors des limites de la collection");
    return tab[index];
  }

  /** Le main sert ici à tester la classe.
  */
  public static void main( String [] args) {
    Collection c= new Collection();
    try {
      c.add( "Toto");
      c.add( "Titi");
      System.out.println( c.get( 0));
      // Essayer l'une des deux lignes ci-dessous
      // en retirant le commentaire
      //System.out.println( c.get( 4));
      //System.out.println( c.get( -3));
    } catch( IndexOutOfBoundsException e) {
    System.out.println( "Exception: "+e.getMessage());
    }
  }
}
```

Dans la méthode **`get`**, deux cas peuvent provoquer l'exception :

- L'index est supérieur au nombre d'éléments dans la collection. Dans ce cas, l'exception est construite dans le programme (**`throw new IndexOutOfBoundsException( ...`**).
- L'index est inférieur à 0. C'est alors l'objet tableau qui envoie l'exception **`IndexOutOfBoundsException`**, qui est propagée vers l'appelant.

Exercice n° 2

Création d'une exception métier.

Classe d'exception CollectionException :

```
public class CollectionException
        extends Exception {
  /** Ce constructeur prend en charge le message
```

```
  * de l'exception. */
  public CollectionException( String message) {
    super( message);
  }
  /** Ce constructeur prend en charge le message
  * et la cause de l'exception. */
  public CollectionException( String message, Throwable cause)
{
    super( message, cause);
  }
}
```

On note que cette classe doit :

- Hériter de **`Exception`**
- Implémenter un constructeur qui prend en argument un message et un constructeur qui prend en argument un message et un objet **`Throwable`** qui représente la cause. Ces deux constructeurs invoquent celui du père.

L'utilisation de cette classe d'exception métier dans **`Collection`** donne pour les méthodes **`add`** et **`get`** le code suivant :

```
/** Méthode pour ajouter un élément à la collection.
  * @param element Référence de l'objet à ajouter.
  * @throws Une CollectionException lorsque la
  * limite de la collection est atteinte.
  */
  public void add( Object element)
        throws CollectionException {
    if( compteur >= tab.length)
      throw new CollectionException(
        "Limite de la collection atteinte");
    tab[compteur]= element;
    compteur++;
  }
  /** Méthode pour récupérer un élément du tableau.
  * @param index Le numéro de l'élément indexé à partir de 0.
  * @return L'objet dont l'index est spécifié en argument.
  * @throws Une CollectionException lorsque l'index
  * passé en argument est invalide (négatif ou supérieur
  * au nombre d'éléments).
  */
  public Object get( int index)
        throws CollectionException {
    if( index >= compteur)
      throw new CollectionException(
        "Index hors des limites de la collection");
    try {
      return tab[index];
    } catch( Exception e) {
      // On génère une CollectionException dont la cause
      // est l'Exception générée ici
```

```
      throw new CollectionException(
        "Index hors des limites de la collection", e);
    }
  }
```

Exercice n° 3

Affichage personnalisé des exceptions à l'aide de la classe **`StackTraceElement`**: On va définir dans CollectionException une méthode qui va afficher le détail de l'exception :

```
  public String getStackTraceElements() {
    String s= "Pile de trace de l'exception:\n";
    StackTraceElement st[]= this.getStackTrace();
    for( int n=0; n < st.length; n++) {
      s= s.concat( st[n].getClassName()+"->"
        +st[n].getMethodName()+"\n");
      s= s.concat( "\tLigne: "+st[n].getLineNumber()+"\n");
    }
    return s;
  }
```

Ajouter simplement dans le main, dans le catch de l'exception **`CollectionException`** la ligne :

```
      System.out.println( e.getStackTraceElements());
```

Atelier 5 : Classes utiles Java

Exercice n° 1

Calcul d'un prêt: Trois méthodes de simulation vont être ajoutées dans la classe **`Emprunt`** :

```
public class Emprunt {

  /** Cette méthode de classe permet de calculer le nombre de
  * mensualités à payer à partir d'un taux, d'une somme et
  * d'un nombre de mensualités.
  * @param taux Le taux de l'emprunt (par exemple pour 5% la
  * valeur de cet argument sera de 0.05
  * @param somme La somme à emprunter
  * @param mensualite Le montant de la mensualité
  */
  public static int getNombreMensualites( double taux,
          double somme, double mensualite) throws Exception {
    // D'abord on test que la mensualité est assez élevée pour
    // au moins payer le loyer de l'argent.
    if( mensualite < somme * taux / 12)
      throw new Exception( "La mensualité est trop faible!");
```

```
    if( mensualite == somme * taux / 12)
      throw new Exception( "Pas d'amortissement: in-fine");
    int nombre=0;
    while( somme > 0) {
      nombre++;
      double loyer= somme * taux / 12;
      somme -= mensualite - loyer;
    }
    return nombre;
  }


  /** Cette méthode de classe permet de calculer le
  * montant de la mensualité à payer à partir d'un taux,
  * d'un capital et d'un nombre de mensualités.
  * @param taux Le taux de l'emprunt (par exemple pour 5% la
  * valeur de cet argument sera de 0.05
  * @param somme La somme à emprunter
  * @param nombreMensualites Le nombre de mensualités à payer
  */
  public static double getMensualite( double taux,
          double somme, int nombreMensualites) {
    double p= Math.pow( 1+(taux / 12), nombreMensualites);
    return somme * p * (taux / 12) / (p-1);
  }


  /** Cette méthode de classe permet de claculer un
  * capital que l'on peut emprunter à partir d'un taux, d'une
  * mensualité d'un nombre de mensualités.
  * @param taux Le taux de l'emprunt (par exemple pour 5% la
  * valeur de cet argument sera de 0.05
  * @param mensualite Le montant de la mensualité
  * @param nombreMensualites Le nombre de mensualités à payer
  */
  public static double getMontant( double taux,
          double mensualite, int nombreMensualites) {
    double p= Math.pow( 1+(taux / 12), nombreMensualites);
    return mensualite * (p - 1) / (p * taux / 12);
  }



  /** Le main permet de tester cet objet.
  */
  public static void main( String[] args) {
    // Hypothèses pour le test:
    int mensualites= 48;
    double taux=0.0551;
    double somme= 135000;
```

```
    double montantMensualite= 1800;

    // Clacul du nombre de mensualités
    try{
      System.out.println( "Pour une somme de "+somme
        +" à un taux de "+taux
        +" en payant une mensualité de "+montantMensualite
        +" le nombre de mensualités est de: "
        +Emprunt.getNombreMensualites( taux, somme,
            montantMensualite));
    } catch( Exception e) {
      System.out.println( "Exception: "+e.getMessage());
    }

    // Calcul du montant de la mensualité
    System.out.println( "Pour une somme de "+somme
      +" à un taux de "+taux
      +" et un nombre de mensualités de: "+mensualites
      +" le montant de la mensualité est de: "
      +Emprunt.getMensualite( taux, somme, mensualites));

    // Calcul de la somme que l'on peut emprunter
    System.out.println( "Pour un nombre de mensualités de "
      +mensualites
      +" avec un montant de mensualité de "+montantMensualite
      +" et un taux de "+taux
      + " on peut emprunter la somme de: "
      +Emprunt.getMontant( taux, montantMensualite,
          mensualites));
  }
}
```

Ces trois méthodes sont statiques. Elles pourront être invoquées directement par le nom de la classe (comme c'est la cas dans le **`main`**).

Exercice n° 2

Générateur de nombres aléatoires :

La classe **`Loto`** simule le résultat des 7 boules du jeu du Loto.

```
import java.util.*;

public class Loto {

  /** Générateur de nombres compris entre 1 et 49
  */
  public static int lotoGenerator() {
    double numero= (Math.random() * 49)+1;
    return (int)numero;
  }
```

```
  /** Générateur d'un vector contenant 7 Integer
  * qui sont les nombres des 7 boules
  */
  public static Vector lotoResultat() {
    Vector boules= new Vector( 7);
    for( int n=0; n < 7; n++) {
      Integer numero;
      // On cherche un numéro qui n'est pas déjà tombé
      do
        numero= new Integer( lotoGenerator());
      while( boules.indexOf( numero) != -1);
      boules.add( numero);
    }
    return boules;
  }

  /** Test de la classe
  */
  public static void main( String [] args) {
    System.out.println( Loto.lotoResultat());
  }
}
```

On remarque dans le **`main`** le **`println`** de **`Loto.lotoResultat()`** qui affiche un **`Vector`**. La méthode **`toString`** de **`Vector`**, qui est alors automatiquement invoquée, renvoie les valeurs des éléments du **`Vector`** encadrés par des crochets. Exemple:

```
[20, 36, 45, 3, 21, 1, 2]
```

Exercice n° 3

Manipulation des chaînes de caractères.

La classe StringTool sera composée de méthodes outils pour la gestion et la manipulation des chaînes de caractères. Voici sa première :

```
/** Cette classe contient des outils pour la gestion et la
* manupulation des chaînes de caractères.
*/
public class StringTool {

  /** Cette méthode retourne le nombre d'occurences d'une
  * chaîne dans une autre chaîne.
  * @param aTrouver La chaîne que l'on recherche
  * @chaine La chaîne dans laquelle on effectue la recherche
  * @return Le nombre d'occurences
  */
  public static int nbOccurences( String aTrouver,
```

```
        String chaine) {
    int index= 0;
    int occurences= 0;
    while( (index= chaine.indexOf( aTrouver, index)) != -1) {
      index++;
      occurences++;
    }
    return occurences;
  }

  // Test de la classe
  public static void main( String[] args) {
    if( args.length < 2) {
      System.out.println( "java StringTool chaineATrouver
chaineOuTrouver");
      return;
    }
    System.out.println( "La chaine "+args[0]
      +" se trouve "
      + StringTool.nbOccurences( args[0], args[1])
      +" fois dans la chaine: "+args[1]);
  }
}
```

Exercice n° 4

Manipulation du StringTokenizer.

Toujour dans StringTool, on crée une méthode qui retourne tous les mots de la chaîne dans un tableau String.

```
...
/** Cette méthode décompose la chaîne en un tableau de mots
clés.
  * @return Un tableau de String dans lequel chaque mot est
stocké
  */
  String[] getWords() {
    StringTokenizer st= new StringTokenizer( chaine,
        " \t\f,;:!.?%&\"\'()[]=-+*/_\\");
    String[] tableau= new String[ st.countTokens()];
    int index= 0;
    while( st.hasMoreTokens()) {
      tableau[index]= st.nextToken();
      index++;
    }
    return tableau;
  }

  // Test de la classe
```

```
  public static void main( String[] args) {
    if( args.length < 1) {
      System.out.println( "java StringTool chaineADecomposer");
      return;
    }
    StringTool st= new StringTool( args[0]);
    String [] ts= st.getWords();
    for( int n=0; n < ts.length; n++)
      System.out.println( ts[n]);
...
```

Exemple d'utilisation :

```
java StringTool "re-bonjour, bonne.journée, bonne!après-midi"
re
bonjour
bonne
journée
bonne
après
midi
```

Exercice n° 5

Utilisation des dates.

Calcul des dates de valeurs sur les opérations : nous allons implémenter cette méthode dans la classe Operation.

```
import java.util.*;
/** La classe Operation représente une opération bancaire
*/
public class Operation {
  // Ces variables statiques définissent les types d'opérations
  public static final int VIREMENT_CREDIT= 1;
  public static final int VIREMENT_DEBIT= 2;
  public static final int DEPOT_CHEQUE= 3;
  public static final int DEBIT_CARTE= 4;
  // Variables d'instances:
  public int typeOperation;
  public double montant;
  public String compte;
  public Date date;

  // Constructeur
  public Operation( int type, double montant, String compte,
        Date date) {
    typeOperation= type;
    this.montant= montant;
    this.compte= compte;
    this.date= date;
  }

  public Date getDateDeValeur() {
```

```
    GregorianCalendar gc= new GregorianCalendar();
    gc.setTime( date);
    switch( typeOperation) {
      // Ajout d'un jour
      case VIREMENT_CREDIT: gc.add( Calendar.DATE, 1);
        break;
      // Soustraction d'un jour
      case VIREMENT_DEBIT: gc.add( Calendar.DATE, -1);
        break;
      // Ajout de deux jours
      case DEPOT_CHEQUE: gc.add( Calendar.DATE, 2);
        break;
      // Le premier du mois suivant
      case DEBIT_CARTE:
        gc.set( Calendar.DATE, 1);
        gc.add( Calendar.MONTH, 1);
      break;
    }
    return gc.getTime();
  }
}
```

Exercice n° 6

Les classes System et RunTime.

```
import java.util.*;

public class Proprietes {
  public static void main( String [] args) {
    // Les propriétés système
    Properties props= System.getProperties();
    Enumeration en= props.propertyNames();
    while( en.hasMoreElements()) {
      String nomProp= (String)en.nextElement();
      System.out.println( nomProp+" -> "
          +System.getProperty( nomProp));
    }
    // Récupération de l'objet Runtime de l'application
    Runtime rt= Runtime.getRuntime();
    System.out.println( "Memoire disponible: "
        +rt.freeMemory());
    System.out.println( "Mémoire totale: "+rt.totalMemory());
    System.out.println( "Nombre de processeurs: "
        +rt.availableProcessors());
  }
}
```

Exemple :

```
java -D"Mon nom=Toto titi" Proprietes
```

Donne :

```
user.variant ->
os.name -> Windows 98
sun.java2d.fontpath ->
Mon nom -> Toto titi
java.library.path ->
G:\JDK1.4\BIN;.;C:\WINDOWS\SYSTEM;C:\WINDOWS;G:\JDK1.4\BIN;
G:\JSK1.4\BIN;C:\WINDOWS;C:\WINDOWS\COMMAND
......
sun.cpu.endian -> little
sun.io.unicode.encoding -> UnicodeLittle
sun.cpu.isalist -> pentium i486 i386
Memoire disponible: 1689864
MÚmoire totale: 2031616
Nombre de processeurs: 1
```

Atelier 6 : Les entrées/sorties

Exercice n° 1

Tailles des répertoires : ce programme est récursif. Il va cumuler les tailles de tous ses fichiers, ainsi que de celles des répertoires.

