Initiation à la programmation objet avec Java

Olivier Capuozzo

17 Avril 2004

Ceci est un support de cours DAIGL (Développement d'Applications Informatiques et Génie Logiciel) à destination d'étudiants en STS IG (Section de Technicien Supérieur en Informatique de Gestion).

Il fait suite à 2 mois d'initiation à la programmation avec PHP.

Ce support est basé sur le langage Java, et mis en oeuvre sous GNU/Linux, mais peut l'être sur d'autres environnements non ouverts, non libres, comme MS-Windows, sans aucun problème particulier.

Attention : ce support de cours demande, de la part des étudiants, un engagement personnel qui va bien au-delà d'un investissement durant les trois petites heures de TP hebdomadaires prévues par le référentiel actuel.

Certaines parties peuvent paraître trop complexes, trop ambitieuses. C'est certainement vrai pour quelques unes (le composant d'envoi de mels pourrait être transformé en une simple application - chapitre 11, ou la présence du composant `JTabbedPane` au chapitre 9), mais pas pour d'autres. Par exemple le polymorphisme, introduit très tôt, est perçu comme un mécanisme naturel.

La progression proposée est expérimentale et n'a pas vocation de modèle. L'initiation à la programmation via PHP (le prérequis) a du bon (souplesse de l'immédiateté) et du moins bon (approche procédurale). Le JDK 1.5 facilitera peut-être une autre approche.

Pour le CERTA [1]et échanges sur ce thème : olivier.capuozzo@reseauCerta.org

Ce document a été réalisé sous GNU/Linux avec vim [2], au format docbook [3], mis en page avec le processeur XSLT saxon [4]développé par Michael Kay et les feuilles de styles de Norman Walsh [5].

Table des matières

[1] http://www.reseaucerta.org/
[2] http://www.vim.org
[3] http://www.oasis-open.org/docbook/
[4] http://saxon.sourceforge.net
[5] http://nwalsh.com/

1. Intro

Les concepts de la programmation orientée objet datent des années 60 avec le langage Algol puis Simula. Le langage Smalltalk est une référence en la matière et plus récemment Eiffel (fin de année 80 début année 90).

Objectifs principaux de l'approche objet.

- Produire des logiciels fiables, garants d'une certaine robustesse.
- Rechercher les qualités de réutilisabilité et d'extensibilité.
- Faciliter le déploiement et la maintenabilité.

C'est, en d'autres mots, répondre aux exigences du **génie logiciel** dont l'objectif est de produire du logiciel de qualité.

Java est un langage de programmation qui permet une certaine approche objet des solutions.

Hormis les types primitifs présentés plus loin dans ce document, tout est objet en Java.

Java est le fruit d'un travail initié par une équipe d'ingénieurs de chez SUN Microsystems. Java s'est fortement inspiré du C++ dont il a éliminé certaines caractéristiques comme les pointeurs et limité d'autres comme l'héritage multiple.

Quelques caractéristiques remarquables de Java

- Fortement typé.

  Distinction forte entre les erreurs survenant à la compilation (erreur de syntaxe, de non concordance de type, de visibilité...) et les erreurs d'exécution (chargement et liaison des classes, génération de code, optimisation...).

- Indépendance des plates-formes d'exécution.

  Dans la mesure où celle-ci dispose d'une machine virtuelle Java (JVM)

- Sûreté de fonctionnement.

  Le système d'exécution Java, inclus dans la JVM, vérifie que le code n'est pas altéré avant son exécution (absence de virus et autre modification du code).

- Gestion automatique de la mémoire.

  Libération automatique de la mémoire réalisée par un garbage-collector (ramasse-miettes). Mais ordre de création des objets à l'initiative du développeur (opérateur new)

- Orienté Objet.

  Diminution du temps de développement grâce à la réutilisation de **composants logiciels** .

- Réalisation d'interface utilisateur graphique.

  Une hiérarchie de classe (Java Fundation Classes - jfc) connue sous le nom de Swing, permet de construire des IHM de qualité.

- Interprété et multitâches.

  Le code est d'abord compilé, traduit en Byte code. C'est ce byte code qui sera interprété par la JVM.

  Le langage supporte le multithreading avec primitives de synchronisation.

- Robuste.

  La gestion des exceptions est intégrée au langage.

- Orienté réseau et client/serveur.

  Client/serveur traditionnel : Application classique, gestion de protocoles réseaux, pont d'accès aux bases de donnée (jdbc), RMI. . .

  Client/serveur Web : Applet côté client et Servlet côté serveur.

- Déploiement facilité.

  Les classes peuvent être assemblées en unités d'organisation physique et logique (package). Présence d'une API d'internationalisation.

- Gratuit et "participatif".

  De grandes entreprises participent avec SUN à la réussite de Java, et croient à l'avenir de ce langage : IBM, ORACLE, BORLAND...

  Dispose d'une grosse bibliothèque de base (plus 1600 classes et interfaces pour le JDK 1.2.2).

1.1. Kit de développement Java

Les principales Versions du jdk

- jdk 1.0 : première version, orienté développement Web (applet)
- jdk 1.1 : version plus costaud, destiné aussi aux applications d'entreprises.
- jdk 1.2 : améliore et enrichie la 1.1 : collections, interface utilisateur swing etc.
- jdk 1.3 : de nouvelles performances et de nouvelles classes (3D . . .)
- jdk 1.4 : amélioration de la vitesse d'exécution, de nouvelles fonctionnalités (XML, String...)
- jdk 1.5 : à venir, collections typées, encapsulation automatique des types primitifs...

1.2. Installation du jdk

Pour développer en java, vous devez disposer d'un `JDK` (*Java Developement Kit*), un kit de développement, ainsi qu'une documentation associée et un éditeur.

Pour installer un JDK sur votre machine, vous pouvez vous référer ici http://www.linux-france.org/prj/edu/archinet/DA [6](rubrique Développement). Vous y trouverez une procédure détaillée, contenant en fin de document un lien vers une ressource d'installation de java pour d'autres systèmes (Mac, Windows).

Parmi les sous-répertoires du jdk, on trouve

- **`bin`** : Fichiers exécutables dépendants de la plate-forme (java, javac . . .)
- **`demo`** : Une collection de programmes de démonstration (jfc et applet)
- **`jre`** : Java Runtime Environment. Un environnement d'exécution pour la plateforme.
- **`lib`** : Diverses librairies.

A la racine de l'installation se trouve le fichier `src.jar` qui contient les sources de la librairie. Il peut être décompressé dans un sous répertoire à créer pour l'occasion (par exemple <rep du jdk>/src).

2. Premier programme

Le point d'entrée d'un programme Java standard est une fonction à l'entête prédéfinie :

```
/*** fonction appelée automatiquement
* au lancement du programme.
* @param args la liste des arguments reçus en ligne de commande
*/
 static public void main(String[] args)
```

Elle est appelée par le système, après chargement, suite à son lancement.

En programmation objet, toute fonction est forcément placée **à l'intérieur** d'une structure appelée **classe** .

Un programme Java est donc composé d'au moins une classe, et au plus... il n'y a pas de limite théorique ! Ainsi il n'est pas rare de voir des projets ayant plus d'un millier de classes !

[6] http://www.linux-france.org/prj/edu/archinet/

Avant tout chose, créons un répertoire. Il est **fortement conseillé** de créer **un répertoire par projet** . Un projet regroupe l'ensemble des fichiers nécessaires à l'application.

```
[java]$ mkdir tp1
[java]$ cd tp1
[tp1]$
```

Bon, créons notre classe. Comment allons nous l'appeler ? Nous avons pris l'habitude en PHP, de bien nommer nos identifiants, nous ferons bien entendu de même avec Java (ou tout autre langage).

Convention de nommage

Par convention, le nom des classes commence **toujours** par une **majuscule** .

A l'inverse, le nom des fonctions (méthodes) et des attributs (variables) commence toujours par une **minuscule** .

Nous enregistrons notre classe dans un fichier nommé `AppTp1.java` .

```
/*** fichier : AppTp1.java
* date : 02-novembre-2003
*/
public class AppTp1 {
 /**
  * fonction appelée automatiquement
  * au lancement du programme.
  * @param args la liste des arguments
  * reçus en ligne de commande
 */
 static public void main(String[] args) {
   // ne fait rien
 }
}
```

Vous constaterez que, comme PHP, les structures de blocs sont délimitées par les symboles `{` et `}` , et le commentaires par `// bout de ligne commenté` , pour une ligne, ou `/* plusieurs lignes */` .

Il existe une convention très intéressante concernant les commentaires : Ceux multilignes de la forme `/** ... */` , observez les **deux étoiles du début** . S'ils sont placés **avant** certaines parties (classe, attributs, méthodes), ils seront perçus comme des commentaires **javadoc** , qui servent de base de données à la construction automatique de la **documentation des interfaces (API)** de votre code. Vous n'avez pas à vous en inquiéter pour le moment.

Vous devez faire attention à deux choses ici : 1/ l'extension d'un programme source java est `.java` , en minuscule. 2/ le nom du fichier est identique au nom de la classe (lorsque celle-ci est déclarée publique).