```
import java.io.*;

public class TailleRepertoires {
  /** Cette méthode est appelée récursivement pour afficher
  * tous les répertoires avec leurs tailles respectives.
  * @param f File du répertoire à lister
  * @param niveau nombre qui s'incrémente à chaque entrée
  * dans un nouveau répertoire pour afficher l'indentation.
  * @return Renvoie la taille du répertoire (en tenant compte
  * aussi de la taille de ses sous répertoires).
  */
  public static long calculeTaille( File f, int niveau)
        throws IOException{
    long taille= 0;
    File[] ft= f.listFiles();
    if( ft==null) return 0;
    for( int n=0; n < ft.length; n++)
      if( ft[n].isFile()) {
        taille += ft[n].length();
      }
      else {
        // Si c'est un répertoire, on entre dedans
        // par un appel réccursif
        taille += calculeTaille( ft[n], niveau + 1);
      }
    // Affichage. On commence par l'indentation (niveau):
    for( int n=0; n< niveau; n++) System.out.print( " ");
```

```
    // Puis on affiche le nom
    System.out.print( f.getName());
    // Puis la bardée de points........:
    for( int n=0; n < 70 - (niveau + f.getName().length());
        n++)
      System.out.print( ".");
    // Enfin la taille en kilo-octets
    System.out.println( (taille/1024)+" ko");
    // On retourne la taille totale pour l'appelant
    return taille;
  }

  public static void main( String[] args) {
    File f= new File( args[0]);
    try {
      System.out.println( "Taille totale: "
          +calculeTaille( f, 0));
    } catch( IOException e) {
      System.out.println( "Erreur IO: "+e.getMessage());
    }
  }
}
```

Exercice n° 2

On fabrique dans cet exercice le fichier contenant l'ensemble des clients de l'agence. Ces clients seront générés à l'aide aléatoirement à partir de trois fichiers contenant des noms, prénoms et adresses :

nom.txt: Contient un nom de famille par ligne

prenoms.txt: Contient un prénom par ligne

adresses.txt: Contient une adresse (sans numéro) au format:

"nom de rue;code postal;ville". Le point-virgule (';') permettra d'extraire chacune des informations à l'aide d'un StringTokenizer.

Remarque:

Des fichiers sont disponibles sur le site internet de l'ouvrage. Ils contiennent des noms, prénoms et adresses tirés de l'œuvre de Victor Hugo: Les Misérables.

Format de chaque enregistrement :

Nom : CHAR(20)

Prénom : CHAR(20)

Adresse : CHAR(40)

Code Postal : INTEGER

Ville : CHAR(10)

```
import java.io.*;
import java.util.*;
/** Cette classe permet la création d'une base aléatoire
```

```
* de clients à partir de fichiers contenant des noms,
* prénoms et adresses inventés
*/
public class CreationClients {
  /** Cette méthode lit dans un tableau chaque ligne
  * d'un fichier dont le nom est spécifié en argument.
  * @return Retourne le nombre de lignes lues
  */
  public static int lit( String[] st, String fichier)
          throws IOException {
    int nbLignesLues= 0;
    FileInputStream fs= new FileInputStream( fichier);
    // Lecture par un BufferedReader pour avoir la
    // méthode readLine()
    BufferedReader b= new BufferedReader(
         new InputStreamReader( fs));
    String ligne;
    while( (ligne= b.readLine()) != null) {
      st[nbLignesLues]= ligne;
      nbLignesLues++;
    }
    return nbLignesLues;
  }
  public static void main( String[] args) {
    int nbNoms, nbPrenoms, nbAdresses;
    String[] tsNoms= new String[100];
    String[] tsPrenoms= new String[100];
    String[] tsAdresses= new String[100];

    try {
      // Lecture dans des tableaux
      nbNoms= lit( tsNoms, "NOMS.TXT");
      nbPrenoms= lit( tsPrenoms, "PRENOMS.TXT");
      nbAdresses= lit( tsAdresses, "ADRESSES.TXT");
      // Création du fichier à accès aléatoire
      RandomAccessFile rf= new RandomAccessFile( "CLIENTS.DAT",
          "rw");
      // Composition des 1000 noms et écriture en fichier
      for( int n=0; n < 1000; n++) {
        String nom= tsNoms[(int)(Math.random()*nbNoms)];
        String prenom=
          tsPrenoms[(int)(Math.random()*nbPrenoms)];
        String adresse=
          tsAdresses[(int)(Math.random()*nbAdresses)];
        // Le numéro est aléatoire entre 1 et 301.
        int numero= (int)(Math.random()*300)+1;
        // L'adresse est composée d'un nom de rue, d'un code
        // postal et d'une ville séparés par des ';'
        StringTokenizer stk= new StringTokenizer( adresse,";");
        String rue=""; int codepost=0; String ville="";
```

```
        if( stk.hasMoreTokens())
          rue= stk.nextToken();
        if( stk.hasMoreTokens())
          codepost= Integer.parseInt( stk.nextToken());
        if( stk.hasMoreTokens())
          ville= stk.nextToken();
        // On affiche à la console le résultat (pour test)
        System.out.println( nom+", "+prenom+" - "+numero+", "
            +rue+" - "+codepost+" "+ville);
        // On écrit ces données dans rf le RandomAccessFile
        ecritEtComplementeAZero( rf, nom, 20);
        ecritEtComplementeAZero( rf, prenom, 20);
        ecritEtComplementeAZero( rf, numero+", "+rue, 40);
        rf.writeInt( codepost);
        ecritEtComplementeAZero( rf, ville, 10);
      }
      // Fermeture du fichier. C'est terminé
      rf.close();
    } catch( IOException e) {
      System.out.println( "Erreur IO: "+e.getMessage());
    }
  }
  // Permet d'écrire une chaine, en la tronquant à la limite
  // de la taille du champ, ou en la complémentant à 0
  static void ecritEtComplementeAZero( RandomAccessFile rf,
      String ch, int ta) throws IOException {
    byte[] tVide= new byte[512]; // Tableau vide pour
        // complémenter les chaînes de taille inférieure
    if( ch.length() >= ta)
      rf.write( ch.getBytes(), 0, ta*2);
    else {
      rf.write( ch.getBytes( "UTF-16BE")); // Encodage UNICODE
      rf.write( tVide, 0, (ta - ch.length())*2);
    }
  }
}
```

On remarque la fonction: **`int lit(String [], String)`** qui prend en argument un tableau de chaînes de caractères et le nom du fichier à lire. Cette méthode retourne le nombre de lignes lues.

D'autre part, la méthode **`void ecritEtComplementeAZero( RandomAccessFile, String, int);`** prend en argument le fichier dans lequel écrire la chaîne de caractères passée en second argument.

Le troisième argument est le nombre de caractères maximum. Il est impératif de respecter la taille fixe de chaque enregistrement. Si la chaine est plus petite que la taille de l'enregistrement, alors elle est complémentée par des octets à zéro. Si par contre elle est plus grande, alors elle sera tronquée.

Le nombre d'octets écrits sur le fichier est le nombre de caractères multiplié par deux. En effet, l'encodage utilisé est "UTF-16BE" (UTF 16 bits en Big Endian), c'est à dire sur deux octets, l'octet de poids fort étant en fin. Il faudra utiliser le même codage à la lecture.

Exercice n° 3

Nous allons créer une classe Client qui contiendra les propriétés d'un client.

```
/** Cette classe permet de représenter un client
* sous la forme d'unb objet.
*/
public class Client {
  public String nom;
  public String prenom;
  public String adresse;
  public int codePost;
  public String ville;
}
```

Elle sera utilisée dans le programme de lecture des enregistrement.

Le programme ci-dessous permet de lire les enregistrements du fichier créé à l'aide de l'outil précédent.

Pour l'utiliser, entrer en argument le numéro de l'enregistrement (de 0 à 999).

```
import java.io.*;

/** Cette classe permet de lire le fichier des clients.
* La méthode main prend en argument le numéro de
l'enregistrement.
*/
public class LitClient {
  RandomAccessFile rf;

  /** Constructeur qui prend en argument le fichier à lire.
  * @param rd Le fichier sous la forme d'un objet de type
RendomAccesFile.
  * @throws IOException
  */
  public LitClient( RandomAccessFile rf) throws IOException{
    this.rf= rf;
  }

  /** Cette méthode lit un enregistrement et le renvoie.
  * @param codeClient Numéro de l'enregistrement.
  * @return Le client dans un objet de type Client.
  */
  public Client lectureClient( int codeClient)
            throws IOException {
    System.out.println( "lecture de l'élément en position: "
        +codeClient);
    Client c= new Client();
    rf.seek( codeClient*184); // 184: La taille enregistrement
    // Lecture de l'enregistrement
    byte[] b= new byte[40];
    rf.readFully( b);
    c.nom= new String( b, "UTF-16BE");
```

```
    b= new byte[40];
    rf.readFully( b);
    c.prenom= new String( b, "UTF-16BE");
    b= new byte[80];
    rf.readFully( b);
    c.adresse= new String( b, "UTF-16BE");
    c.codePost= rf.readInt();
    b= new byte[20];
    rf.readFully( b);
    c.ville= new String( b, "UTF-16BE");
    return c;
  }
  /** Le main prend en argument (args[0]) le numéro
  * de l'enregistrement et affiche l'enregistrement
  * correspondant sur la console Java.
  */
  public static void main( String [] args) {
    if( args.length < 1) {
      System.out.println( "java LitClient codeClient");
      System.exit( 0);
    }
    try {
      RandomAccessFile rf= new RandomAccessFile( "CLIENTS.DAT",
        "r");
      LitClient lc= new LitClient( rf);
      int codeClient= Integer.parseInt( args[0]);
      Client c= lc.lectureClient( codeClient);
      rf.close();
      // Affichage de l'élément
      System.out.println( "Nom: "+c.nom);
      System.out.println( "Prénom: "+c.prenom);
      System.out.println( "Adresse: "+c.adresse);
      System.out.println( c.codePost+" "+c.ville);
    } catch( IOException e) {
      System.out.println( "Erreur IO: "+e.getMessage());
    }
  }
}
```

La classe LitClient permet donc la récupération d'un client (classe Client) à l'aide de la méthode **`void lectureClient( int codeClient);`** Le codeClient est en fait l'index de l'élément dans le fichier (0 à 999). La méthode retourne un objet Client qu'elle a créée.

Le constructeur prend en argument le RandomAccessFile, qui devra être refermé à la fin du programme (close).

Noter la taille de l'enregistrement: 184. Cette taille se calcule comme suit:

Champ	Type	Taille (octets)
Nom	CHAR(20)	40

Prénom	CHAR(20)	40
Adresse	CHAR(40)	80
Code Postal	INTEGER	4
Ville	CHAR(10)	20
TOTAL:		184

Atelier 7 : collections d'objets

Exercice n° 1 et 2

On reprend la classe Client créée dans l'exercice précédent.

On implémente l'interface Comparable afin de pouvoir trier ces types d'objets

On y implémente un main qui va :

- Créer une instance de LitClient.
- L'utiliser pour lire tous les enregistrements.
- Stocker tous les enregistrements dans des objets Client, regroupés dans un Vector.
- Créer un tableau à partir du Vector.
- Trier ce tableau à l'aide de la classe Arrays (exit TriBulle).

```
import java.util.*;
import java.io.*;

/** Cette classe interne permet de représenter un client
* sous la forme d'unb objet.
*/
public class Client implements Comparable {
  public String nom;
  public String prenom;
  public String adresse;
  public int codePost;
  public String ville;

  /** Méthode de comparaison de l'interface Comparable
  * @param o Objet à comparer
  * @return Un entier à 0 si l'objet est identique, positif
s'il est
  * plus grand, négatif s'il est plus petit.
  */
  public int compareTo( Object o) {
    if( !(o instanceof Client))
      throw new ClassCastException( "Comparaison de Client");
    Client c= (Client)o;
    // On compare en fait deux chaînes qui son la concaténation
    // des champs à trier (nom+prenom+adresse+code postal
    String s= nom+prenom+adresse+codePost;
```

```
    return s.compareToIgnoreCase(
        c.nom+c.prenom+c.adresse+c.codePost);
  }

  /** Le main va créer le tableau de clients et le trier
  */
  public static void main( String [] args) {
    Vector v= new Vector();
    try {
      RandomAccessFile rf= new RandomAccessFile( "CLIENTS.DAT",
        "r");
      LitClient lecteur= new LitClient( rf);
      // Boucle sans fin. En fait, elle s'arrêtera lorsqu'il
      // y aura une exception I/O (arrivée en fin de fichier).
      for( int n=0;; n++) {
        Client c= lecteur.lectureClient( n);
        v.add( c);
      }
    } catch( IOException e) {
      System.out.println( "Fin de lecture: "+e.getMessage());
    }
    // On crée un tableau
    Object[] tab= v.toArray();
    // On le trie à l'aide de la méthode statique sort
    // de la classe Array
    Arrays.sort( tab);
    for( int n=0; n < tab.length; n++)
      // On affiche chaque client (cela nécessite
l'implémentation
      // de la méthode toString (ci-dessous)
      System.out.println( tab[n]);
  }

  /** Renvoie une chaine contenant les propriétés du client
  */
  public String toString() {
    return prenom+" "+nom+", "+adresse+", "+codePost+" -
"+ville;
  }
}
```

On utilise la méthode de comparaison non "case sensitive" de String. Pour simplifier, on compare des chaînes qui sont le résultat de la concaténation du nom, du prénom , de l'adresse et du code postal (la ville n'est pas nécessaire).

Atelier 8 : Java et le multi-thread

Exercice n° 1

```
import java.io.*;

/** Cette classe permet de tester l'accès à du code synchronisé
*/
public class TraceFichier extends Thread {
  /** Ce constructeur prend en argument le nom de la thread */
  public TraceFichier( String nom) {
    super( nom);
  }
  /** La méthode run qui sera exécutée dans une tâche. */
  public void run() {
    for( int n=0; n < 30; n++) {
      int nombre= (int)(Math.random()*1000);
      trace( nombre);
      yield();
    }
  }
  /** Cette méthode est synchronisée car elle fait
  * un certain nombre d'opérations d'entrées sorties qui
  * ne peuvent pas être dissociées. */
  public synchronized void trace( int nombre) {
    FileWriter out;
    try {
      out= new FileWriter( "Trace.txt", true);
      out.write( "-> "+getName()+" - "+nombre+"\r\n");
      out.close();
    } catch( IOException e) {
      System.out.println( "Erreur I/O: "+e.getMessage());
    }
  }
  public static void main( String [] args) {
    for( int n=0; n < 10; n++) {
      TraceFichier t= new TraceFichier( "Thread numéro "+n);
      t.start();
    }
  }
}
```

Exercice n° 2

On crée une première classe: ReversePipe. Elle est multi-thread et hérite de Thread.

Cette classe possède un PipedReader et un PipedWriter (elle gère des caractères).

```
import java.io.*;

/** Cette classe est un pipe dont le rôle d'inverser les
```

```
* caractères d'une chaîne de caractères (le premier
* en dernier, le dernier en premier, etc...
*/
public class ReversePipe extends Thread {

  PipedWriter pout; PipedReader pin;

  Reader in; Writer out;
  /** Le constructeur est susceptible de renvoyer
  * une IOException.
  */
  public ReversePipe() throws IOException {
    pout= new PipedWriter();
    pin= new PipedReader();
    pin.connect( pout);
    // Equivalent à: pout.connect( pin);
    out=pout;
  }

  /** Celle méthode va lancer le thread de
  * communication. */
  public void connect( Reader r) {
    in= r;
    start();
  }

  /** Accesseur sur la sortie.
  */
  public Reader getOutput() {
    return pin;
  }

  /** Code de la tâche qui effectue le traitement
  * de ce pipe.
  */
  public void run() {
    if( (in!= null) && (out!=null)) {
      try {
        PrintWriter p= new PrintWriter( out);
        BufferedReader br= new BufferedReader( in);
        String ligne;
        while( (ligne= br.readLine()) != null) {
          p.println( reverse(ligne));
          p.flush();
        }
        p.close();
      } catch( IOException e) {
        System.out.println( "IOException: "+e.getMessage());
      }
    }
```

```
  }

  /** Cette méthode inverse la chaîne de caractères qui
  * lui est passée en argument.
  */
  public String reverse( String ligne) {
    // Utilisation d'un StringBuffer
    StringBuffer sb= new StringBuffer( ligne.length());
    for( int n= ligne.length()-1; n>=0; n--)
      sb.append( ligne.charAt( n));
    return sb.toString();
  }
}
```

Le constructeur crée les PipedReader et PipedWriter et les connecte ensemble.

La méthode connect permet d'assurer la connexion (récupération du Reader)

La méthode getOutput renvoie la sortie du PipeReader.

La méthode run lit les entrées, inverse les lignes, les renvoie en sortie.

Pour le tri, on suit la même logique:

```
import java.io.*;
import java.util.*;

/** Cette classe est un pipe qui trie les lignes qui
* lui sont envoyées. */
public class SortPipe extends Thread {
  PipedWriter pout;
  PipedReader pin;
  Reader in;
  Writer out;

  /** Le constrcuteur est susceptible de renvoyer
  * une IOException. */
  public SortPipe() throws IOException {
    pout= new PipedWriter();
    pin= new PipedReader();
    pin.connect( pout);
    out=pout;
  }
  /** Méthode de connexion: Démarre le thread. */
  public void connect( Reader r) {
    in= r;
    start();
  }
  /** Accesseur du reader */
  public Reader getOutput() {
    return pin;
  }
```

```
  /** Tâche de traitement du pipe */
  public void run() {
    if( (in!= null) && (out!=null)) {
      try {
        Vector v= new Vector();
        BufferedReader br= new BufferedReader( in);
        String ligne;
        while( (ligne= br.readLine())!=null) {
          v.add( ligne);
        }
        String[] ts= new String[v.size()];
        Enumeration e= v.elements();
        int n=0;
        while (e.hasMoreElements()) {
          ts[n]= (String) e.nextElement();
          n++;
        }
        Arrays.sort(ts);
        PrintWriter p= new PrintWriter( out);
        for( n=0; n < ts.length; n++) {
          p.println( ts[n]);
        }
        p.flush();
        p.close();
      } catch( IOException e) {
        System.out.println( "IOException: "+e.getMessage());
      }
    }
  }

  public static void main( String args[]) {
    try {
      ReversePipe rp= new ReversePipe();
      ReversePipe rp2= new ReversePipe();
      SortPipe sp= new SortPipe();
      rp.connect( new InputStreamReader(System.in));
      sp.connect( rp.getOutput());
      rp2.connect( sp.getOutput());
      Reader out= rp2.getOutput();
      String ligne;
      BufferedReader br= new BufferedReader( out);
      while( (ligne= br.readLine())!= null)
        System.out.println( ligne);
    } catch( IOException e) {
      System.out.println( "IOException: "+e.getMessage());
    }
  }
}
```

On note que l'on s'appuie sur le tri implémenté dans la classe Arrays.

Le main crée deux ReversePipe et un SortPipe, les connecte entre eux, connecte l'entrée standard sur l'entrée du premier ReversePipe, puis récupère la sortie du dernier ReversePipe pour un affichage sur la sortie standard.

Exercice n° 3

Affichage à l'écran de la liste des threads et des groupes de threads :

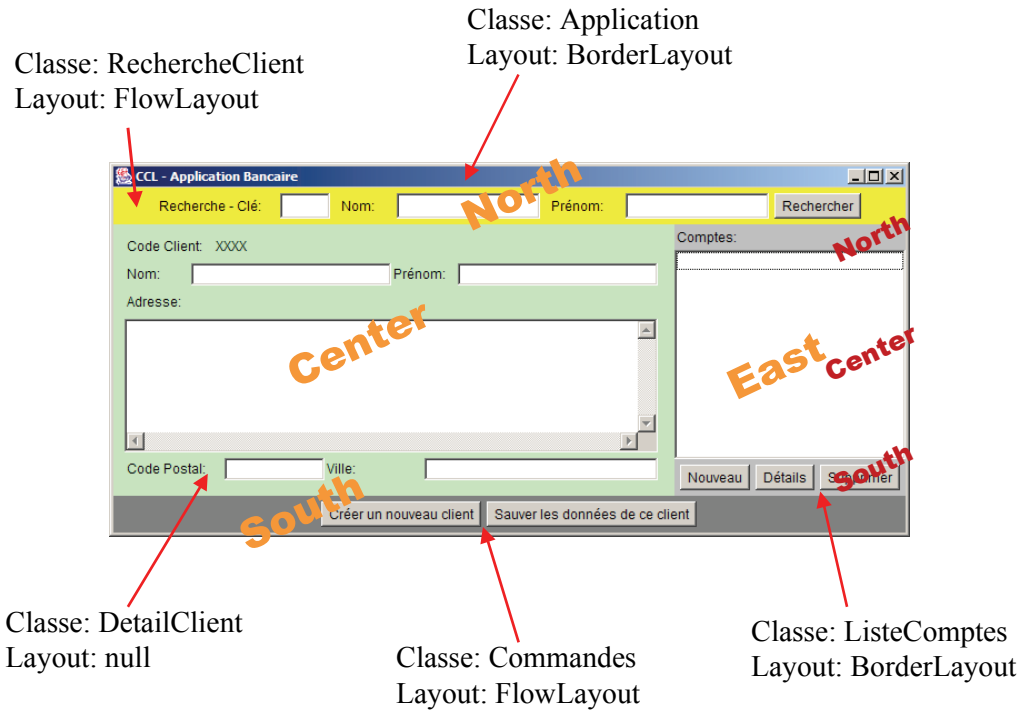
```
public class ListeThreads {

  public static ThreadGroup getRootThreadGroup() {
    Thread t= Thread.currentThread();
    ThreadGroup tg= t.getThreadGroup();
    while( tg.getParent() != null) tg=tg.getParent();
    return tg;
  }
  public static void main( String[] args) {
    System.out.println( "Liste des threads de la JVM: ");
    // Recherche de la racine des threads:
    ThreadGroup root= getRootThreadGroup();
    // Sortie sur la console des groupes et threads
    root.list();
  }
}
```

Atelier 9 : AWT et le développement d'interfaces graphiques

Exercice n° 1

Application de gestion des clients



Cette application est composée d'un ensemble de panels que l'on ve implémenter dans les classes suivantes :

- **Application** extends Frame: Frame de l'application. Elle possède un layout de type BorderLayout, avec le panel de recherche au nord, le détail du client au sud, la liste des comptes d'un client à l'est et une toolbar au sud.
- **RechercheClient** extends Panel: Cette barre de recherche possède un layout de type FlowLayout. Le bouton recherche permet de lancer une recherche soit à partir d'une clé (code client) ou bien, si ce champ est vide, à partir du nom et du prénom.
- **DetailClient** extends Panel: Ce formulaire de possède pas de layout (valeur mise à null par la méthode setLayout). Les éléments seront donc positionnés en pixels à l'aide de la méthode setBounds. On y trouve le détail du client (propriétés de l'objet de type Client: Nom, prénom, adresse, code postal et ville.
- **ListeComptes** extends Panel: Il possède un layout de type BorderLayout, avec au nord un label ("Comptes:"), au centre une liste, au sud un petit panel en FlowLayout, (en classe interne) qui comprendra trois boutons: "Nouveau" pour créer un nouveau compte, "Détails" pour voir le détail d'un compte (solde, dernières opérations, etc…) et "Supprimer" pour refermer un compte.
- **Commandes** extends Panel: Barre d'outils permettant d'accéder directement à certaines commandes: "Créer un nouveau client" et "Sauver les données de ce client".

Classe **`Application`** :

```
package ihm;
```

```
import java.awt.*;
import java.io.*;

public class Application extends Frame {

  RechercheClient rech;
  ListeComptes listeComptes;
  DetailClient det;
  Commandes cmd;
  Client[] clients= new Client[1000];

  public Application() {
    super("CCL - Application Bancaire");
    setLayout( new BorderLayout());

    // Chargement de la base des clients
    try {
      RandomAccessFile rf= new RandomAccessFile( "CLIENTS.DAT",
        "r");
      LitClient lc= new LitClient( rf);
      for( int n=0; n < 1000; n++) {
        clients[n]= lc.lectureClient( n);
      }
      rf.close();
    } catch( IOException e) {
      System.out.println( "Erreur IO: "+e.getMessage());
    }

    // Panel de recherches
    rech= new RechercheClient( clients);
    rech.setBackground( Color.yellow);
    add( "North", rech);
    // Panel de la liste des comptes
    listeComptes= new ListeComptes();
    listeComptes.setBackground( Color.lightGray);
    add( "East", listeComptes);
    // Panel des commandes
    cmd= new Commandes();
    cmd.setBackground( Color.gray);
    add( "South", cmd);
    // Panel du détail d'un client
    det= new DetailClient();
    det.setBackground( new Color( 200, 255, 200));
    add( "Center", det);
    // Initialisation de la taille
    Dimension ecran= getToolkit().getScreenSize();
    setBounds( 0, 0, ecran.width, ecran.height-30);
    setVisible( true);
  }
```

```
  public static void main( String[] args) {
    Application a= new Application();
  }
}
```

On remarquera :

- les propriétés contenant les références vers les Panels de l'application.
- la propriété clients contenant un tableau de tous les clients de l'agence. C'est sur ce tableau que se fera la recherche, ce qui explique que l'on passe sa référence au constructeur de ce panel.

Classe **RechercheClient** :

```
package ihm;

import java.awt.*;

public class RechercheClient extends Panel {
  Client[] clients;
  TextField tfCle, tfNom, tfPrenom;
  Button bpRecherche;

  public RechercheClient( Client[] Clients) {
    this.clients= clients;
    setLayout( new FlowLayout());
    add( new Label( "Recherche - Clé:"));
    tfCle= new TextField("", 3);
    add( tfCle);
    add( new Label( " Nom: "));
    tfNom= new TextField("", 15);
    add( tfNom);
    add( new Label( " Prénom: "));
    tfPrenom= new TextField("", 15);
    add( tfPrenom);
    bpRecherche= new Button( "Rechercher");
    add( bpRecherche);
  }
}
```

Elle possède en propriété :

- le tableau des clients de l'agence (pour effectuer les recherches),
- les champs de saisie et le bouton rechercher (pour effectuer la recherche). Nous verrons l'activation du bouton rechercher dans le prochain module sur les événements.

Classe **ListeComptes** :

```
package ihm;
import java.awt.*;
```

```
public class ListeComptes extends Panel {

  Button bpNouveau= new Button( "Nouveau");
  Button bpDetails= new Button( "Détails");
  Button bpSupprimer= new Button( "Supprimer");
  List liste= new List();
  Toolbar toolbar= new Toolbar();

  class Toolbar extends Panel {
    public Toolbar() {
      add( bpNouveau);
      add( bpDetails);
      add( bpSupprimer);
    }
  }

  public ListeComptes() {
    setLayout( new BorderLayout());
    add( "North", new Label( "Comptes:"));
    add( "Center", liste);
    add( "South", toolbar);
  }
}
```

Cette classe permet d'afficher la liste des comptes d'un client. Il est intéressant de voir l'utilisation d'une classe interne (classe **`Toolbar`**), uniquement destinée à l'affichage des boutons dans un **`Panel`** en **`FlowLayout`**. Les références des boutons sont directement en variables d'instance de **`ListeComptes`**.

Classe **`Commandes`** :

```
package ihm;

import java.awt.*;

public class Commandes extends Panel{

  Button bpNouveau, bpSauver;

  public Commandes() {
    setLayout( new FlowLayout());
    bpNouveau= new Button( "Créer un nouveau client");
    bpSauver= new Button( "Sauver les données de ce client");
    add( bpNouveau);
    add( bpSauver);
  }
}
```

Classe **DetailClient :**