2.1. Compilation

Tel quel, le programme n'est pas exécutable.

Il doit être **compilé** , c'est à dire traduit en un autre langage, appelé **byte code** pour être compris par l'interpréteur. Au cours de cette traduction, de nombreuses et précieuses vérifications sont réalisées. Si des erreurs sont rencontrées par le compilateur (le traducteur), il ne générera pas la version compilée, charge alors au développeur de corriger sa copie. Exemple de compilation avec **`javac`** :

```
[tp1]$ ll
total 4
-rw-rw-r-- 1 kpu   kpu   94 nov  1 20:46 AppTp1.java
```

```
[tp1]$
[tp1]$ javac AppTp1.java
[tp1]$
[tp1]$ ll -tr
total 8
-rw-rw-r--   1 kpu   kpu  94 nov  1 20:46 AppTp1.java
-rw-rw-r--   1 kpu    kpu   257 nov  1 20:47 AppTp1.class
[tp1]$
```

La compilation a bien fonctionné. Un nouveau fichier est apparu, avec une extension `.class` .

2.2. Exécution

Nous utiliserons la commande `java` (comme la commande précédente, mais sans le `c` pour compilateur). En prenant bien soin de ne pas mentionner l'extension ! Exemple :

```
[tp1]$ java AppTp1
[tp1]$
```

Ainsi, il ne se passe rien ! Que s'est-il passé au juste ?

La commande java a fait en sorte de charger la classe `AppTp1.class` en mémoire, puis la méthode `main` est appelée. Comme elle ne comporte aucune instruction, son exécution se termine et le programme s'arrête.

2.3. Modification

Modifions notre programme afin qu'il affiche un message à l'écran :

```
public class AppTp1 {
  static public void main(String[] arg) {
   System.out.println ("Hello, ce n'est pas encore tout à fait");
   System.out.println ("de la programmation objet !");
  }
}
```

Pour réaliser l'affichage, nous utilisons l'instruction **`System.out.println(...)`** . Au regard de la convention de nommage, et avec notre expérience de la programmation en PHP, nous pouvons affirmer sans erreur que :

- `System` est une classe; le premier caractère est une majuscule, la suite en minuscule.
- `out` est un identificateur qui commence par une minuscule et s'utilise **sans parenthèse** : c'est donc une variable (mais on dit **attribut** ou **champ**). Ici `out` est un attribut publique détenu par la classe `System` , **il n'y a pas de variables globales en programmation objet** .
- `println()` débute par une minuscule et s'utilise avec des parenthèses, c'est donc une fonction (mais on dit **méthode**). **Il n'y a pas de fonctions globales en programmation objet** .

Équivalent PHP : `print()` ou `echo()` .

Après avoir sauvegardé les modifications, nous devons **recompiler** le programme pour, de nouveau, le tester :

```
[tp1]$ vim AppTp1.java     # ici nous avons utiliser l'éditeur vim
```

```
[tp1]$ javac AppTp1.java   # compilation
[tp1]$ java AppTp1         # exécution
Hello, ce n'est pas encore tout à fait
de la programmation objet !
[tp1]$
```

Voilà, le tour est joué. Mais pourquoi avoir mentionné que ce *n'est pas tout à fait de la programmation objet* ?

Simplement parce qu' **aucun objet n'a été crée !** Et pourtant il y a un lien extrêmement étroit entre une classe et un objet.

Dans ce qui suit, nous allons vous montrer comment concevoir et utiliser un, ou des objets, en faisant évoluer progressivement notre projet.

3. Première notion d'objet.

3.1. Introduction

La notion d'objet

Un objet est une sorte de module relativement indépendant qui représente un objet du domaine étudié.

On peut dire qu'un objet est un petit programme en soi : il dispose de ses propres variables 'globales' (ses attributs), et de fonctions (ses méthodes).

Imaginons un programme qui manipule des lignes de commandes. Toutes les lignes de commande sont construites sur le **même moule** , à savoir une ligne de commande est composée de la référence à un produit, son intitulé, son prix unitaire, la quantité commandée et son prix total. Ces caractéristiques seront donc réunies dans une **classe** . Il est coutume de représenter une classe par un diagramme UML (*Unified Modeling Language*), un langage de modélisation standard.

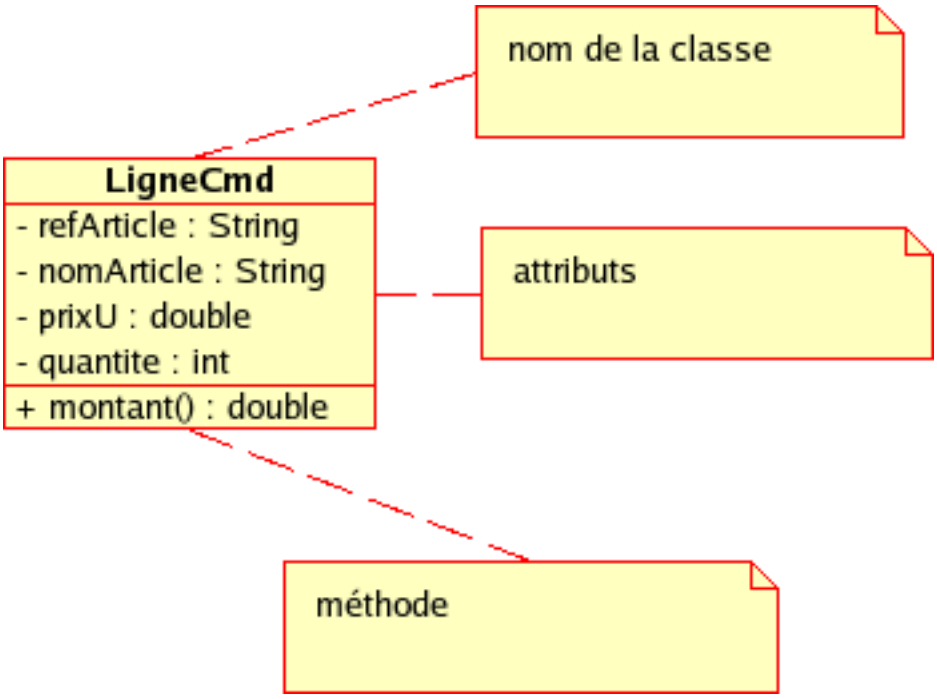


Figure 1. Classe LigneCmd, version UML

Ce que dit ce diagramme :

- La classe se nomme `LigneCmd` .
- Elle a 4 attributs (ou champs), 2 de type chaîne de caractères (`String`), un réel (`double`) et un entier (`int`).
- Elle a une méthode (`montant`) qui retourne un réel (`double`)

  Remarque : fonction et méthode sont des termes synonymes.

Nous traduirons ce diagramme par une **classe Java** . Comme le montre la figure ci-dessous, la traduction est directe :

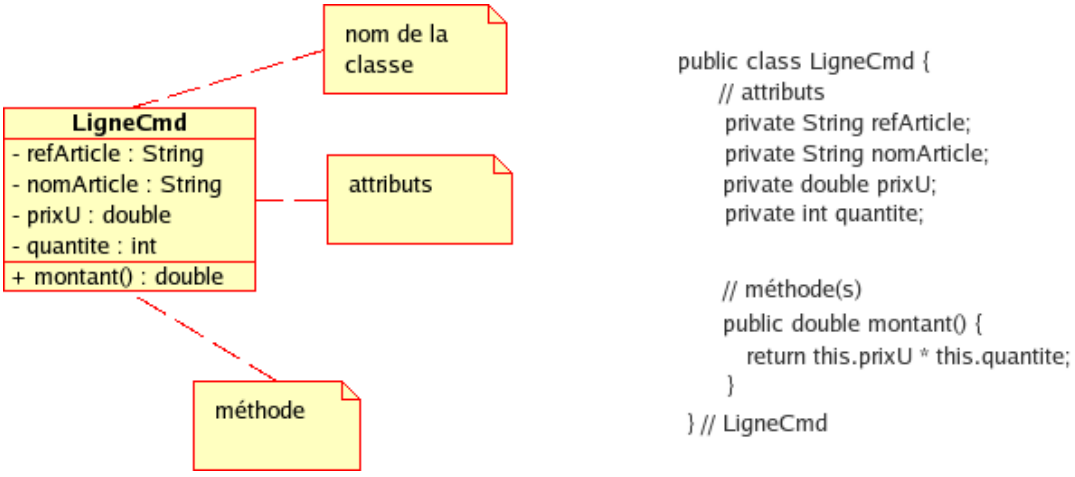


Figure 2. Classe LigneCmd, UML et Java

Notons que certains modeleurs sont capables de réaliser une traduction bidirectionnelle UML <-> code source.

Notion de classe

Définir une classe c'est concevoir un **type d'objet nouveau** . Lorsque l'on déclare un type d'objet (une classe), on rassemble dans une même entité, à la fois les données **et** les opérations sur ces données.

Les données sont des constantes et/ou des variables, et les opérations sont des méthodes (fonctions).