```
package ihm;

import java.awt.*;

public class DetailClient extends Panel {
  public Label lblCodeClient;
  public TextField tfNom, tfPrenom;
  public TextArea taAdresse;
  public TextField tfCodePost, tfVille;

  public DetailClient() {
    setLayout( null); // Positionnement statique
    Label l= new Label( "Code Client:");
    l.setBounds( 10, 10, 80, 20);
    add( l);
    lblCodeClient= new Label("XXXX");
    lblCodeClient.setBounds( 90, 10, 60, 20);
    add( lblCodeClient);
    l= new Label( "Nom:");
    l.setBounds( 10, 35, 60, 20);
    add( l);
    tfNom= new TextField();
    tfNom.setBounds( 70, 35, 180, 20);
    add( tfNom);
    l= new Label( "Prénom:");
    l.setBounds( 250, 35, 60, 20);
    add( l);
    tfPrenom= new TextField();
    tfPrenom.setBounds( 310, 35, 180, 20);
    add( tfPrenom);
    l= new Label( "Adresse:");
    l.setBounds( 10, 60, 60, 20);
    add( l);
    taAdresse= new TextArea();
    taAdresse.setBounds( 10, 85, 480, 120);
    add( taAdresse);
    l= new Label( "Code Postal:");
    l.setBounds( 10, 210, 90, 20);
    add( l);
    tfCodePost= new TextField();
    tfCodePost.setBounds( 100, 210, 90, 20);
    add( tfCodePost);
    l= new Label( "Ville:");
    l.setBounds( 190, 210, 90, 20);
    add( l);
    tfVille= new TextField();
    tfVille.setBounds( 280, 210, 210, 20);
```

```
    add( tfVille);
  }
}
```

Cette implémentation s'appuie sur un positionnement absolu en pixels de chaque contrôle graphique.

Nous reviendrons sur ce TP dans le prochain module, où nous allons gérer les événements, notamment ceux des boutons.

Exercice n° 2

Construction d'un contrôle graphique: Le **`Scroller`** multi-thread en double-buffering :

```
package ihm;

import java.awt.*;

public class Scroller extends Canvas implements Runnable {
  String texte; // Texte à scroller
  int decalage; // Décalage progressif

  public Scroller( String texte) {
    setBackground( Color.yellow);
    setTexte( texte);
    Thread t= new Thread( this);
    t.start();
  }
  public void setTexte( String texte) {
    this.texte= texte;
    decalage=0;
    repaint();
  }
  public Dimension getMinimumSize() {
    return getPreferredSize();
  }
  public Dimension getPreferredSize() {
    // La taille préférée est celle du texte à afficher
    if( getFont()==null)
      setFont( new Font("Helvetica", 0, 12));
    FontMetrics fm= getFontMetrics( getFont());
    return new Dimension( fm.stringWidth( texte),
        fm.getHeight());
  }
  public void update( Graphics g) {
    if( getFont()==null)
      setFont( new Font("Helvetica", 0, 12));
    FontMetrics fm= getFontMetrics( getFont());
    int largeurImage;
    // La largeur de l'image est le plus grand entre
```

```
    // la taille de l'objet et la largeur du texte
    if( getSize().width < fm.stringWidth( texte))
      largeurImage= fm.stringWidth( texte);
    else
      largeurImage= getSize().width;
    // Construction de l'image Off Screen
    Image offScreen= createImage( largeurImage,
        fm.getHeight());
    Graphics offScreenGraphics= offScreen.getGraphics();
    // On peint le texte dans l'offScreen
    paint( offScreenGraphics);
    // On décale de 3 pixels
    if( decalage >= 3)
      decalage-=3;
    else {
      // Si on est au début, on attend 3 secondes pour
      // donner à l'utilisateur le temps de prendre un
      // peu d'avance sur le défilement
      try {
        Thread.sleep( 3000);
      } catch( InterruptedException e){}
      decalage=largeurImage;
    }
    // On affiche l'image off screen avant et
    // après le décalage (pour éviter les restes
    // de l'ancien affichage)
    g.drawImage( offScreen, decalage-largeurImage,
        0, this);
    g.drawImage( offScreen, decalage, 0, this);
  }
  public void paint( Graphics g) {
    if( getFont()==null)
      setFont( new Font("Helvetica", 0, 12));
    FontMetrics fm= getFontMetrics( getFont());
    g.drawString( texte, 0, fm.getAscent());
  }

  public void run() {
    while( true) {
      repaint(); // On repeint
      try {
        Thread.sleep(20); // Délai d'attente dans la boucle
      } catch( InterruptedException e) {
      }
    }
  }
public static void main( String [] args) {
    Frame f= new Frame(); // Une frame pour tester
    Scroller s= new Scroller( args[0]);
    f.add( "Center", s);
```

```
    f.setVisible( true);
  }
}
```

Atelier 10 : La gestion des événements

Exercice n° 1

Gestion de la fermeture de l'application. On reprend la classe Application et on utilise une classe anonyme qui hérite de **`WindowAdapter`**.

Attention à bien importer le package **`java.awt.event`**.

```
package ihm;

import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class Application extends Frame {

  RechercheClient rech;
  ListeComptes listeComptes;
  DetailClient det;
  Commandes cmd;
  Client[] clients= new Client[1000];

  public Application() {
    super("CCL - Application Bancaire");
    setLayout( new BorderLayout());

    // Gestion de la fermeture
    addWindowListener( new WindowAdapter() {
      public void windowClosing( WindowEvent e) {
        System.exit( 0);
      }
    });
...
```

Exercice n° 2

Gestion de l'action des boutons "Nouveau" et "Sauver".

La stratégie ici est de faire du panneau des détails (classe **`DetailClient`**), le listener des événements Action des boutons (classe **`Commandes`**).

On crée dans la classe **`Commande`** une méthode permettant d'abonner la fenêtre **`DetailClientqui`** devra implémenter l'interface **`ActionListener`**:

On utilisera l'"Action Command" des boutons pour reconnaître celui qui a été pressé.

Dans la classe **Application**:

```
...
    det= new DetailClient();
    det.setBackground( new Color( 200, 255, 200));
    add( "Center", det);

    // Abonnement de lu panneau détails aux événements
    // Action des boutons de commande
    cmd.abonner( det);

    // Initialisation de la taille
    Dimension ecran= getToolkit().getScreenSize();
    setBounds( 0, 0, ecran.width, ecran.height-30);
...
```

Dans la classe **Commandes** on crée la méthode abonner et on ajoute aux boutons l'"Action Command" :

```
package ihm;
import java.awt.*;
import java.awt.event.*;

public class Commandes extends Panel{
  Button bpNouveau, bpSauver;

  // Cette méthode permet d'abonner un objet aux événements
  // Action des boutons
  public void abonner( ActionListener l) {
    bpNouveau.addActionListener( l);
    bpSauver.addActionListener( l);
  }
  public Commandes() {
    setLayout( new FlowLayout());
    bpNouveau= new Button( "Créer un nouveau client");
    // L'action Command permet au listener de reconnaitre
    // quel bouton est à l'origine de l'action
    bpNouveau.setActionCommand( "NOUVEAU");
    bpSauver= new Button( "Sauver les données de ce client");
    bpSauver.setActionCommand( "SAUVER");
    add( bpNouveau);
    add( bpSauver);
  }
}
```

Il reste à implémenter l'interface **ActionListener** dans la classe **DetailClient** :

```
package ihm;

import java.awt.*;
```

```
import java.awt.event.*;

public class DetailClient extends Panel implements
ActionListener {
  public Label lblCodeClient;
  public TextField tfNom, tfPrenom;
  public TextArea taAdresse;
  public TextField tfCodePost, tfVille;

  public void actionPerformed( ActionEvent e) {
    // On vérifie bien qu'il s'agit d'un bouton
    // (et pas d'un menu ou autre...)
    if( e.getSource() instanceof Button) {
      Button b= (Button)e.getSource();
      // Est-ce le nouveau?
      if( b.getActionCommand().equals( "NOUVEAU")) {
        lblCodeClient.setText( "");
        tfNom.setText( "");
        tfPrenom.setText( "");
        taAdresse.setText( "");
        tfCodePost.setText( "");
        tfVille.setText( "");
      }
      // Est-ce le sauver?
      if( b.getActionCommand().equals( "SAUVER")) {
        System.out.println( "Client numéro: "
            + lblCodeClient.getText());
        System.out.println( "Nom: " + tfNom.getText());
        System.out.println( "Prénom: " + tfPrenom.getText());
        System.out.println( "Adresse: " + taAdresse.getText());
        System.out.println( "Code postal: "
            + tfCodePost.getText());
        System.out.println( "Ville: " + tfVille.getText());
      }
    }
  }
...
```

Exercice n° 3

Gestion du bouton "Rechercher".

Dans la classe Application, on construit les panneaux dans l'ordre suivant :

- En premier, l'objet **`DetailClient`**, à qui on passera dans le constructeur la référence un tableau des clients (afin qu'il dispose de l'accès aux données à afficher).
- En second l'objet **`RechercheClient`** à qui on passe aussi la référence du tableau des clients et la référence de l'objet **`DetailClient`** afin qu'il puisse communiquer avec lui pour afficher le détail du client trouvé lors d'une recherche.

```
...
    // Panel du détail d'un client
```

```
    det= new DetailClient( clients);
    det.setBackground( new Color( 200, 255, 200));
    add( "Center", det);

    // Abonnement de lu panneau détails aux événements
    // Action des boutons de commande
    cmd.abonner( det);

    // Panel de recherches
    rech= new RechercheClient( clients, det);
    rech.setBackground( Color.yellow);
    add( "North", rech);
...
```

Dans **RechercheClient** :

- On modifie le constructeur afin qu'il prenne en argument (et qu'il stocke en variable d'instance) le tableau des clients et la référence de l'objet **DetailClient**.
- Puis on ajoute la gestion de l'événement Action du bouton "Rechercher". La stratégie employée ici est l'usage d'une classe interne.

```
package ihm;
import java.awt.*;
import java.awt.event.*;

public class RechercheClient extends Panel {
  Client[] clients;
  DetailClient detail;
  TextField tfCle, tfNom, tfPrenom;
  Button bpRecherche;

  public RechercheClient( Client[] Clients, DetailClient det) {
    this.clients= clients;
    detail= det;
    setLayout( new FlowLayout());
    add( new Label( "Recherche - Clé:"));
    tfCle= new TextField("", 3);
    add( tfCle);
    add( new Label( " Nom: "));
    tfNom= new TextField("", 15);
    add( tfNom);
    add( new Label( " Prénom: "));
    tfPrenom= new TextField("", 15);
    add( tfPrenom);
    bpRecherche= new Button( "Rechercher");
    bpRecherche.addActionListener( new ActionListener() {
      public void actionPerformed( ActionEvent e) {
        try {
          int i= Integer.parseInt( tfCle.getText());
          // On recherche le client et on le passe à la fenêtre
```

```
          // DetailClient
          detail.setClient( i);
        } catch( NumberFormatException ex) {
          // Si la clé entrée n'est pas un nombre, on
          // efface le champ
          tfCle.setText( "");
        }
      }
    });
    add( bpRecherche);
  }
}
```

On voit que la pression sur le bouton "Rechercher" sera prise an main par la méthode **`actionPerformed`** d'une classe interne. Dans le code de cette méthode, on invoque simplement la méthode **`setClient`** dans l'objet **`DetailClient`** (on va l'implémenter juste après).

On remarque le **`try / catch( NumberFormatException)`**. Cette exception sera envoyée par le Integer.parseInt lorsque l'utilisateur entre dans le champ "clé" quelque chose qui n'est pas numérique. Dans ce cas, le champ sera vidé (**`tfCle.setText( "");`**).

On ajoute simplement dans **`DetailClient`** la méthode **`setClient`** :

```
...
// Met à jour l'ecran à partir du numéro du client
  public void setClient( int index) {
    codeClient= index;
    try {
      Client c= clients[codeClient];
      lblCodeClient.setText( ""+codeClient);
      tfNom.setText( c.nom);
      tfPrenom.setText( c.prenom);
      taAdresse.setText( c.adresse);
      tfCodePost.setText( ""+c.codePost);
      tfVille.setText( c.ville);
    } catch( IndexOutOfBoundsException e) {
      System.out.println( "Numéro de client entré invalide");
    }
  }
...
```

On utilise un **`try /catch( IndexOutOfBoundsException)`** afin de détecter des numéros de clients qui sont au delà des limites du tableau. La gestion de l'erreur devra être développée ultérieurement (on fait un simple System.out.println pour l'instant).

Exercice n° 4

```
package ihm;

import java.awt.*;
import java.awt.event.*;
```

```
public class MoletteTextField extends TextField implements
MouseWheelListener {

  public MoletteTextField() {
    // Le constructeur abonne le composant à l'événement
    // MouseWheel
    addMouseWheelListener( this);
  }

  public synchronized void mouseWheelMoved(MouseWheelEvent e) {
    int d= getSelectionStart();
    int f= getSelectionEnd();
    if( d == f) {
      // Si pas de sélection, on déplace simplement le curseur
      d= d+e.getWheelRotation();
      setSelectionStart( d);
      setSelectionEnd( d);
    }
    else {
      // S'il y a une sélection, on l'agrandit ou on le réduit
      setSelectionStart( getSelectionStart()
        +e.getWheelRotation());
      setSelectionEnd( getSelectionEnd()-e.getWheelRotation());
    }

  }

  public static void main(String[] args) {
    Frame f= new Frame();
    f.add( "North", new MoletteTextField());
    f.setVisible( true);
  }
}
```

Atelier 11 : Accès aux bases de données avec JDBC

Exercice n° 2

Programme d'interrogation universel de la base de données (ISQL) :

```
package sgbdr;

import java.sql.*;

/**
 * Cette classe permet d'interroger toute base de données qui
 * possède un driver JDBC.
 * @author Jérôme BOUGEAULT
```

```
 * @version 1.0
 */
public class ISql {
  Connection c;
  public ISql( String driver, String urlJDBC) throws Exception{
    try {
      Class.forName( driver).newInstance();
      c= DriverManager.getConnection( urlJDBC);
    } catch( ClassNotFoundException e) {
      throw new Exception( "Classe Driver non trouvée");
    } catch( IllegalAccessException e) {
      throw new Exception( "Le Driver JDBC n'est pas valide: "
        +e.getMessage());
    } catch( SQLException e) {
      throw new Exception( "Problème SQL: "
        +e.getMessage()+"\n"+e.getSQLState());
    }
  }

  // Cette méthode envoir la requête et affiche le résultat
  public void requete( String req) throws Exception{
    System.out.println( "Requête: "+req);
    try {
      Statement s= c.createStatement();
      boolean b= s.execute( req);
      // Si la requête est un select, affichage des résultats
      if( b) {
        // Analyse du ResultSet pour l'affichage
        ResultSet rs= s.getResultSet();
        ResultSetMetaData m= rs.getMetaData();
        for( int n=1; n <= m.getColumnCount(); n++)
          System.out.print( m.getColumnName(n)+"\t");
        System.out.println("");
        while( rs.next()) {
          for( int n=1; n <= m.getColumnCount(); n++)
            System.out.print( rs.getString(n)+"\t");
          System.out.println( "");
        }
      }
    } catch( SQLException e) {
      throw new Exception( "Problème SQL: "
        +e.getMessage()+"\n"+e.getSQLState());
    }
  }

  /* Le main permet de tester le programme
  */
  public static void main(String[] args) {
    if( args.length < 3) {
      System.out.println(
```

```
        "Syntaxe: java ISql driver url requete");
      System.out.println( "Exemple:");
      System.out.println(
          "java ISql sun.jdbc.odbc.JdbcOdbcDriver "
          +"jdbc:odbc:NommDSN requete");
    }
    else {
      try {
        // Création de l'objet ISql avec driver et url entrés
        // à la ligne de commande.
        ISql is = new ISql( args[0], args[1]);
        // Envoi de la requête de la ligne de commande
        is.requete( args[2]);
      } catch( Exception e) {
        System.out.println( "Problème: "+e.getMessage());
      }
    }
  }
}
```

Exercice n° 3

La requête de création de la table est la suivante :

```
create table CLIENTS (CLE INTEGER PRIMARY KEY
AUTO_INCREMENT, NOM CHAR(20), PRENOM CHAR(20), ADRESSE
CHAR(40), CODEPOST INTEGER, VILLE CHAR(10))
```

La classe **AppliClientSQL** :

```
package sgbdr;

import java.sql.*;
import ihm.Client;

/**
 * Cette classe permet d'interroger la base de données
 * contenant les clients.
 * @author Jérôme BOUGEAULT
 * @version 1.0
 */
public class AppliClientSQL {
  // La connection est créée dans le constructeur et reste
  // permanente. C'est un choix.
  Connection c;

  public AppliClientSQL() throws Exception{
    try {
      Class.forName( "com.mysql.jdbc.Driver").newInstance();
      c= DriverManager.getConnection(
```

```
      "jdbc:mysql://localhost/test");
  } catch( ClassNotFoundException e) {
    throw new Exception( "Classe Driver non trouvée");
  } catch( IllegalAccessException e) {
    throw new Exception( "Le Driver JDBC n'est pas valide: "
      +e.getMessage());
  } catch( SQLException e) {
    throw new Exception( "Problème SQL: "
      +e.getMessage()+"\n"+e.getSQLState());
  }
}


/** Retourne les données du client dont la clé est passée en
* argument.
* Envoie une exception en cas d'erreur SQL ou lorsque la clé
* est invalide.
*/
public Client getClient( int cle) throws SQLException {
  String requete=
    "select CLE, NOM, PRENOM, ADRESSE, CODEPOST, VILLE "
    +" from CLIENTS where CLE= "+cle;
  Statement s= c.createStatement();
  ResultSet rs= s.executeQuery( requete);
  if( !rs.next()) // Si pas de résultat, clé invalide
    throw new SQLException( "Pas de client à ce numéro");
  Client c= new Client();
  c.nom= rs.getString( 2);
  c.prenom= rs.getString( 3);
  c.adresse= rs.getString( 4);
  c.codePost= rs.getInt( 5);
  c.ville= rs.getString( 6);
  s.close();
  return c;
}


/** Retourne la clé du nouveau client créé. Envoir une
* exception en cas d'erreur SQL.
*/
public int nouveauClient() throws SQLException {
  String requete= "insert into CLIENTS (NOM) values ('')";
  Statement s= c.createStatement();
  s.executeUpdate( requete);
  ResultSet rs= s.getGeneratedKeys();
  if( rs.next()) {
    int cle= rs.getInt( 1);
    return cle;
  }
  else
```

```
      throw new SQLException( "Pas de clé générée");
  }



  /** Sauve les données d'un client en spécifiant sa clé
  * en argument. Envoie une exception en cas d'erreur SQL
  * ou lorsque la clé est invalide.
  * On utilise ici une requete préparée.
  */
  public void sauveClient( int cle, String nom, String prenom,
        String adresse, int codePost, String ville)
          throws SQLException {
    System.out.println( "Sauvegarde du client "+cle+", nom: "
      +nom);
    PreparedStatement ps= c.prepareStatement(
      "update CLIENTS set NOM=?, PRENOM=?, ADRESSE=?, "
      +"CODEPOST=?, VILLE=? where CLE=?");
    ps.setString( 1, nom);
    ps.setString( 2, prenom);
    ps.setString( 3, adresse);
    ps.setInt( 4, codePost);
    ps.setString( 5, ville);
    ps.setInt( 6, cle);
    ps.executeUpdate();
    ps.close();
  }
}
```

Il reste à utiliser cette classe depuis l'application.

On ne touche pas à **`Application.java`**. Tout sera géré depuis la classe **`DetailClient`**.

On va utiliser une objet de type **`AppliClientSQL`**, ce qui nécessite d'importer le package **`sgbdr`**, et de mettre cet objet en variable d'instance :

```
package ihm;

import java.awt.*;
import java.awt.event.*;
import sgbdr.*;
import java.sql.*;

public class DetailClient extends Panel implements
ActionListener {
  int codeClient;
//  Client[] clients; Cela n'est plus nécessaire
  AppliClientSQL asql;
  public Label lblCodeClient;
  public TextField tfNom, tfPrenom;
```

```
  public TextArea taAdresse;
  public TextField tfCodePost, tfVille;
...
```

Cet objet (que l'on appelle ici **asql**) sera créé dans le constructeur de la classe :

```
...
public DetailClient( Client [] clients) {
//    this.clients= clients;

    try {
      asql= new AppliClientSQL();
    } catch( Exception e) {
      System.out.println( "Erreur à la création: "
        +e.getMessage());
    }

    setLayout( null); // Positionnement statique
...
```

L'utiliser pour la création d'un nouvel enregistrement.

La méthode **actionPerformed** va maintenant, pour chacun des boutons "Nouveau" et "Sauver", utiliser les méthodes de l'objet **asql** :

```
...
public void actionPerformed( ActionEvent e) {
    // On vérifie bien qu'il s'agit d'un bouton
    // (et pas d'un menu ou autre...)
    if( e.getSource() instanceof Button) {
      Button b= (Button)e.getSource();
      // Est-ce le nouveau?
      if( b.getActionCommand().equals( "NOUVEAU")) {
        try {
          codeClient= asql.nouveauClient();
          lblCodeClient.setText( ""+codeClient);
        } catch( SQLException ex) {
          System.out.println( "Erreur: "+ex.getMessage());
        }
      }
      // Est-ce le sauver?
      if( b.getActionCommand().equals( "SAUVER")) {
        try {
          asql.sauveClient( codeClient, tfNom.getText(),
            tfPrenom.getText(), taAdresse.getText(),
            Integer.parseInt(tfCodePost.getText()),
              tfVille.getText());
        } catch( SQLException ex) {
          System.out.println( "Erreur à la sauvegarde: "
            +ex.getMessage());
        }
```

```
      }
    }
  }
...
```

Enfin, la méthode **setClient** est aussi modifée pour utiliser l'objet **asql** :

```
...
  // Met à jour l'ecran à partir du numéro du client
  public void setClient( int index) {
    codeClient= index;
    try {
      Client c= asql.getClient( index);
      lblCodeClient.setText( ""+codeClient);
      tfNom.setText( c.nom);
      tfPrenom.setText( c.prenom);
      taAdresse.setText( c.adresse);
      tfCodePost.setText( ""+c.codePost);
      tfVille.setText( c.ville);
    } catch( SQLException ex) {
      System.out.println( "Erreur SQL: "+ex.getMessage());
    }
}
...
```

Cet exercice met bien en évidence les prémices d'une architecture importante dans laquelle on distingue bien la partie IHM (classe **Application**), la partie logique métier (classe **Client**) et la partie accès aux données (classe **AppliClientSQL**).

Exercice n° 4

Utilisation des champs de grande taille :

Pour modifier la table des clients, la raquête est la suivante :

```
alter table CLIENTS add (PHOTO BLOB)
```

Le programme est dans la classe **SauveImage** :

```
package sgbdr;

import java.sql.*;
import java.io.*;

public class SauveImage {
  public static void main( String [] args) {
    if( args.length < 4) {
      System.out.println( "java SauveImage classe.du.driver "
        +" url numeroClient fichierImage");
      return;
    }
    try {
      Class.forName( args[0]).newInstance();
```

```
      Connection c= DriverManager.getConnection( args[1]);
      PreparedStatement ps= c.prepareStatement(
        "update CLIENTS set PHOTO=? where CLE=?");
      // Ouverture du fichier à charger dans la base
      File f= new File( args[3]);
      // Envoi du contenu du fichier dans la requête
      // préparée
      ps.setBinaryStream( 1, new FileInputStream( f),
        (int)f.length());
      ps.setString( 2, args[2]);
      ps.executeUpdate();
    } catch( ClassNotFoundException e) {
      System.out.println( "Classe Driver non trouvée"
        +e.getMessage());
    } catch( InstantiationException e) {
      System.out.println( "Erreur à l'instanciation: "
        +e.getMessage());
    } catch( IllegalAccessException e) {
      System.out.println( "Accès illégal: "+e.getMessage());
    } catch( SQLException e) {
      System.out.println( "Problème SQL: "
        +e.getMessage()+"\n"+e.getSQLState());
    } catch( IOException e) {
      System.out.println( "Erreur I/O: "+e.getMessage());
    }
  }
}
```

Exemple d'utilisation (à entrer à la ligne de commande sur une seul ligne) :

```
java sgbdr.SauveImage com.mysql.jdbc.Driver
jdbc:mysql://localhost/test 1 jb.jpg
```

Atelier 12 : Les JavaBeans

Exercice n° 1

Transformation du scroller de texte en JavaBean: On reprend le composant ihm.**`Scroller`** du T.P.2. On le modifie pour un faire un JavaBean, avec la possibilité de configurer :

- Le texte.
- La vitesse de défilement.
- L'arrêt et la reprise du défilement.

On n'oublie pas de faire un constructeur sans paramètres. C'est une obligation de la norme JavaBean, afin que le composant puisse-t-être créé par n'importe quel conteneur de JavaBeans.

La classe **ScrollerBean** :

```
package ihm;
import java.awt.*;
import java.io.Serializable;

public class ScrollerBean extends Canvas
    implements Runnable, Serializable {
  private int decalage; // Décalage progressif

  // Les propriétés du bean:
  private String texte; // Texte à scroller
  private int vitesse= 20; // Vitesse de défilement
  // Arrêt / reprise du défilement:
  private boolean scrolling= true;
  // On hérite des propriétés de Canvas:
  // font, foreGround et backGround

  public ScrollerBean( String texte) {
    setBackground( Color.yellow);
    setTexte( texte);
    Thread t= new Thread( this);
    t.start();
  }
  // Constructeur sans argument
  public ScrollerBean() {
    // On appelle le constructeur qui prend une
    // chaine en argument et on lui passe une
    // chaine vide.
    this( "");
  }

  // Méthodes d'accès aux propriétés:
  public String getTexte() {
    return texte;
  }
  public void setTexte( String texte) {
    this.texte= texte;
    decalage=0;
    repaint();
  }
  public int getVitesse() {
    return vitesse;
  }
  public void setVitesse( int vitesse) {
    this.vitesse= vitesse;
  }
  public boolean isScrolling() {
    return scrolling;
  }
  public void setScrolling( boolean scrolling) {
```

```
    this.scrolling= scrolling;
  }

  public Dimension getMinimumSize() {
    return getPreferredSize();
  }
  public Dimension getPreferredSize() {
    // La taille préférée est celle du texte à afficher
    if( getFont()==null)
      setFont( new Font("Helvetica", 0, 12));
    FontMetrics fm= getFontMetrics( getFont());
    return new Dimension( fm.stringWidth( texte),
        fm.getHeight());
  }

  public void update( Graphics g) {
    if( getFont()==null)
      setFont( new Font("Helvetica", 0, 12));
etc...
voir la classe Scroller
...
    // On affiche l'image off screen avant et
    // après le décalage (pour éviter les restes
    // de l'ancien affichage)
    g.drawImage( offScreen, decalage-largeurImage,
        0, this);
    g.drawImage( offScreen, decalage, 0, this);
  }
  public void paint( Graphics g) {
    if( getFont()==null)
      setFont( new Font("Helvetica", 0, 12));
    FontMetrics fm= getFontMetrics( getFont());
    g.drawString( texte, 0, fm.getAscent());
  }

  public void run() {
    while( true) {
      if( scrolling) // On test si le scrolling est actif
        repaint();
      try {
        Thread.sleep( vitesse); // La durée modifie la vitesse
      } catch( InterruptedException e) {
      }
    }
  }

  public static void main( String [] args) {
    Frame f= new Frame();
    ScrollerBean s= new ScrollerBean();
    s.setTexte(  args[0]);
```

```
    f.add( "Center", s);
    f.setVisible( true);
  }
}
```

Pour déployer ce JavaBean, on crée un fichier JAR par la commande :

jar cfm ScrollerBean.jar ScrollerBeanManifest.txt ihm/ScrollerBean.class

- **`cfm`**: Création du jar, en passant le nom du Fichier et le nom du fichier Manifest.
- **`ScrollerBean.jar`**: Nom du fichier.
- **`ScrollerBeanManifest.txt`**: nom du fichier manifest qui se trouve dans le répertoire courant.
- **`ihm/ScrollerBean.class`**: nom de la classe du JavaBean.

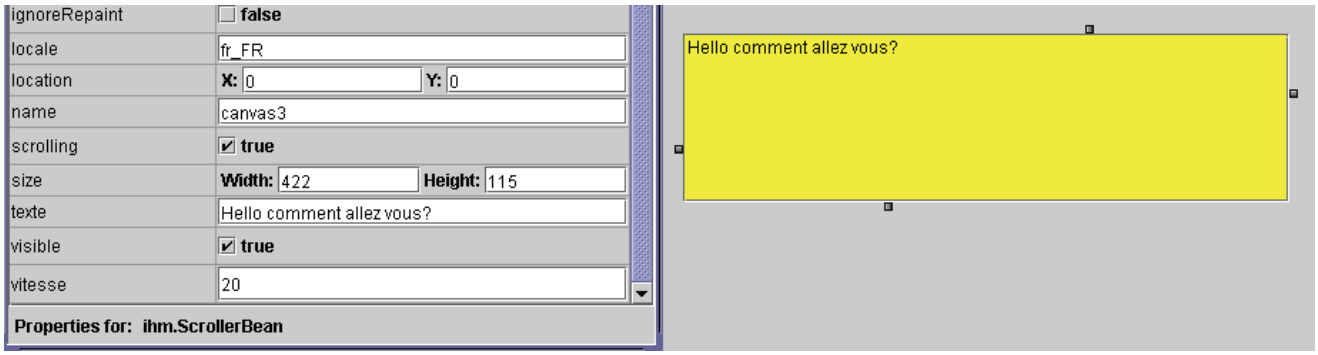
Le fichier **`ScrollerBeanManifest.txt`** est le suivant:

```
Manifest-Version: 1.0
Name: ihm/ScrollerBean.class
Java-Bean: True
Created-By: J.Bougeault
```

Pour tester le JavaBean dans le BeanBuilder, on sélectionne dans la palette l'option menu "File / Load JAR File…", puis son sélectionne le fichier JAR que l'on vient de créer.

Normalement, on doit voir apparaître dans la palete utilisateur (onglet "User") notre JavaBean. S'il n'existe pas c'est qu'il n'est pas conforme.

On peut alors simplement l'instancier dans le conteneur :



On voit bien dans la fenêtre des propriétés les champs permettant de modifier les propriétés, notamment celles que l'on a implémenté: scrolling, texte, visible et vitesse.

Exercice n° 2

On modifie la méthode **`main`** de la classe **`ihm.ScrollerBean`** pour y implémenter la sérialisation et la désérialisation :

```
...
public static void main( String [] args) {
    if( args.length < 1) {
      System.out.println( "java ScrollerBean NomDeFichier"
        +" \"Texte à scroller\"");
```

```
      System.out.println( "Ou en lecture:");
      System.out.println( "java ScrollerBean NomDeFichier");
      System.exit( 0);
    }
    ScrollerBean s= null;
    if( args.length == 2) { // On est en écriture
      s= new ScrollerBean();
      s.setTexte(  args[1]);
      try {
        ObjectOutputStream out= new ObjectOutputStream(
          new FileOutputStream( args[0]));
        out.writeObject( s);
      } catch( IOException e) {
        System.out.println( "Sérialisation échouée: "
          +e.getMessage());
      }
    }
    if( args.length == 1) { // On est en lecture
      try {
        ObjectInputStream in= new ObjectInputStream(
          new FileInputStream( args[0]));
        s= (ScrollerBean)in.readObject();
      } catch( IOException e) {
        System.out.println( "Problème de désérialiation: "
          +e.getMessage());
      } catch( ClassNotFoundException e) {
        System.out.println( "Classe non trouvée: "
          +e.getMessage());
      }
    }
    Frame f= new Frame();
    if( s!= null)
      f.add( "Center", s);
    f.setVisible( true);
  }
}
```

Lancer le programme en testant les deux cas. D'abord en écriture, par exemple :

`java ihm.ScrollerBean test.ser "Texte qui va scroller"`

puis en lecture :

`java ihm.ScrollerBean text.set`

On remarque que en lecture, le texte ne scrolle pas. En effet, la thread n'a pas été relancée.

Exercice n° 3

La solution au problème constaté dans le précédent exercice lors de la désérialisation est de relancer la thread de scrolling lorsque l'objet est réinstancié en mémoire. Pour cela, il faut implémenter l'interface **`Externalizable`**.

Les deux méthodes de l'interface (readExternal et writeExternal) doivent être implémentées, et gérer l'ensemble des propriétés à rendre persistantes, ainsi que le redémarrage de la thread lors de la désérialisation :

```
package ihm;

import java.awt.*;
import java.io.*;

public class ScrollerBean extends Canvas
    implements Runnable, Externalizable {
  private int decalage; // Décalage progressif
...
// Les méthodes de l'interface Externalizable:
  public void readExternal( ObjectInput in)
      throws IOException, ClassNotFoundException {
    texte= (String) in.readObject();
    vitesse= ((Integer)in.readObject()).intValue();
    scrolling= ((Boolean)in.readObject()).booleanValue();
    // Redémarrage de la thread
    Thread t= new Thread( this);
    t.start();
  }
  public void writeExternal( ObjectOutput out)
      throws IOException {
    out.writeObject( texte);
    out.writeObject( new Integer(vitesse));
    out.writeObject( new Boolean(scrolling));
  }
...
```

Atelier 13 : JFC et Swing

Exercice n° 1

Implémentation de TreeModel pour afficher l'arborescence des fichiers.

La classe FileTreeModel :

```
package outils;

import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.event.*;
```

```
import java.awt.event.*;
import java.util.*;
import java.io.*;

/** Exemple d'implémentation du TreeModel
* sur le système de gestion de fichiers du système
* d'exploitation.
* @author Jérôme BOUGEAULT
* @version 1.0
*/

public class FileTreeModel implements TreeModel {

  // Ceci est le gestionnaire de listeners dévénements
  // "List" sur lequel on s'appuie
  protected EventListenerList listeners;
  // Ceci est la racine qui contient l'ensemble des unités
  private Vector racine;

  /** Le constructeur ne prend pas d'argument. Il affiche
  * toutes les unités disponibles dans le système
  */
  public FileTreeModel() {
      listeners = new EventListenerList();
      racine = new Vector();
      File[] racFs;
      racFs= File.listRoots();
      // racine est un Vector qui contient toutes les
      // unités.
      for( int n=0; n < racFs.length; n++)
          racine.add(racFs[n]);
  }
  /**
  * Renvoie la racine qui est le Vector qui contient
  * la liste de toutes les unités.
  */
  public Object getRoot() {
      return racine;
  }
  /** Renvoie true si c'est un fichier, false si c'est
  * un répertoire.
  */
  public boolean isLeaf(Object node){
    // Si c'est une instance de Vector, c'est que
    // c'est la racine, donc pas une fichier
    if( node instanceof Vector)
      return false;
    else {
      // Sinon, c'est que c'est un objet File
      File f= (File) node;
```

```
    if( f.isDirectory())
      return false;
    else
      return true;
  }
}
/** Renvoie le nombre d'enfants d'un noeud
*/
public int getChildCount(Object node) {
  // Si c'est une instance de Vector, c'est que
  // c'est la racine, on renvoie le nombre d'éléments
  // du Vector, c'est à dire le nombre d'unités.
  if( node instanceof Vector)
    return ((Vector)node).size();
  else {
    File f= (File) node;
    if( f.isDirectory()) {
      // f est un tableau de File
      return f.listFiles().length;
    }
    else
      return 0;
  }
}

/** Renvoie e fils dont l'index est passé en argument
*/
public Object getChild(Object parent, int index) {
  if( parent instanceof Vector)
    return ((Vector)parent).get( index);
  else {
    File f= (File) parent;
    if( f.isDirectory()) {
      return f.listFiles()[index];
    }
    else
      return null;
  }
}

/** Renvoie l'index du fils passé en argument
*/
public int getIndexOfChild(Object parent, Object child) {
  if( parent instanceof Vector) {
    return ((Vector)parent).indexOf( child);
  }
  else {
    File f= (File) parent;
    if( f.isDirectory()) {
      Object[] tableau= f.listFiles();
```

```
        for( int n=0; n < tableau.length; n++)
          if( tableau[n].equals( child))
            return n;
        // Renvoie -1 si le fils n'est pas trouvé
        return -1;
      }
      else
        return -1;
    }
  }
  public void valueForPathChanged(TreePath path, Object value){
  }
  // La gestion des événements est délégué
  public void addTreeModelListener(TreeModelListener l) {
    listeners.add(TreeModelListener.class, l);
  }
  public void removeTreeModelListener(TreeModelListener l) {
    listeners.remove(TreeModelListener.class, l);
  }

  public static void main(String[] args) {
    JTree tree = new JTree(new FileTreeModel());
    JFrame f = new JFrame("Arborescence fichiers");
    f.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });
    f.getContentPane().add(new JScrollPane(tree));
    f.pack();
    f.setVisible(true);
  }
}
```

Exercice n° 2

Mini éditeur HTML :

```
package outils;

import javax.swing.*;
import javax.swing.event.*;

/**
* Cette classe est un outil sommaire pour créer des pages HTML
* @author Jérôme BOUGEAULT
* @version 1.0
*/
public class HTMLEditor extends JFrame implements
ChangeListener {
  // Les composants graphiques:
```

```
  JTabbedPane tab;
  JEditorPane htmlEdit;
  JEditorPane codeEdit;
  String texte;

  /** Constructeur sans argument
  */
  public HTMLEditor() {
    super ("Editeur HTML");
    texte= "<HTML><HEAD><TITLE>Titre</TITLE></HEAD>"
      +"<BODY>texte</BODY></HTML>";
    // JEditorPane pour l'affichage HTML:
    htmlEdit= new JEditorPane( "text/html", texte);
    // JEditorPane pour l'affichage du code:
    codeEdit= new JEditorPane( "text/plain", texte);
    tab= new JTabbedPane();
    // Abonnement pour être prévenu lorsque l'utilisateur passe
    // d'un panneau à l'autre:
    tab.addChangeListener( this);
    // Ajout des deux panneaux au JTabbedPane:
    tab.add( "WYSIWYG", new JScrollPane(htmlEdit));
    tab.add( "CODE", new JScrollPane( codeEdit));
    getContentPane().add( "Center", tab);
  }

  /** Cette méthode est importante, elle est invoquée par le
  * ChangeEvent lorsque l'utilisateur passe d'un
  * panneau à l'autre du JTabbedPane.
  * C'est là que l'on gère la mise à jour du panneau qui
  * prend la main.
  */
  public void stateChanged( ChangeEvent e) {
    if( tab.getSelectedIndex() == 0) {
      texte= codeEdit.getText();
      // On met le code HTML dans htmlEdit
      htmlEdit.setText( texte);
    }
    if( tab.getSelectedIndex() == 1) {
      texte= htmlEdit.getText();
      // On met le code HTML dans codeEdit
      codeEdit.setText( texte);
    }
  }

  public static void main( String [] args) {
    HTMLEditor html= new HTMLEditor();
    html.setBounds( 0, 0, 600, 400);
    html.setVisible( true);
  }
}
```

Exercice n° 3

Outil graphique d'interrogation des bases de données JDBC.

On va créer une modèle de table spécifique à la structure d'un ResultSet de JDBC :

```
package sgbdr;

import javax.swing.*;
import javax.swing.table.*;
import java.sql.*;

/**
* Cette classe est une implémentation de TableModel pour
afficher
* dans une JTable un résultat JDBC de type ResultSet
* @author Jérôme BOUGEAULT
* @version 1.0
*/
public class ResultSetTableModel extends AbstractTableModel {
  // Le résultat
  private ResultSet resultat;

  /**
  * Renvoie le nombre de colonnes du ResultSet
  */
  public int getColumnCount() {
    if( resultat == null)
      return 0;
    try {
      ResultSetMetaData m= resultat.getMetaData();
      return m.getColumnCount();
    } catch( SQLException e) {
      return 10;
    }
  }

  /**
  * Renvoie le nombre de lignes du ResultSet
  */
  public int getRowCount() {
    if( resultat == null)
      return 0;
    try {
      resultat.last();
      return resultat.getRow();
    } catch( SQLException e) {
      return 100;
    }
  }
```

```
/**
* Renvoie la valeur de la ligne et de la colonne du
*ResultSet.
* Le résultat est toujours de type String.
* Le résultat est "NULL" si la cellule est à null dans la
* base.
*/
public Object getValueAt( int ligne, int colonne) {
  if( resultat == null)
    return "";
  try {
    resultat.absolute( ligne+1); // On ajoute 1 car le numéro
    // de la ligne est indexé à partir de 1 et non pas de 0
    String valeur= resultat.getString( colonne+1); // On
    // ajoute aussi 1 pour la même raison.

    // On teste si le résulat est null, dans ce cas on
    // affiche "NULL" dans la cellule
    if( resultat.wasNull())
      valeur= "NULL";
    return valeur;
  } catch( SQLException e) {
    return "<Error>";
  }
}


/**
* Cette méthode retourne toujours false car on ne permet pas
* d'éditer la ResultSet.
*/
public boolean isCellEditable( int ligne, int colonne) {
  return false;
}


/**
* Cette méthode permet de retourner un nom à chaque colonne.
* Ce nom est celui utilisé dans la requête.
*/
public String getColumnName( int numero) {
  if( resultat == null)
    return ""+numero;
  try {
    ResultSetMetaData m= resultat.getMetaData();
    return m.getColumnName( numero + 1);
  } catch( SQLException e) {
    return "<Error>";
  }
}
```

```
  /**
  * Cette méthode permet de spécifier un nouveau ResultSet,
  * lorsque l'on a fait une nouvelle requête.
  */
  public void setResultSet( ResultSet rs) {
    resultat= rs;
    // Il faut prévenir la JTable que la structure de la table
    // a changée:
    fireTableStructureChanged();
    // Cette méthode est implémentée dans AbstractTableModel
    // dont on hérite, et lance l'événement TableModelEvent.
  }
}
```

Atelier 14 : programmation internet et réseaux

Exercice n° 1

Applet de simulation de prêt :

AppletEmprunt hérite de JApplet, contient le JTabbedPane dans lequel on trouve les trois panneaux de simulation.

Code de AppletEmprunt :

```
package ihm.emprunt;

import javax.swing.*;

public class AppletEmprunt extends JApplet {
  CombienDeTemps ct= new CombienDeTemps();
  QuelleSomme qs= new QuelleSomme();
  CombienParMois cm= new CombienParMois();
  JTabbedPane jt= new JTabbedPane();

  /** Dans l'init, on remplie le JTabbedPane
  * et on le met dans le contentPane de l'applet
  */
  public void init() {
    jt.add( "Combien de temps", ct);
    jt.add( "Quelle somme", qs);
    jt.add( "Combien par mois", cm);
    getContentPane().add( "Center", jt);
  }

  /** Le main permet de tester l'applet, mais on
  * peut aussi l'utiliser dans l'AppletViewer
  */
```

```
  public static void main( String [] args) {
    JFrame f= new JFrame();
    AppletEmprunt ap= new AppletEmprunt();
    ap.init();
    f.getContentPane().add( "Center", ap);
    f.setBounds( 100, 100, 400, 200);
    f.setVisible( true);
  }
}
```

La classe CombienParMois :