Et les objets dans tout ça ?

Objet

Un objet est une **instance d'une classe**

Mais encore ?

Prenons l'exemple PHP suivant :

```
<php>
...
$quantite=2;
...
?>
```

Le type de `$quantité` est **int** (entier).

On dit que **$quantité** est *une instance de* `int` , une de ses multiples représentations.

L'équivalent en Java serait :

```
...
int quantité = 2;
...
```

Remarque : Java dispose, grosso modo, des mêmes types de base que PHP, qui, comme PHP, ne sont **pas des objets** .

Ramené à notre exemple, nous pouvons imaginer 2 instances de lignes de commandes, par exemple :

- une instance qui aurait comme valeur : refArticle :"1234", nomArticle : "Parapluie aluminium", prixU: 6.25, quantite:8
- une autre instance : refArticle :"1222", nomArticle : "Briquet gaz", prixU: 1.25, quantite:21

Comment créer de tels objets (instances) ?

C'est ce que nous allons mettre en oeuvre maintenant.

3.2. Exemple de conception d'une classe et création d'objets

Dans cette partie, nous allons modifier le comportement de notre programme, par l'intermédiaire de sa méthode principale `main` , afin qu'il **crée** 2 instances de la classe `LigneCmd` , conformément à notre exemple, puis les affiche.

A titre d'information, voici les différentes étapes que nous allons suivre :

1. Créer, **dans notre répertoire de projet** , un fichier, nommé **`LigneCmd.java`** et y inscrire le code de la classe Java présenté dans la figure précédente.
2. Compiler la classe `LigneCmd` (avec la commande `javac LigneCmd.java`). Corriger les erreurs si besoin.
3. Modifier la méthode `main` du fichier `AppTp1.java` dans le respect du scénario imaginé (création de 2 instances puis affichage de ces instances).
4. Mettre au point le tout.

Nous serons amenés à revenir plusieurs fois sur ces étapes (on parle d'itérations).

3.2.1. Classe `LigneCmd` : première version (incomplète)

Cette classe a déjà été présentée, la voici :

```
/
/**
*
* Fichier : LigneCmd.java
* Date : 02 nov 2003
* Représente une ligne de commande d'article
*
*/
public class LigneCmd {
  // attributs
  private String refArticle;
  private String nomArticle;
  private double prixU;
  private int quantite;

  // méthode(s)
  /**
  *  @return le montant total de la ligne
  */
  public double montant() {
    return this.prixU * this.quantite;
  }
}// LigneCmd
```

Nous venons de concevoir une structure contenant quatre variables et une fonction.

3.2.2. Compilation

Compilons cette classe :

```
[tp1]$ javac LigneCmd.java
[tp1]$
```

Ok. Vérifions l'existence du `.class`

```
[tp1]$ ll -tr
total 16
-rw-rw-r--    1 kpu      kpu     184 nov  1 21:02 AppTp1.java
-rw-rw-r--    1 kpu      kpu     477 nov  1 21:02 AppTp1.class
-rw-rw-r--    1 kpu      kpu     406 nov  2 16:59 LigneCmd.java
-rw-rw-r--    1 kpu      kpu     381 nov  2 17:01 LigneCmd.class
[tp1]$
```

3.2.3. Première modification de `AppTp1.java`

Nous interviendrons dans la méthode `main` .

Rappelons ce que nous souhaitons réaliser : créer 2 objets de type `LingeCmd` puis afficher leur valeur. Cela sera réalisé en trois étapes, détaillées ci-dessous :

1. **Déclarer** 2 variables de type `LigneCmd` .

```
...
LigneCmd ligneA;
LigneCmd ligneB;
...
```

2. **Créer** , un à un, les deux objets. Pour cela nous utiliserons l' **opérateur de construction d'objet `new`** .

```
...
ligneA = new LigneCmd();
ligneB = new LigneCmd();
...
```

3. **Afficher** l'état des deux objets.

```
...
System.out.println(ligneA);
System.out.println(ligneB);
...
```

En résumé, voici la fonction `main` ainsi redéfinie :

```
public class AppTp1 {
  static public void main(String[] arg) {
    LigneCmd ligneA;
    LingeCmd ligneB;
    ligneA = new LigneCmd();
    ligneB = new LigneCmd();
    System.out.println(ligneA);
    System.out.println(ligneB);
  }
}
```

Tentons une compilation :

```
[tp1]$ javac AppTp1.java
AppTp1.java:4: cannot resolve symbol
symbol  : class LingeCmd
location: class AppTp1
    LingeCmd ligneB;
    ^
1 error
[tp1]$
```

Une erreur de frappe est localisée ! Nous avons écrit `Linge` au lieu de `Ligne` . Du coup, le compilateur a cherché, et n'a pas trouvé, dans le répertoire de travail une classe nommée `Linge` . Corrigeons et recompilons.

```
[tp1]$ javac AppTp1.java
[tp1]$
```

Ok. Nous pouvons maintenant passer à l'exécution :

```
[tp1]$ java AppTp1
LigneCmd@f5da06 LigneCmd@bd0108
[tp1]$
```

Petite victoire ! le programme n'affiche pas les valeurs des objets mais leur type et leur adresse en mémoire.

Nous expliquons ici ce qui s'est passé :

Lorsque vous affichez l'état d'un objet, le système affiche en fait une représentation sous forme de chaîne de caractères (`String`) de l'objet.

Comment ?

En appelant implicitement la méthode `toString()` de l'objet.

En effet, le code ci-dessous est **exactement équivalent** à celui que nous avons conçu :

```
public class AppTp1 {
  static public void main(String[] arg) {
    LigneCmd ligneA;
    LingeCmd ligneB;
    ligneA = new LigneCmd();
    ligneB = new LigneCmd();
    System.out.println(ligneA .toString() );
    System.out.println(ligneB .toString() );
 }
}
```

Vérifiez cela, svp ; en modifiant notre fonction précédente `main` en conséquence.

...

Convaincu ?

En fait, par un mécanisme qui vous sera expliqué un peu plus tard, la classe `LigneCmd` est pourvue également des caractéristiques d'une autre classe, nommée `Object` . Or la classe `Object` dispose d'une méthode nommée `toString()` :

Class Object

java.lang.Object

---

public class **Object**

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

Method Summary	
boolean	**equals**(Object obj) Indicates whether some other object is "equal to" this one.
String	**toString**() Returns a string representation of the object.

Figure 3. Classe `Object`

Ce ne sont pas les seules méthodes héritées, mais concentrons nous sur `toString()` : le comportement de cette méthode héritée de `Object` **ne nous convient pas !** Nous allons donc la **redéfinir** .

Lorsque l'on redéfinit une méthode, il est nécessaire de **respecter le contrat initial** . Quel est celui de la méthode `toString()` ? Il est inscrit dans l'image précédente, à savoir :

```
public String toString()
   Returns a string representation of the object
```

Allons y :

```
/**
*
* Fichier : LigneCmd.java
* Date : 02 nov 2003
* Représente une ligne de commande d'article
*
*/

public class LigneCmd {
  // attributs
  private String refArticle;
  private String nomArticle;
  private double prixU;
  private int quantite;

  // méthode(s)
  /**
  *  @return le montant total de la ligne
  */
  public double montant() {
    return this.prixU * this.quantite;
  }

 /**
 * @return les caractéristiques de l'objet
 */
 public String toString() {
   String res = "Référence du produit : " + this.refArticle;
   res = res + "\nNom de l'article : " + this.nomArticle;
   //
   // à compléter
   //
   return res;
 }

}// LigneCmd
```

Il ne nous reste plus qu'à compiler et exécuter de nouveau le programme principal.

```
[tp1]$ javac LigneCmd.java
[tp1]$ java AppTp1
Référence du produit : null
Nom de l'article : null
Référence du produit : null
Nom de l'article : null
[tp1]$
```

Mouais... pas mal, mais que veut dire **`null`** ???

3.2.4. `null` : une valeur de référence

Si nous jetons un oeil sur le code que nous avons produit, à aucun moment nous n'avons affecté de valeur aux attributs `refArticle` , `nomArticle` , `prixU` et `quantite` .

Valeur par défaut des attributs

La valeur **`null`** est la valeur par défaut de tout attribut (variable) référençant un objet.

Mais alors... si `refArticle` à la valeur `null` (voir `toString()`), c'est qu'il référence un objet. Donc le type `String` est un type objet, une classe ?! oui, mais pas `int` , ni `double` . Nous présenterons plus loin un tableau des types non objets.

Exercice

- Complétez la méthode `toString()` de la classe `LigneCmd` afin de prendre connaissance de la valeur par défaut des variables de type numérique.

Comment allons nous affecter des valeurs aux objets ?