```
package ihm.emprunt;

import outils.Emprunt; // Classe des méthodes de calcul
import javax.swing.*;
import javax.swing.border.BevelBorder;
import java.awt.Dimension;
import java.awt.event.*;

public class CombienParMois extends JPanel
      implements ActionListener{
  JTextField tfTaux= new JTextField();
  JTextField tfMontant= new JTextField();
  JTextField tfDuree= new JTextField();
  JLabel lblMensualite= new JLabel(
      "    Montant de la mensualité: 0 €    ");
  JButton bpCalculer= new JButton( "Calculer");

  /** Dans le constructeurn on construit l'IHM à base de
  * conteneurs Box
  */
  public CombienParMois() {
    // Le Box qui contient la saisie du taux
    Box bTaux= Box.createHorizontalBox();
    bTaux.add( new JLabel( "Taux: "));
    bTaux.add( tfTaux);
    bTaux.add( new JLabel( " % "));
    // Le Box qui contient la saisie du montant
    Box bMontant= Box.createHorizontalBox();
    bMontant.add( new JLabel( "Montant: "));
    bMontant.add( tfMontant);
    bMontant.add( new JLabel( " € "));
    // Le Box qui contient la saisie du nombre de mensualités
    Box bDuree= Box.createHorizontalBox();
    bDuree.add( new JLabel( "Durée: "));
    bDuree.add( tfDuree);
    bDuree.add( new JLabel( " mois "));
    // Le Box qui contient l'affichage du montant de la
    // mensualité
```

```
    Box bMensualite= Box.createHorizontalBox();
    bMensualite.setBorder( new BevelBorder(
      BevelBorder.LOWERED));
    bMensualite.add( Box.createHorizontalGlue());
    bMensualite.add( lblMensualite);
    bMensualite.add( Box.createHorizontalGlue());

    Box principal= Box.createVerticalBox();
    principal.add( bTaux);
    principal.add( bMontant);
    principal.add( bDuree);
    principal.add( bMensualite);
    principal.add( bpCalculer);

    add( principal);

    bpCalculer.addActionListener( this);
  }

  /** Lorsque l'utilisaeur appuie sur le bouton,
  * on procède au calcul en utilisant la méthode
  * statique de la classe Emprunt
  */
  public void actionPerformed( ActionEvent e) {
    // On lance le calcul:
    try {
      double mensualite= Emprunt.getMensualite(
        Double.parseDouble( tfTaux.getText())/100,
        Double.parseDouble( tfMontant.getText()),
        Integer.parseInt( tfDuree.getText()));
        lblMensualite.setText("    Montant de la mensualité: "
          +java.lang.Math.floor(mensualite)+"  €    ");

    } catch( Exception ex) {
      lblMensualite.setText(
        "    Données saisies incorrectes   ");
    }


  }
}
```

Les deux autres classes sont sur le même modèle.

La classe CombienDeTemps :

```
package ihm.emprunt;

import outils.Emprunt; // Classe des méthodes de calcul
import javax.swing.*;
```

```
import javax.swing.border.BevelBorder;
import java.awt.Dimension;
import java.awt.event.*;

public class CombienDeTemps extends JPanel
      implements ActionListener{
  JTextField tfTaux= new JTextField();
  JTextField tfSomme= new JTextField();
  JTextField tfMensualite= new JTextField();
  JLabel lblDuree= new JLabel(
    "    Durée de remboursement: 0 mois    ");
  JButton bpCalculer= new JButton( "Calculer");

  /** Dans le constructeurn on construit l'IHM à base de
  * conteneurs Box
  */
  public CombienDeTemps() {
    // Le Box qui contient la saisie du taux
    Box bTaux= Box.createHorizontalBox();
    bTaux.add( new JLabel( "Taux: "));
    bTaux.add( tfTaux);
    bTaux.add( new JLabel( " % "));
    // Le Box qui contient la saisie du montant
    Box bSomme= Box.createHorizontalBox();
    bSomme.add( new JLabel( "Somme: "));
    bSomme.add( tfSomme);
    bSomme.add( new JLabel( " € "));
    // Le Box qui contient la saisie du montant de la
    // mensualité
    Box bMensualite= Box.createHorizontalBox();
    bMensualite.add( new JLabel( "Mensualité: "));
    bMensualite.add( tfMensualite);
    bMensualite.add( new JLabel( " € "));
    // Le Box qui contient l'affichage du nombre de mensualités
    Box bDuree= Box.createHorizontalBox();
    bDuree.setBorder( new BevelBorder( BevelBorder.LOWERED));
    bDuree.add( Box.createHorizontalGlue());
    bDuree.add( lblDuree);
    bDuree.add( Box.createHorizontalGlue());

    Box principal= Box.createVerticalBox();
    principal.add( bTaux);
    principal.add( bSomme);
    principal.add( bMensualite);
    principal.add( bDuree);
    principal.add( bpCalculer);

    add( principal);

    bpCalculer.addActionListener( this);
```

```
  }

  /** Lorsque l'utilisaeur appuie sur le bouton,
  * on procède au calcul en utilisant la méthode
  * statique de la classe Emprunt
  */
  public void actionPerformed( ActionEvent e) {
    // On lance le calcul:
    try {
      int nombreMens= Emprunt.getNombreMensualites(
        Double.parseDouble( tfTaux.getText())/100,
        Double.parseDouble( tfSomme.getText()),
        Double.parseDouble( tfMensualite.getText()));
        lblDuree.setText("    Durée de remboursement: "
          +nombreMens+" mois    ");
    } catch( Exception ex) {
      lblDuree.setText( "    Données saisies incorrectes   ");
    }
  }
}
```

La classe QuelleSomme :

```
package ihm.emprunt;

import outils.Emprunt; // Classe des méthodes de calcul
import javax.swing.*;
import javax.swing.border.BevelBorder;
import java.awt.Dimension;
import java.awt.event.*;

public class QuelleSomme extends JPanel
      implements ActionListener{
  JTextField tfTaux= new JTextField();
  JTextField tfMensualite= new JTextField();
  JTextField tfDuree= new JTextField();
  JLabel lblMontant= new JLabel(
    "    Montant emprunté: 0 €    ");
  JButton bpCalculer= new JButton( "Calculer");

  /** Dans le constructeurn on construit l'IHM à base de
  * conteneurs Box
  */
  public QuelleSomme() {
    // Le Box qui contient la saisie du taux
    Box bTaux= Box.createHorizontalBox();
    bTaux.add( new JLabel( "Taux: "));
    bTaux.add( tfTaux);
    bTaux.add( new JLabel( " % "));
    // Le Box qui contient la saisie du montant de la
```

```
    // mensualité
    Box bMensualite= Box.createHorizontalBox();
    bMensualite.add( new JLabel( "Mensualité: "));
    bMensualite.add( tfMensualite);
    bMensualite.add( new JLabel( " € "));
    // Le Box qui contient la saisie du nombre de mensualités
    Box bDuree= Box.createHorizontalBox();
    bDuree.add( new JLabel( "Durée: "));
    bDuree.add( tfDuree);
    bDuree.add( new JLabel( " mois "));
    // Le Box qui contient l'affichage du montant
    Box bMontant= Box.createHorizontalBox();
    bMontant.setBorder( new BevelBorder( BevelBorder.LOWERED));
    bMontant.add( Box.createHorizontalGlue());
    bMontant.add( lblMontant);
    bMontant.add( Box.createHorizontalGlue());

    Box principal= Box.createVerticalBox();
    principal.add( bTaux);
    principal.add( bMensualite);
    principal.add( bDuree);
    principal.add( bMontant);
    principal.add( bpCalculer);

    add( principal);

    bpCalculer.addActionListener( this);
  }

  /** Lorsque l'utilisaeur appuie sur le bouton,
  * on procède au calcul en utilisant la méthode
  * statique de la classe Emprunt
  */
  public void actionPerformed( ActionEvent e) {
    // On lance le calcul:
    try {
      double montant= Emprunt.getMontant(
        Double.parseDouble( tfTaux.getText())/100,
        Double.parseDouble( tfMensualite.getText()),
        Integer.parseInt( tfDuree.getText()));
        lblMontant.setText("    Montant emprunté: "
          +java.lang.Math.floor(montant)+"  €    ");

    } catch( Exception ex) {
      lblMontant.setText(
        "    Données saisies incorrectes   ");
    }

  }
}
```

Exercice n° 2

Classe de gestion de messages :

```
package reseau;

import java.util.*;
import java.io.*;
import java.net.*;

public class DataBox extends Thread {
  // Les messages sont stockés dans ce Vector
  Vector messages= new Vector();

  // Sémaphore qui permet de refermer la thread d'écoute
  boolean ouvert= true;

  // Numéro de port d'écoute
  int port;

  /** Constructeur.
  * Il prend en argument le port. Ce constructeur est utilisé
  * pour créer un DataBox en écoute de messages.
  */
  public DataBox( int port) {
    this.port= port;
    // Création du thread qui écoute le datagram
    start();
  }

  /** Le run d'écoute qui tournera dans une thread
  */
  public void run() {
    DatagramPacket dp;
    DatagramSocket ds;
    try {
      dp= new DatagramPacket( new byte[5000], 5000);
      ds= new DatagramSocket( port);
      ds.setSoTimeout( 3000); // Timeout de 3000 millisecondes
      } catch( IOException e) {
        System.out.println( "Erreur I/O: "+e.getMessage());
        return;
      }
      while( ouvert) {
        try {
          ds.receive( dp);
          byte[] buffer= dp.getData();
          String message= new String(dp.getData());
          messages.add( message);
        } catch( IOException e) {
```

```
        // Time out de 3000 millisecondes
      }
    }
}


/** Demande de fermeture du DataBox.
* Le thread en cours d'écoute se terminera à la fin
* du timeout sur la méthode receive.
*/
public void close() {
  // Si la thread est en cours, elle est arrêtée
  ouvert= false;
}

/** Envoi d'un message. La méthode est static
*/
public static void send( String adresse, int port,
    byte[] message) {
  try {
    DatagramPacket dp= new DatagramPacket( message,
      message.length,
      InetAddress.getByName( adresse), port );
    DatagramSocket ds= new DatagramSocket();
    ds.send( dp);
  } catch( UnknownHostException e) {
    System.out.println( "Host inconnu: "+e.getMessage());
  } catch( IOException e) {
    System.out.println( "Erreur I/O: "+e.getMessage());
  }
}

/** Si le vector n'est pas vide, c'est qu'il y
* a des messages.
*/
public boolean messageReady() {
  return( !messages.isEmpty());
}

/** Renvoie le message suivant (c'est une pile FIFO)
*/
public byte[] getNextMessage() {
  String message= (String) messages.get(0);
  messages.removeElementAt( 0);
  return message.getBytes();
}


/** Le main permet de texte dans les deux cas: envoi et
* réception
```

```
  */
  public static void main( String [] args) {
    switch( args.length) {
      case 3: // Envoi d'un message
        DataBox.send( args[0], Integer.parseInt( args[1]),
          // le code 6 permet d'être utilisé comme
          // indicateur de fin de message
          (args[2]+"\6").getBytes());
        break;
      case 1: // Ecoute d'un message
        DataBox db= new DataBox( Integer.parseInt( args[0]));
        while( !db.messageReady());
        System.out.println( "Message arrivé:");
        String message= new String( db.getNextMessage());
        System.out.println( message.substring( 0,
          message.indexOf( "\6")));
        db.close();
        break;
      default:
        System.out.println( "Mauvaise syntaxe");
    }
  }
}
```

Exercice n° 3

Classe de gestion de messages avec accusés de réception :

On reprend le même code que dans l'exercice précédent. Par contre:

- après toute réception d'un message on envoit un message d'accusé de réception
- après toute émission d'un message, on attend l'accusé de réception par un receive. On met un timeout de 10 secondes d'attente de l'accusé de réception. Au delà, on considère que le message a été perdu et on retourne false. On remarque que l'on utilise, pour l'accusé de réception, le même port que pour le message (alloué dynamiquement).

Question: Que se passe t-il si le message d'accusé de réception est perdu ?

L'émetteur pense que le message a été perdu, alors qu'il a bien été reçu. Généralement, l'émetteur renvoie à nouveau le message.Il faudra donc prévoir dans le récepteur un moyen d'éliminer les messages reçus deux fois (par exemple par une clé unique spécifiée dans chaque message). Je vous laisse le soin d'améliorer ce programme dans ce sens.

La classe ARDataBox :

```
package reseau;

import java.util.*;
import java.io.*;
import java.net.*;
```

```
public class ARDataBox extends Thread {
  // Les messages sont stockés dans ce Vector
  Vector messages= new Vector();

  // Sémaphore qui permet de refermer la thread d'écoute
  boolean ouvert= true;

  // Numéro de port d'écoute
  int port;

  /** Constructeur.
  * Il prend en argument le port. Ce constructeur est utilisé
  * pour créer un ARDataBox en écoute de messages.
  */
  public ARDataBox( int port) {
    this.port= port;
    // Création du thread qui écoute le datagram
    start();
  }

  /** Le run d'écoute qui tournera dans une thread
  */
  public void run() {
    DatagramPacket dp;
    DatagramSocket ds;
    try {
      dp= new DatagramPacket( new byte[5000], 5000);
      ds= new DatagramSocket( port);
      ds.setSoTimeout( 3000); // Timeout de 3000 millisecondes
      } catch( IOException e) {
        System.out.println( "Erreur I/O: "+e.getMessage());
        return;
      }
      while( ouvert) {
        try {
          ds.receive( dp);

          // Si je reçois un message, je renvoie un accusé
          // de réception
          DataBox.send( dp.getAddress().getHostAddress(),
            dp.getPort(), "OK".getBytes());
          byte[] buffer= dp.getData();
          String message= new String(dp.getData());
          messages.add( message);
        } catch( IOException e) {
          // Time out de 3000 millisecondes
        }
      }
  }
  /** Demande de fermeture de l'ARDataBox.
```

```
* Le thread en cours d'écoute se terminera à la fin
* du timeout sur la méthode receive.
*/
public void close() {
  // Si la thread est en cours, elle est arrêtée
  ouvert= false;
}
/** Envoi d'un message.
*/
public static boolean send( String adresse, int port,
    byte[] message) {
  try {
    DatagramPacket dp= new DatagramPacket( message,
      message.length,
      InetAddress.getByName( adresse), port );

    DatagramSocket ds= new DatagramSocket();
    System.out.println( "Adresse: "
      +dp.getAddress().getHostAddress()+":"+ dp.getPort());
    ds.send( dp);
    // Ecoute de l'Accusé de Reception avec un timeout
    // de 10 secondes
    ds.setSoTimeout( 10000);
    ds.receive( dp);
  } catch( UnknownHostException e) {
    System.out.println( "Host inconnu: "+e.getMessage());
    return false;
  } catch( IOException e) {
    System.out.println( "Erreur I/O: "+e.getMessage());
    return false;
  }
  return true;
}
/** Si le vector n'est pas vide, c'est qu'il y
* a des messages. */
public boolean messageReady() {
  return( !messages.isEmpty());
}
/** Renvoie le message suivant (c'est une pile FIFO)
*/
public byte[] getNextMessage() {
  String message= (String) messages.get(0);
  messages.removeElementAt( 0);
  return message.getBytes();
}
/** Le main permet de texte dans les deux cas: envoi
* et réception */
public static void main( String [] args) {
```

```
    switch( args.length) {
      case 3: // Envoi d'un message
        if( ARDataBox.send( args[0],
          Integer.parseInt( args[1]),
            (args[2]+"\6").getBytes()))
          System.out.println( "Message bien recu");
        else
          System.out.println( "Message NON RECU");
        break;
      case 1: // Ecoute d'un message
        ARDataBox db= new ARDataBox(
          Integer.parseInt( args[0]));
        while( !db.messageReady());
        System.out.println( "Message arrivé:");
        String message= new String( db.getNextMessage());
        System.out.println( message.substring( 0,
          message.indexOf( "\6")));
        db.close();
        break;
      default:
        System.out.println( "Mauvaise syntaxe");
    }
  }
}
```

Annexe G : Glossaire

abstract Modificateur pour une classe ou une méthode. Il permet de spécifier que l'implémentation de la classe ou de la méthode est à redéfinir dans une sous-classe.

accesseur Méthode permettant l'accès aux propriétés d'un objet.

ACID Atomicité, Consistence, Isolation, Durabilité. Ce sont les propriétés qui définissent une transaction.

API Application Programming Interface. Ensemble de fonctions ou de méthodes qui permettent la programmation d'une application.

applet Petite application qui s'exécute à l'intérieur d'un navigateur Internet.

AWT Abstract Window Toolkit. C'est l'API de Java pour la création d'interfaces graphiques à partir de l'API du système d'exploitation de la machine.

bac à sable Environnement sécurisé dans lequel sont instanciées les applets des navigateurs.

Bean "Grain de café". C'est un objet Java qui sera utilisé comme composant offrant des services.

bytecode Code généré par le compilateur Java et qui est indépendant de la machine sur laquelle il va s'exécuter.

casting Conversion explicite d'un type vers un autre.

champ Variable d'instance d'un objet.

classe Définition du comportement d'un ensemble d'objets. La classe est le moule qui permet de fabriquer des objets. Elle représente aussi le type de l'objet.

CLASSPATH Variable d'environnement permettant de localiser les packages et les fichiers JAR dont a besoin Java ?

client Représente l'application visible par l'utilisateur.

clône Copie parfaite d'un objet. Un clône d'un objet doit être du même type et avoir les mêmes valeurs dans les propriétés significatives.

commit Ordre de terminaison d'une transaction.

compilateur Programme permettant de transformer un fichier dans lequel le programme est écrit en claire (dans un langage de programmation comme Java) en un fichier contenant des instructions binaires.

composant Objet répondant à des spécifications pour lui permettre de s'integrer dans un conteneur de composants.

constructeur Méthode d'un objet qui est appelée uniquement au moment de son instanciation, alors qu'il vient d'être mis dans la mémoire pour la première fois.

conteneur Programme dont la fonction est de contenir des composants, et de prendre en charge à la fois leur cycle de vie et leurs communications avec l'extérieur.

core API API de base de Java.

deprecated Terme anglais. Voir dépréciation.

dépréciation Mise à l'index de méthodes ou de classes jugées inadéquates ou mal faites. Pour des raisons de compatibilité ascendante, elles sont toujours présentes dans Java mais il est conseillé de ne plus les utiliser.

dérivation Lorsque l'on crée un nouvelle classe qui en hérite d'une autre.

encapsulation Moyen de cacher l'implémentation d'un objet derrière son interface.

exception Evénement anormal qui doit être capturé et analyse.

extends Mot clé permettante de définir qu'une classe en hérite d'un autre.

field Voir champ.

final Modificateur pour une classe, un attribut ou une méthode qui définit la notion de constante. Pour une propriété, sa valeur ne peut être modifiée, pour une classe ou une méthode, elles ne peuvent être redéfinies dans des classes filles.

friendly Modificateur pour des classes ou des membres limittant leur accès depuis des classes du même package.

FTP File Tranfert Protocol. Protocole de gestion de transfert de fichiers sous TCP/IP.

garbage collector Voir ramasse-miettes.

généralisation Démarche qui consiste à partir de classes très spécialisées et à créer une hiérarchie vers des classes les plus génériques possibles. Cela permet de favoriser la réutilisation des classes.

GIF Format de fichiers d'images très répendu sur Internet.

GUI Graphical User Interface, voir IHM

héritage Réutilisation d'une classe pour définir une nouvelle classe.

HTML Hyper Text Modeling Language. Langage de marquage pour composer des documents Hyper-textes. C'est le standard le plus utilisé sur le Web.

HTTP Hyper Text Transfert Protocole. Protocole applicatif sur TCP/IP pour transmettre des fichiers entre en serveur et un client.

HTTPS Version sécurisée de HTTP. C'est l'utilisation du cryptage pour cacher le contenu des fichiers transmis.

identificateur Nom d'un élément (variable, objet, etc) dans le langage de programmation.

IHM Interface Homme Machine. Elle définit comment doit se passer l'échange entre l'utilisateur et l'ordinateur.

implements Mot clé permettante de définir qu'une classe imlémente une ou plusieurs interfaces.

import Mot clé pour signaler que l'on souhaite utiliser un package ou une classe dans le fichier source Java.

instance Représente un objet. C'est donc une incarnation d'une classe en mémoire centrale.

interface Sorte de classe qui ne contient que des signatures de méthodes. Si une classe implémente une interface, elle a l'obligation d'implémenter toutes les méthodes de l'interface.

Internet Réseau utilisant le protocole TCP/IP et dont le maillage couvre pratiquement toute la planète. C'est aujourd'hui le plus grand réseau de communication permettant de relier les humains entre eux.

interpréteur Programme qui interprète un programme écrit dans un langage informatique vers du langage machine pour l'exécuter.

introspection Mécanisme qui s'appuie sur des méthodes implémentées dans des composants pour récupérer leurs caractéristiques (propriétés et méthodes).

IP Internet Protocol. C'est un protocole réseau très répendu dans le monde.

JAR Java Archive. Ces fichiers sont destinés à contenir tous les fichiers (programmes et données) nécessaires au déploiement d'une application Java. JAR est à la fois l'extention de ces fichiers et le nom du programme du JDK qui permet de les générer.

JavaBeans Norme pour construire des composants en Java.

JDBC Java DataBase Connectivity. C'est l'API de Java qui permet d'accéder à des moteurs de bases de données relationnelles.

JDK Java Development Kit. Kit de développement Java, minimaliste mais suffisant, proposé gratuitement par Sun.

JFC Java Foundation Classes. Ensemble de classes destinées à faciliter le développement d'applications clientes écrites en Java. Elles sont principalement graphiques et d'une très grande richesse.

JIT Compiler Just In Time Compiler. Technologie de machine virtuelle destinée à améliorer les performances lors de l'exécution d'une application Java.

JPEG Format de fichiers d'images très répendu sur Internet, utilisant un algorithme de compression particulièrement adapté aux photographies.

JRE Java Runtime Environment. Environnement permettant l'exécution de programmes Java. C'est donc une JVM et tout son environnement.

JVM Java Virtual Machine. Interpréteur de programmes Java.

Machine virtuelle Voir JVM.

membre Tout ce qui fait partie d'un objet est un membre: Variable (propriété) ou fonction (méthode) ou classe interne.

méthode Fonction interne à un objet et éventuellemen accessible de l'extérieur de cet objet (invoquer une méthode).

modificateur Mot clé permettant de spécifier une caractéristique à une classe, une méthode ou une propriété. Par exemple: public, private, protected, static, final, etc.

multi-thread Particularité des programmes qui lancent plusieurs traitements en parallèle.

null Mot clé qui représente la nullité d'une variable objet. Cela signifie que cette variable ne pointe sur aucun objet.

objet Programme en mémoire centrale, créé à partir d'une classe, qui possède ses propres propriétés (variables d'instance) et qui peut communiquer avec d'autres objets. Un objet est une instance d'une classe.

overloading voir surcharger.

overriding voir redéfinir.

package Ensemble de classes rassemblées dans un répertoire ou un fichier JAR.

persistence Mécanisme qui permet de stocker un objet (binaire) au travers d'un flot de données (fichier, serveur, base de données…)

pixel Point graphique doté d'une couleur. Une image est un ensemble de pixels.

polymorphisme Notion objet qui consiste à coder une méthode de différentes manières suivant le contexte ou les arguments passés.

POP 3 Post Office Protocol.

private Modificateur concernant un membre d'une classe afin de ne permettre son accès qu'à elle même.

process Programme s'exécutant sur un ordinateur. Il peut être composé d'un ou plusieurs Threads.

propriété Variable d'instance stockée dans un objet.

protected Modificateur concernant un membre d'une classe afin de ne permettre son accès qu'à elle même et ses héritiers.

public Modificateur concernant une classe ou un membre d'une classe pour spécifier qu'il est accessible de l'extérieur sans limitation.

ramasse-miettes Tâche qui s'exécute dans la JVM et qui a pour fonction de nettoyer régulièrement la mémoire des objets dont on a plus besoin.

RDBMS Relational Data Base Management System. Voir SGBDR.

redéfinir Implémenter une méthode qui existe déjà une classe dont on hérite afin de redéfinir son comportement. Les méthodes doivent avoir exéctement la même signature.

référence Adresse d'un objet.

reflexion Mécanisme propre à Java qui permet d'analyser automatiquement les particularités d'un objet (méthodes et prorpiétés).

RFC Request For Comment: Document produit par les membres de l'IETF (Internet Task Force) dont l'objectif est de clarifier ou normaliser un concept ou une règle autour de l'Internet.

RMI Remote Method Invocation: Mécanisme propre à Java pour invoquer les méthodes d'un objet qui se trouve sur une machine distante.

rollback Ordre de terminaison d'une transaction en annulant tout ce qui y a été fait.

sandbox voir bac à sable.

sérialisation Mécanisme qui permet d'envoyer un objet vers un flux binaire afin de le rendre persistant.

SGBDR Système de Gestion de Base de Données Relationnelles. Moteur de base de données accessible au travers d'un réseau.

signature Caractéristiques d'une méthode. Cela prend en compte son nom, le nombre de ses arguments et les types de ses arguments. S'il est possible dans une classe d'avoir plusieurs méthodes qui portent le même nom, il est impossible d'avoir deux méthodes qui ont la même signature. Elle doit être unique dans chaque classe.

skeleton Objet serveur en RMI qui permet d'invoquer les méthodes d'un objet distant.

SMTP Simple Mail Transfert Protocol. Protocole applicatif pour la transmission de courriers électroniques sur Internet.

socket Interface de programmation pour communiquer entre deux machines avec TCP/IP.

spécialisation Démarche qui consiste à partir de classes très générales et à exploiter l'héritage pour construire de nouvelles classes de plus en plus spécialisées.

SQL Structured Query Language. Langage quasi universel pour interroger les bases de données relationelles.

SSL Secure Socket Layer. Version sécurisée (à base de cryptage) des sockets.

static Modificateur pour spécifier qu'un membre est un membre de classe. Pour une propriété, sa valeur est partagée parmi toutes les instances de la classe, pour une méthode elle ne pourra faire référence qu'à des propriétés de classe.

stub Objet RMI qui simule un objet distant, qui prend en charge les invocations de ses méthodes mais qui les lui transmet par l'intermédiaire d'un skeleton.

super Ce mot clé représente, dans un objet, l'instance de l'objet dont on hérite.

surcharger Créer des méthodes qui ont le même nom (qui ont donc la même fonction) mais des arguments différents, et donc une implémentation différente. C'est une des bases du polymorphisme.

Swing Ensemble des classes graphiques des JFC.

synchronisation Possibilité donnée à une méthode ou une portion de code d'empêcher plusieurs accès simultanés par plusieurs threads. Cela s'avère nécessaire pour du code non réentrant, c'est à dire qui utilise une ressource non partageable (fichier, espace graphique, etc.)

synchronized Mot clé qui permet de mettre en œuvre la synchronisation.

TCP/IP Transmission Control Protocol sur IP. Protocole réseau pour la communication par des flux de données.

this Ce mot clé représente, dans un objet, sa propre instance.

thread Unité d'exécution de code. Ce n'est rien d'autre qu'un pointeur de programme s'exécutant en parallèle avec d'autres. Un process est composé d'au moins un thread.

transaction Ensemble d'opérations dont la finalité est qu'elles se soient toutes exécutées sans erreur ou bien (en cas d'erreur) qu'aucune ne se soit exécutée.

transient Modificateur pour une prorpiété pour spécifier qu'elle est temporaire et qu'elle ne doit pas être sérialisée au cas où l'objet dont elle fait partie serait sérialisé.

UDP Unreliable Datagram Protocol. Ce protocole permet la communication sur le réseau par messages (sans garantie de réception).

UML Unified Modeling Language. Notation qui permet de représenter graphiquement toute une structure orientée objet.

URI Uniform Resource Identifier (RFC 1630).

URL Uniform Resource Locator.

variable Unité de stockage en mémoire, nommée, typée et qui peut recevoir une valeur numérique ou l'adresse d'un objet (suivant son type).

variable de classe Variable statique. Son contenu est partagé parmi tous les objets issus de la mêm classe.

Web Toile qui définit l'accès à toutes les ressources de l'Internet.

widget Petit composant graphique écrit en Java.