Voici une première version, qui ne fonctionne pas ! Nous la présentons parce qu'elle va permettre de nous poser de bonnes questions.

```
public class AppTp1 {
  static public void main(String[] arg) {
    LigneCmd ligneA;
    LingeCmd ligneB;
    ligneA = new LigneCmd();
    // refArticle :"1234", nomArticle : "Parapluie aluminium",
    // prixU: 6.25, quantite:8
    // Tentative de valorisation (infructueuse !)
    ligneA.refArticle="1234";

    ligneB = new LigneCmd();
    System.out.println(ligneA.toString());
    System.out.println(ligneB.toString());
  }
}
```

Voici une tentative de compilation :

```
[tp1]$ javac AppTp1.java
AppTp1.java:9: refArticle has private access in LigneCmd
     ligneA.refArticle="1234";
           ^
1 error
[tp1]$
```

La variable `refArticle` n'est pas accessible. En effet, si vous regardez le code source, on se rend compte que cette variable, et plus généralement tous les attributs de la classe `LgneCmd` sont déclarés **`private`** . Ceci est conforme au principe suivant :

Principe "Expert en information"

Un objet est seul responsable des informations dont il a la charge.

Donc, si on souhaite donner une valeur à chacun des attributs d'un objet, **on s'adresse à ses services.**

Question : Comment sont représentés les services d'un objet ?

Réponse : Par ses méthodes publiques (des fonctions).

Donc, en tant que concepteur de la classe `LigneCmd` , nous devons fournir ce qu'on appelle des **accesseurs** (accessors) permettant d'**interroger l'état** d'un objet et des **mutateurs**, ou accesseurs en écriture, en fait des fonctions d'accès en écriture, afin que les programmes appelants (les programmes utilisateurs) puissent modifier la valeur de certains attributs de l'objet. Un mutatueur, posséde un paramètre dont le type est compatible avec la valeur de l'attribut concerné.

3.2.5. Classe `LigneCmd` : getter/setter

Lorsque l'on souhaite fournir des accès aux attributs d'un objet, celui-ci doit fournir des fonctions adéquates. Les plus connues sont nommées **getter/setter** , respectivement pour **lire** (get : obtenir) et **écrire** (set : modifier) une information détenue par un objet.

Dans l'exemple ci-dessous, on fournit un accès en lecture et en écriture à l'attribut `refArticle` :

```
/**
*
* Fichier : LigneCmd.java
* Date : 02 nov 2003
* Représente une ligne de commande d'article
*
*/

public class LigneCmd {
  // attributs
  private String refArticle;
  private String nomArticle;
  private double prixU;
  private int quantite;

  // méthode(s)
  /**
  *  @return le montant total de la ligne
  */
  public double montant() {
    return this.prixU * this.quantite;
  }

/**
* @return les caractéristiques de l'objet
*/
public String toString() {
  String res = "Référence du produit : " + this.refArticle;
  res = res + "\nNom de l'article : " + this.nomArticle;
  //
  // à compléter
  //
  return res;
}

/**
*  @return la référence à l'article de la ligne
*/
public String getRefArticle() {
  return this.refArticle;
}

/**
*  Change la valeur de refArticle de la ligne
*  @param uneRefArticle une nouvelle référence
*/
public void setRefArticle(String uneRefArticle) {
  this.refArticle = uneRefArticle;
```

```
}
}// LigneCmd
```

Nous modifions le code de la fonction `main` de `AppTp1` :

```
public class AppTp1 {
  static public void main(String[] arg) {
    LigneCmd ligneA;
    LigneCmd ligneB;
    // refArticle :"1234", nomArticle : "Parapluie aluminium",
    // prixU: 6.25, quantite:8
    // Tentative de valorisation ok.
    ligneA = new LigneCmd();
    ligneA.setRefArticle("1234");

    // refArticle :"1222", nomArticle : "Briquet gaz",
    // prixU: 1.25, quantite:21
    ligneB = new LigneCmd();
    ligneB.setRefArticle("1222");

    System.out.println(ligneA);
    System.out.println(ligneB);
  }
}
```

Après une compilation réussie :

```
[tp1]$ java AppTp1
Référence du produit : 1234
Nom de l'article : null
Référence du produit : 1222
Nom de l'article : null
[tp1]$
```

Hourra ! Les valeurs ont bien été affectées aux bons objets.

Convention de nommage des getter/setter (extrait)

Par convention, le nom des méthodes `getter/setter` commence soit par `get` soit par `set` , suivi du nom de l'attribut avec le **premier caractère en majuscule** .

Exemple :

```
public class Etudiant {
  ...
  private double moyenne ;
  ...
  public double getMoyenne() {
   return this.moyenne;
  }
  public void setMoyenne (double moy) {
    this.moyenne = moy;
  }
  ...
}// Etudiant
```

Il ne vous reste plus qu'à finaliser la conception des classes `LigneCmd` et `AppTp1` en conséquence. Ce sera l'objet du premier des exercices proposés ci après.

3.3. Résumé

Nous avons pris connaissance des concepts généraux suivants:

- Un objet permet de regrouper dans une même structure à la fois des variables et des fonctions agissant avec et sur ces variables.
- Un objet est décrit, défini par une structure appelé **classe** . Les données portées par un objet, et représentées le plus souvent par des variables, sont appelées **attributs** ou **champs** . Les fonctions sont appelées **méthodes** .
- On a vu qu'un objet **cache** ses données aux programmes utilisateurs. Il en contrôle l'accès par des méthodes publiques, de type **getter/setter** .
- Une classe est composée d'une partie **définition des attributs** et d'une partie **définition des opérations (méthodes)** .

Concernant Java

- Le point d'entrée d'un programme autonome Java est une fonction `main` .
- Pour compiler et exécuter un programme on utilise les commandes `javac` et `java` .
- On crée un répertoire par projet.
- Les caractéristiques cachées d'un objet sont mentionnés `private` .
- Les caractéristiques publiées d'un objet sont mentionnés `public` .
- Pour construire un objet, on utilise l'opérateur **new** suivi du nom de la classe et d'un jeu de parenthèses.
- L'appel d'un service d'un objet se réalise par la **notation pointée** (parfois appelé *sélecteur de champs*). Par exemple `o.toString()` ou `ligneA.montant()` .

3.4. Exercice

1. Terminer le projet support de ce cours (`LigneCmd` et `AppTp1`), de sorte que l'exécution produise la sortie suivante :

```
[tp1]$ java AppTp1

 *** Référence du produit : 1234 ***
Nom de l'article : Parapluie aluminium
 => prix     : 6.25
 => quantité : 8
Montant : 50.0

 *** Référence du produit : 1222 ***
Nom de l'article : Briquet gaz
 => prix     : 1.25
 => quantité : 21
Montant : 26.25

Montant total : 76.25
[tp1]$
```

2. Créez un nouveau projet (un nouveau répertoire donc, par exemple `tp1Exo2`)

   Traduire en Java la classe décrite ci-dessous :

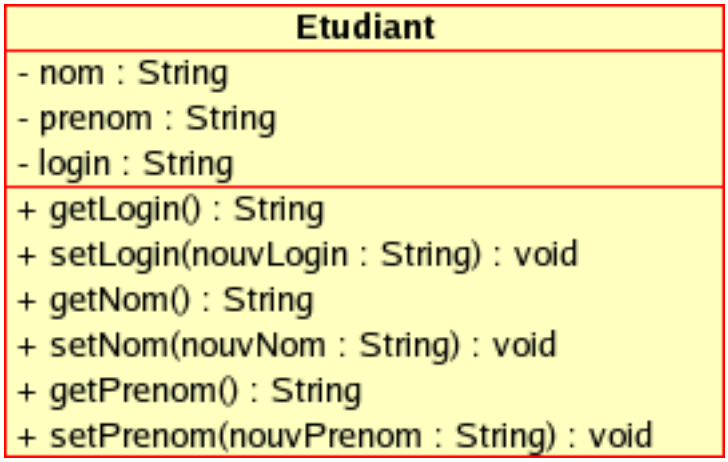


Figure 4. Diagramme de classe UML : Etudiant

Recopiez le programme principal suivant dans le répertoire du projet :

```
public class Application {
 static public void main(String[] args) {
    Etudiant e = new Etudiant();
    System.out.println(e.toString());
 }
}//Application
```

Mettre au point. Bien entendu, vous fournirez, **dans la méthode `main`** , un nom, prénom et login pour les tests (par exemple "Dupond, "polo", "pdupond"), et vous valoriserez l'objet créé par des fonctions de type `setter` . Vous redéfinirez la méthode `toString` de l'objet `Etudiant` . Exemple de sortie souhaitée :

```
[tp1exo2]$ java Application

*** Etudiant ***
nom    : Dupond
prenom : polo
login  : pdupond
[tp1exo2]$
```

3. La fonction main reçoit en paramètres les arguments de la ligne de commande, comme montré dans l'exemple :

```
public class Application {
 static public void main(String[] args) {

   Etudiant e = new Etudiant();
   System.out.println("Je reçois "+args.length+" argument(s).");
   int i = 0;
   while (ii<args.length) {
     System.out.println(args[i]);
     i++;
   }

   System.out.println(e.toString());
 }
}//Application
```

Exemple d'exécution **sans** arguments en ligne de commande :

```
[tp1exo2]$ java Application
Je reçois 0 argument(s).
Etudiant@f5da06
```

Exemple d'exécution **avec** arguments en ligne de commande :

```
[tp1exo2]$ java Application Dupond polo pdupond
Je reçois 3 argument(s).
Dupond polo pdupond Etudiant@f5da06
[tp1exo2]$
```

En analysant cet exemple, vous apprendrez à connaître le nombre d'arguments placés en ligne de commande. L'accès à un élément d'un tableau est équivalent à ce que nous avons déjà rencontré avec PHP.

On vous demande de valoriser les attributs l'objet de type `Etudiant` , lorsque l'utilisateur fournit 3 arguments, comme dans l'exemple précédent, à savoir le nom, prénom et login.

```
[tp1exo2]$ java Application Amigo jacko jamigo
*** Etudiant ***
nom    : Amigo
prenom : jacko
login  : jamigo
[tp1exo2]$
```

4. Trier 3 lignes de commandes : Reprise du premier exercice (extension)

   Créer, dans le `main` , une nouvelle ligne de commande concernant l'article de référence "x32", nomArt : stylo, prix : 0,33, quantite : 10.

   Vous modifiez votre programme afin qu'il affiche le nom et le prix, comme ici :

```
[tp1]$
java AppTp1
Parapluie aluminium (6.25)
Briquet gaz (1.25)
stylo (0.33)
[tp1]$
```

   Ensuite, vous interviendrez sur le programme afin qu'il affiche les lignes par ordre croissant de prix unitaires, sans connaître à l'avance leur valeur bien entendu :-)

   Remarque : ce n'est pas forcément simple à faire ; il faut jouer avec les if, et éventuellement les & (un ET logique), Exemple :

```
 //if (ligneA.getPrixU() <= ligneB.getPrixU() &
//    ligneA.getPrixU() <= ligneC.getPrixU())
// alors que peut-on affirmer ici ?
```

   Exemple de sortie souhaitée :

```
[tp1]$ java AppTp1
stylo (0.33)
Briquet gaz (1.25)
Parapluie aluminium (6.25)
[tp1]$
```

5. Sur la base de l'exercice précédent, une fois terminé, vous introduirez, **au début du `main`** , 3 nouvelles variables :

```
LigneCmd lig1, lig2, lig3;
```

Où `lig1` référencera un objet de prix unitaire égal au minimum des prix unitaires, `lig3` référencera un objet de prix unitaire égal au maximum des prix unitaires et `lig2` référencera le troisième objet.

Modifier votre programme de sorte que le bloc d'instructions suivant, **situé à la fin de la méthode main** , affiche A TOUS les coups, les lignes triées sur le prix unitaire :

```
System.out.println(lig1.getNomArticle() + "(" +
   lig1.getPrixU() + ")");
System.out.println(lig2.getNomArticle() + "(" +
   lig2.getPrixU() + ")");
System.out.println(lig3.getNomArticle() + "(" +
   lig3.getPrixU() + ")");
```

Variable de type référence

Plusieurs variables peuvent référencer le même objet. Exemple :

```
lg1 = ligneA;
   // Si ligneA référence un objet,
   // alors lg1 et ligneA référencent le
   // même objet en mémoire.

  // c'est à dire que
  System.out.println(l1);
  // et
  System.out.println(ligneA);

  // sont alors 2 instructions parfaitement
  // équivalentes.
```

Une variable référence ne peut contenir que 2 types de valeur : **`null`** (si elle ne référence pas un objet) ou une **référence à un objet en mémoire** .

Remarque : Il est courant de confondre une variable référence avec l'objet qu'elle référencie. Exemple, on dit " *l'objet ligneA* ", alors que l'on devrait dire " *l'objet référencé par la variable `ligneA`* ".

Le pb consiste donc à affecter les bonnes valeurs de références à `lg1, lg2 et lg3` , celles détenues par `ligneA, ligneB et ligneC` .

=>une solution ici : LigneCmd.java [7], AppTp1.java [8]

6. Reprise du deuxième exercice

[7] solution/tp1/exo1/LigneCmd.java
[8] solution/tp1/exo1/AppTp1.java

Si le main ne reçoit que 2 arguments (nom et prénom), il affecte la valeur " (chaîne vide) au login. Si le main ne reçoit qu'un argument, il sera considéré comme le nom, le prénom sera affecté à " (chaîne vide) et le login à "anonymous".

Si il n'y a aucun argument, la valeur "anonymous" sera affectée au nom et au login, et la valeur " (chaîne vide) au prénom.

Rendre des copies écran des différents tests.

=>une solution ici : Etudiant.java [9], AfficheEtudiant.java [10]

4. Java et les concepts de base de l'objet

4.1. Intro

Nous avons présenté lors du chapitre précédent qu'un objet est composé d' **attributs** - nommés aussi champs (*field*), de variables d'instance - toutes déclarées **privées** (*private*) et de **méthodes** - nommées parfois opérations ou fonctions - et déclarées publiques (*public*).

Ainsi, les **services** proposés par un objet sont représentés par des fonctions.

Les services "les plus simples" sont ceux qui gèrent les accès aux informations détenues par les objets. Ce que nous avons appelé les accesseurs ou **getter/setter** (accesseurs en lecture et/ou écriture).

Cette technique, qui consiste à cacher les attributs, est une façon de réaliser un des concepts fondateurs de l'objet, nommé **encapsulation** . Les trois autres sont **l'héritage, l'abstraction et le polymorphisme** , nous les présentons succinctement ici :

- L' **héritage** est un mécanisme qui permet de concevoir une classe sur la base d'une autre.

  Par exemple notre classe `Etudiant` dispose d'une méthode `toString` tout simplement parce qu'elle **hérite de la classe `Object`** .

[9] solution/tp1/exo2/Etudiant.java
[10] solution/tp1/exo2/AfficheEtudiant.java

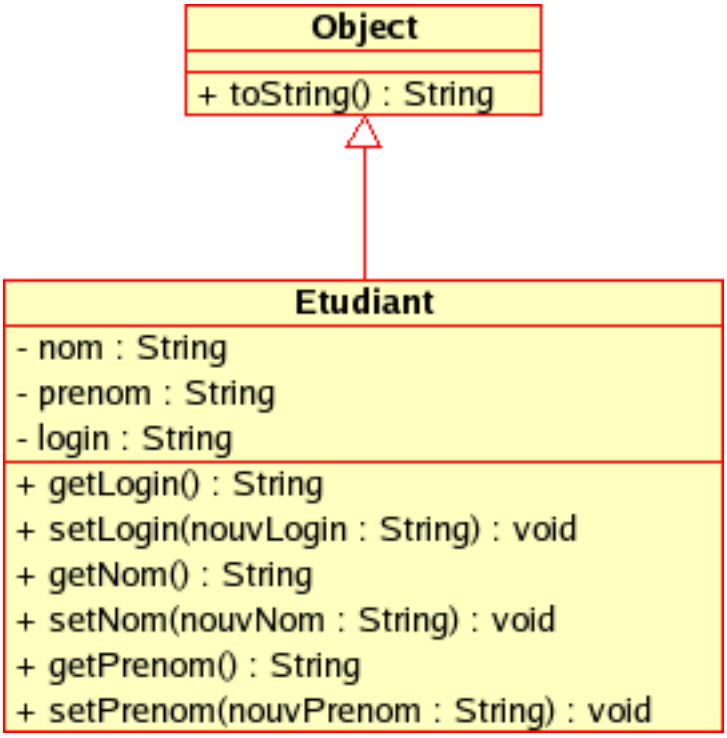


Figure 5. Etudiant hérite de Object

```
class Etudiant extends Object { ... }
```

- Le **polymorphisme** (de poly et morphe : plusieurs formes) est un mécanisme qui permet de **redéfinir le comportement de méthodes héritées** .

  C'est ce que nous avons fait dans la classe `Etudiant` , lorsque nous avons redéfini la méthode `toString` . Ce n'est pas plus compliqué que cela.

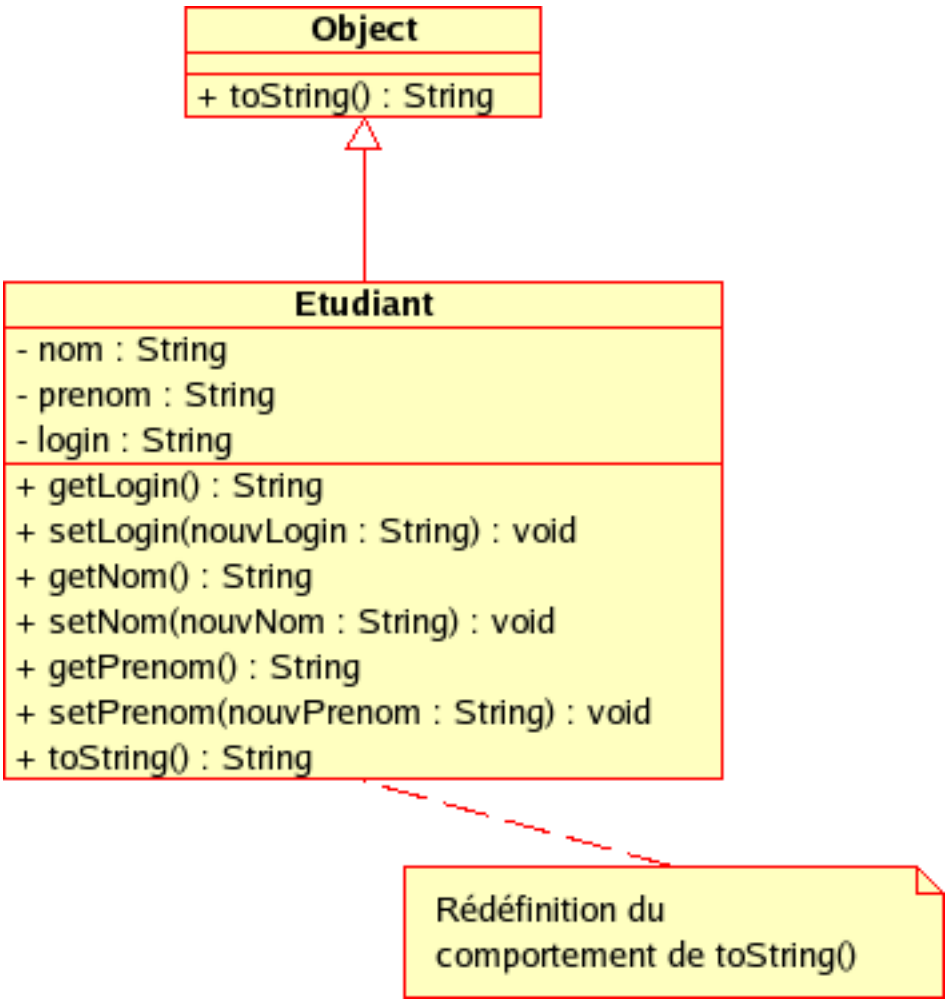


Figure 6. Etudiant hérite de Object

- L' **abstraction** concerne des classes déclarant des méthodes (fonctions) dont certaines n'ont pas de corps. Charge alors aux classes descendantes (celles qui hériteront de cette classe) de fournir un corps aux méthodes n'en disposant pas (nommée également *méthodes abstraites*). Une notion que nous étudierons prochainement.

4.2. Tout n'est pas objet

En Java, une variable est de type **primitif** et ou **référence** .

De ce fait le langage Java ne peut être qualifié de langage "purement objet".

Java traite les variables de type primitif selon une sémantique par valeur et les variables de type objet (classe, interface, tableau - array -) selon une sémantique par référence (ce n'est pas la valeur de la variable qui nous intéresse, mais l'objet référencé). Exemple :

Le fragment de code suivant n'EST PAS VALIDE :

```
...
int x = 2;
System.out.println(x .toString() ); // INTERDIT !!
// x ne référence pas un objet.
```

```
// int n'est pas une classe d'objet, mais un type primitif
...
```

Java propose 9 types primitifs, leur nom est **entièrement en minuscule** (`int, boolean, char, byte, float, double, long, short, void`) :

4.2.1. Types primitifs

Les types primitifs ne sont pas des classes.

Type Primitif	Taille	Minimum	Maximum	Valeur par défaut	Classe
types non signés					
boolean	16-bit	–	–	false	Boolean
char	16-bit	Unicode 0	Unicode 2^{16} - 1	\u0000	Character
types signés					
byte	8-bit	-128	+127	0	Byte
short	16-bit	-2^{15}	$+2^{15} - 1$	0	Short
int	32-bit	-2^{31}	$+2^{31} - 1$	0	Integer
long	64-bit	-2^{63}	$+2^{63} - 1$	0	Long
float	32-bit	IEEE754	IEEE754	0.0	Float
double	64-bit	IEEE754	IEEE754	0.0	Double
void	–	–	–	–	Void

Figure 7. Types primitifs de Java

Chaque type primitif a son correspondant en type Objet. Alors pourquoi pas le "tout objet" ?

Les créateurs du langage Java ont souhaité conserver l'utilisation des types primitifs dans un but d'efficacité. Un point de vue qui sera peut-être remis en cause avec la version 1.5 du jdk... à suivre.

Les "objets" de type primitif sont automatiquement créés sur la pile, et leur valeur confondue avec leur identificateur (sémantique par valeur). Toutes les autres entités sont de véritables objets, créés sur le tas (et non sur la pile) accessible via un ou plusieurs identificateurs; des variables de type référence (sémantique par référence).

Le développeur a la possibilité de représenter une valeur de type primitif par un objet. Ceci est possible car il existe une classe pour chaque type primitif (colonne de droite du tableau des types primitifs). Ces classes sont porteuses d'informations telles que les limites du domaine de définition. Voici un exemple.

```
public final class Float extends Number {
...
```

```
  public static final float MIN_VALUE = 1.4e-45f;
  public static final float MAX_VALUE = 3.4028235e+38f;
  public static final float NaN = 0.0f/0.0f;

...

}
```

`NaN` signifie Not-a-Number.

Le fragment de code suivant EST VALIDE :

```
...
int x = 2;
Integer y = new Integer(x);
System.out.println(y.toString()); // ok !
//  x ne référence pas un objet.
//  y référence un objet.
...
```

L'utilisation d'un objet de type `Integer` est complètement inutile dans cet exemple. Par contre cette technique s'avère nécessaire lorsque l'on souhaite stocker des entiers (par exemple) dans des collections. Une problématique qui sera étudiée prochainement. Retenons pour l'heure que les types primitifs ne sont pas des classes d'objets, mais qu'ils peuvent être représentés par ces derniers.

De plus, les classes qui **enveloppent** (*wrapper*) les types primitifs fournissent quelques **fonctions de conversion** bien pratiques, tel que :

- **Conversion de type `String` vers un type primitif** :

  ```
  ...
   String s1 = "123";
   String s2 = ".23";

   int    x1 = Integer.parseInt(s1);
   double x2 = Double.parseDouble(s2);
   float  x3 = Float.parseFloat(s2);
   float  x4 = Float.valueOf(s2); // même effet
  ...
  ```

  ... une méthode de la forme `typePrimitif ClasseWrapper.parseTypePrimitif(String s)` .

- diverses autres fonctions de conversion, comme par exemple la classe `Integer` qui propose par exemple les méthodes `static String toBinaryString(int i)` , `static String toHexString(int i)` , etc. voir l'aide en ligne.

4.3. Création et initialisation d'un objet : les constructeurs

Nous avons vu que, pour créer un objet, l'opérateur **`new`** s'impose :

```
public class Application {
  static public void main(String[] args) {
    Etudiant e = new Etudiant();
    System.out.println(e);
```

```
    // affiche "Nom : null, Prenom : null, login : null"
  }
}//Application
```

A droite du `new` nous avons écrit le **nom de la classe suivi d'un jeu de parenthèses** . Or, nous savons que lorsqu'on met des parenthèses après un identificateur, ce dernier est supposé être une **fonction** . Si on jette un coup d'oeil au code source de la classe `Etudiant` , on ne trouvera nulle part de définition d'une telle fonction ?!

Est-elle héritée de la classe de base `Object` ? Non, certainement pas, les concepteurs de la classe `Object` ne sont pas sensés savoir que l'on construira un jour une classe `Etudiant` !

En fait une telle fonction s'appelle **constructeur** . Elle sert à **initialiser l'état de l'objet à sa création** . Si vous n'en définissez pas dans votre classe, Java le fait pour vous. Donc, sans que vous vous en doutiez, la classe `Etudiant` dispose d'un **constructeur par défaut** , qui ne fait rien.

Caractéristiques d'un constructeur

- **Même nom que sa classe**

  Exemple :

  ```
  static public void main(String[] args) {
    Object e = new Etudiant();
    System.out.println(e.toString());
       // affiche "Nom : null, Prenom : null, login : null"
  }
  ```

  On appelle ici le constructeur par défaut de la classe `Etudiant` , précisant ainsi que l'on construit un objet de cette classe.

  Bien que la variable soit déclarée de type `Object` , cet exemple est valide car la classe `Etudiant` est une sous-classe de `Object` , et que l'on appelle la méthode `toString` connue de `Object` .

- **Pas de type de retour**

  Exemple :

  ```
  public class Etudiant extends Object {
    private String nom;
    private String prenom;
    private String login;
    public Etudiant() {    }
   ...
  }
  ```

  On remarquera l'absence de type entre l'attribut de visibilité (`public`) et le nom du constructeur.

4.3.1. Exemple de constructeur

Il n'est pas conseillé de laisser Java concevoir pour vous un constructeur par défaut. Nous allons en créer un dans la classe `Etudiant` . Nous en profiterons pour initialiser les attributs :

```
public class Etudiant extends Object {
  private String nom;
```

```
  private String prenom;
  private String login;
  public Etudiant() {
     this.nom    = "anonymous";
     this.prenom = ";
     this.login  = "anonymous";
  }
  ...
}
```

Du coup, le programme suivant :

```
public class Application {
  static public void main(String[] args) {
     Etudiant e = new Etudiant();
     System.out.println(e);
       // affiche "Nom : anonymous , Prenom : , login : anonymous"
  }
}//Application
```

... n'affichera pas la valeur null, mais les valeurs que vous avez définies dans le constructeur, au lieu des valeurs par défaut de Java (voir le tableau des types primitifs).

Valeur par défaut

En absence de constructeur, Java affecte aux attributs de l'objet leur **valeur par défaut** : **`0`** pour les valeurs numériques (y compris le type char, voir le tableau des types primitifs), **`false`** pour le type boolean et **`null`** pour le type référence (variable de type objet).

Comme on peut s'y attendre, on peut passer des paramètres à un constructeur. Exemple :

```
public class Etudiant extends Object {
  private String nom;
  private String prenom;
  private String login;
  public Etudiant() {
    this.nom    = "anonymous";
    this.prenom = ";
    this.login  = "anonymous";
  }
  public Etudiant(String nom, String prenom) {
     this.nom    =  nom;
     this.prenom =  prenom;
     this.login  = ";
  }
  ...
}
```

Du coup, le programme suivant se contente de sélectionner le bon constructeur :

```
public class Application {
  static public void main(String[] args) {
    Etudiant e;
    if (args.length==2) {
      e = new Etudiant(args[0], args[1]);
    }else {
      e = new Etudiant();
    }
    System.out.println(e);
  }
}//Application
```

On voit tout de suite l'intérêt des constructeurs : on évite ainsi l'usage des setter pour initialiser les objets. Moins de lignes à écrire !

4.3.2. Exercice

On souhaite reprendre une solution d'un exercice précédent.

Rappel de l'énoncé : *Afficher la valeur des attributs (nom, prenom, login) d'un objet `Etudiant` : Si le main ne reçoit que 2 arguments (nom et prénom), il affecte la valeur " (chaîne vide) au login. Si le main ne reçoit qu'un argument, il sera considéré comme le nom, le prénom sera affecté à " (chaîne vide) et le login à "anonymous". Si il n'y a aucun argument, la valeur "anonymous" sera affectée au nom et au login, et la valeur " (chaîne vide) au prénom.*

La solution proposée (Etudiant.java [11], AfficheEtudiant.java [12]) pourrait être bien plus élégante en utilisant des constructeurs (comme présenté dans l'exemple ci-dessus, qui sélectionne un constructeur approprié).

A vos claviers ! (vous devrez donc intervenir sur les classes `Etudiant` et `AfficheEtudiant`).

Nous présentons, dans la série suivante, d'autres moyens d'interagir avec l'utilisateur que celui qui consiste à exploiter les arguments en ligne de commande (paramètres de la fonction `main`).

5. Interaction utilisateur <--> application

5.1. Exemple de fonctions d'interaction avec l'utilisateur (clavier/écran)

Nous changeons de registre ici, et présentons quelques moyens pratiques d'interagir avec l'utilisateur d'un programme écrit en java.

5.1.1. Afficher des informations en mode Console

On utilise l'instruction `System.out.println()` ou `System.out.print()` - même fonction mais ne provoque pas de retour chariot. Exemple :

```
public class Application {
   static public void main(String[] args) {
      System.out.print("He");
      System.out.println("llo !");
   }
}//Application
```

Ce qui donne :

```
$ java Application
Hello !
$
```

5.1.2. Afficher des informations en mode Graphique

On utilise l'instruction **`showMessageDialog()`** .

Attention, cette méthode n'EST PAS UNE MÉTHODE D'INSTANCE (elle est déclarée `static`).

[11] solution/tp1/exo2/Etudiant.java
[12] solution/tp1/exo2/AfficheEtudiant.java

C'est une fonction disponible dans le module (la classe) `JOptionPane` . On **préfixera le nom de cette méthode par le nom de la classe dans laquelle elle est définie** , soit ici `JOptionPane` , qui se trouve dans le package `javax.swing` :

```
public class Test {
  static public void main(String[] args) {
    Object e = new Etudiant();
    javax.swing.JOptionPane.showMessageDialog(null, e);
  }
}
```

Ce qui donne :

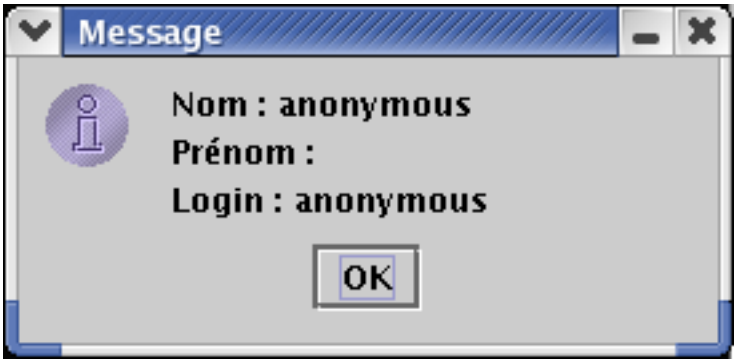


Figure 8. JOptionPane.showMessageDiaglog

En absence de contexte graphique, nous pouvons utiliser l'entrée standard avec un echo à l'écran. C'est cette technique que nous utilisons ci-après.

5.1.3. Lire des informations entrées au clavier en mode Console

C'est un peu plus compliqué. Nous vous proposons ici une fonction qui réalise cela, comme celle que nous vous avons proposé en PHP. Cette fonction utilise des objets spécialisés dans la gestion de flux et une structure de gestion des exceptions (`try/catch)` sur lesquels nous vous demandons de ne pas trop porter attention pour l'instant.

Nous vous présentons donc une classe, fortement réutilisable, que nous nommerons `Console` :

```
import java.io.*;

public class Console {

static public String lireClavier(String msg) {
  // variable contenant la valeur qui sera retournée
  String res = null;
  // affichage du message
  System.out.print(msg);

  // tentative de lecture de l'entrée standard
  // référencée par System.in
  try {
    BufferedReader stdin =
        new BufferedReader(new InputStreamReader(System.in));
    res = stdin.readLine();
  }
  catch (IOException e) {
    // affiche un message d'erreur
    System.out.println("ERREUR : " + e.getMessage());
```

```
  }

  // retoune le résultat
  return res;
}

// un test
static public void main(String[] args) {
  String nom = Console.lireClavier("Entrez votre nom : ");
  System.out.println("Vous vous appelez " + nom);
}

}//Console
```

Ce qui donne :

```
$ javac Console.java
$ java Console
Entrez votre nom : kpu
Vous vous appelez kpu
```

Pour le moment, vous pouvez simplement placer le fichier `Console.class` dans vos répertoires projet, afin de bénéficier de ses services.

5.1.4. Lire des informations entrées au clavier en mode Graphique

Nous utiliserons la méthode `showInputDialog` de la classe `JOptionPane` .

L'exemple ci-dessous, demande un nom à l'utilisateur. Si ce dernier clique sur `Annuler` , la fonction rendra la valeur `null` , mais si l'utilisateur ne saisit aucune valeur et clique sur OK la fonction rendra une chaîne vide. Dans le cas où l'utilisateur fournit une valeur, elle est affectée au nom de l'étudiant.

```
import javax.swing.*;
public class Test {
  static public void main(String[] args) {
    Etudiant e = new Etudiant();
    String nom = JOptionPane.showInputDialog("Entrez un nom");

    if (nom == null) {
       nom = "";
    }

    if (!nom.equals("")) {
       e.setNom(nom);
    }

    javax.swing.JOptionPane.showMessageDialog(null, e);
    //System.out.println(e.toString());
  }
}
```

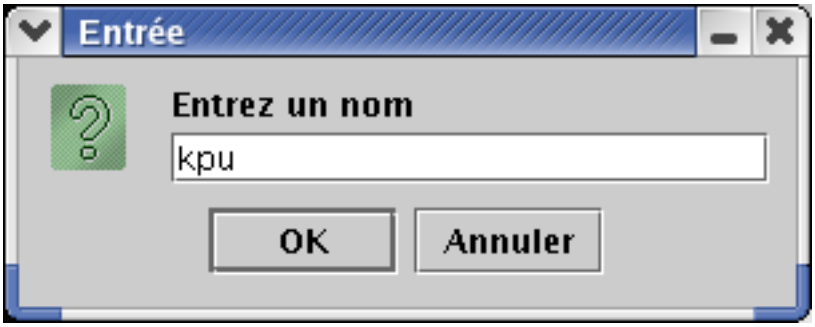


Figure 9. JOptionPane.showInputDiaglog

Vous êtes bien entendu invité à tester ce programme.

Plus d'info sur http://www.linux-france.org/prj/edu/archinet/DA/ [13]

Test d'égalité

Pour comparer deux valeurs primitives on utilise (comme en PHP) l'opérateur `==` . Avec les objets, on utilise la méthode **`equals`** , comme dans l'exemple ci-dessus, qui compare **l'état des objets** , et non leur référence.

5.2. Exercice

On souhaite réaliser un programme qui tire "au hasard" et dans le secret un nombre compris entre 0 et *n* . La valeur de *n* est fournie par l'utilisateur au début du programme.

Le jeu consiste à inviter l'utilisateur à trouver le nombre secret en un minimum de coups.

A chaque fois que l'utilisateur donne un nombre erroné, le programme indique si le nombre entré est supérieur ou non au nombre à découvrir.

Voici le programme principal, qui crée un objet de type `Devine` , une classe que vous devez concevoir.

```
/**
*  fichier : JeuDevine.java
*
*  Lycée Léonard de Vinci - Melun
*  BTS IG - première année
*
*  15 novembre 2003
*/
public class JeuDevine {
  public static void main(String[] args){
    String str_n;
    String joueur;
    joueur  = Console.lireClavier("Entrez votre nom : ");
    str_n   = Console.lireClavier("Entrez un nombre > 0 : ");

    // convertir la chaîne str_n en entier
    int n = Integer.parseInt(str_n);

    // création d'un représentant du jeu (un objet)
    // en fournissant la limite supérieure de
    // l'intervalle dans lequel le nombre secret
    // sera extrait : [0..n[
    // ainsi que le nom du joueur
    Devine devine = new Devine(n, joueur);
```

[13] http://www.linux-france.org/prj/edu/archinet/DA/

```
    // on demande au jeu de tirer un nombre aléatoire
    devine.determineNombreSecret();

    // séquence d'interactions avec l'utilisateur
    // jusqu'à ce que l'utilisateur trouve le nombre secret
    // ou abandonne.
    devine.utilisateurJoue();

    // affiche le nombre secret
    System.out.println("Le nombre secret est : " +
        devine.getNbSecret());

    System.out.println("Merci "+devine.getJoueur() + ".");

    // System.out.println("Profil utilisateur : " +
    //       devine.getProfilUtilisateur());
  }
```

1. Créer un répertoire de projet.
2. Placer y les fichiers `JeuDevine.java` et `Console.java` .
3. Concevoir, dans `Devine.java` , la classe `Devine` , afin d'exécuter le scénario représenté par la fonction `main` de `JeuDevine` .

   Voici le diagramme de classe de l'application à développer :

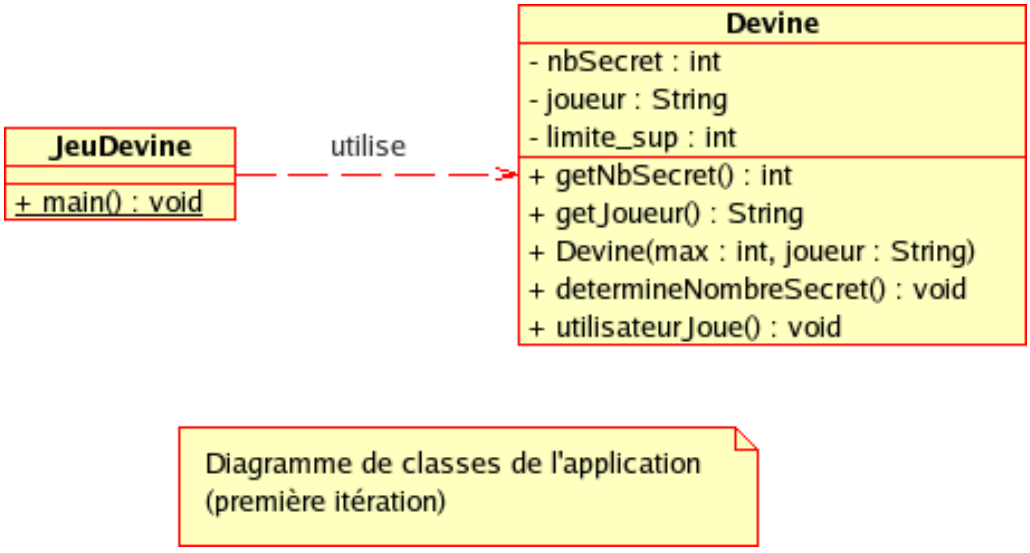


Figure 10. Architecture de l'application, première version

4. Mettre au point le jeu.
5. Amélioration possible(à utiliser avec l'aide de l'annexe ci-après). Sachant que, si le joueur tient compte de ce que lui affiche le programme à chaque coup loupé (à savoir si le nombre à rechercher est supérieur ou non au nombre précédemment introduit par le joueur), ce dernier découvrira le nombre secret en un nombre de coup au plus égal à $log_2(n)$.

   Vous déterminerez donc le profil du joueur en fonction de cette valeur et du nombre de coups réellement joués :

- **Chanceux (ou intuitif...)** : Si le nombre de coups joués est inférieur à la valeur arrondie au supérieur de $log_2(n)$.
- **Rationnel** : Si le nombre de coups joués est inférieur ou égal à la valeur arrondie au supérieur de $log_2(n)$.
- **Pas malin** : Si le nombre de coups joués est supérieur à la valeur arrondie au supérieur de $log_2(n)$.

Vous mettrez au point la méthode `public String getProfilUtilisateur()` de la classe `Devine` .

5.2.1. ANNEXE

- Pour arrondir au supérieur un nombre réel, on utilise la fonction `Math.ceil` de java.
- Il n'y a pas de fonction logarithme à base 2 en java. Nous la traduirons par la formule utilisant le logarithme naturel représenté par la fonction `Math.log` de la bibliothèque `Math` de java :

  Rappel : $\log_2(n) = x$ si $2^x = n$

  Par exemple : $\log_2(1024) = 10$ car $2^{10} = 1024$

  Sachant que $\log_2(n) = \log_e(n) / \log_e(2)$, nous déduisons la fonction suivante :

```
static public int log2(int n) {
  // on force un retour en int
  return (int) Math.ceil(Math.log(n)/Math.log(2));
}
```

- Vous utiliserez la fonction `nextInt` d'un objet de la classe `Random` (du package `util`), pour déterminer la valeur du nombre secret à découvrir.

```
package : java.util
classe  :  Random

méthode :

  public int nextInt(int n)
  // Retourne un nombre pseudo aléatoire
  // compris dans l'intervalle : [0..n[
```

Attention, la méthode `nextInt` n'est pas `static` . Vous devez donc créer un objet de la classe `Random` (avec new) pour pouvoir l'utiliser.

Bonne programmation !

6. Exercices de révision

Extraits de sujets d'examens 2000-2002 - STS IG lycée Léonard de Vinci 77000 Melun -

6.1. Question I - Truc : Valeur par défaut et constructeur (7 points)

Soit la classe `Truc`

```
class Truc {
   private String nom;
   private double prix;

   public void setNom(String nom) {
     this.nom=nom;
   }
   public void setPrix(double prix) {
     this.prix=prix;
   }
}
```

Et le programme suivant :

```
public class Test {
  static public void main(String[] args) {
    Truc t1 = new Truc();
    // dans quel état est le truc
    // référence par t1 ? ===>  REPONSE A
    t1.setNom("bidule");
    t1.setPrix(12.3);
    // dans quel état est le truc t1 ? ===> REPONSE B
  }
}
```

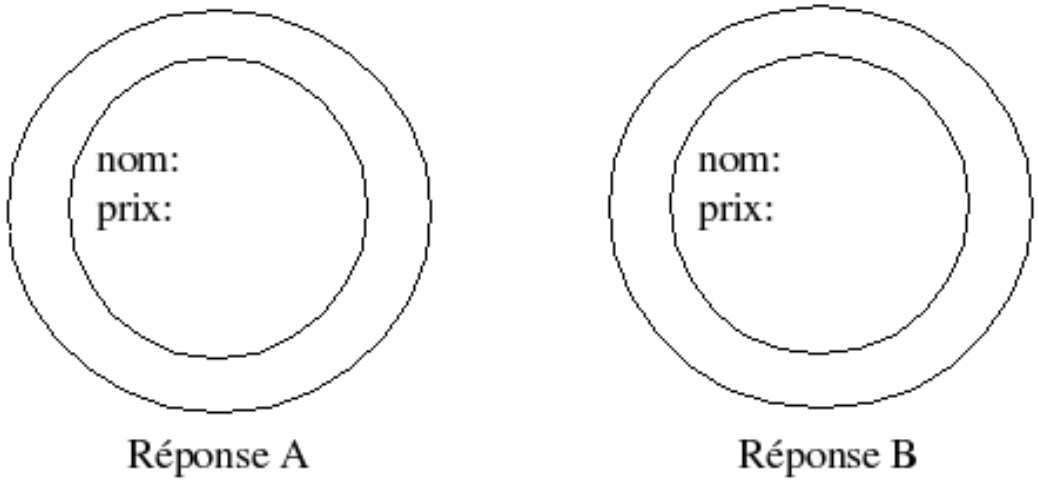


Figure 11. Répondre aux questions A et B en complétant la figure ci-dessous (2 points)

Justifier vos réponses (**2 points**)

Comment faire pour initialiser un objet à sa création ? Fournir un exemple (**3 points**)

=>une solution ici : exemple de réponses aux questions [14]

6.2. Question II - Voiture : Activité de conception (4

[14] solution/Truc.html

points)

On vous demande de concevoir une classe nommée `Voiture` permettant au programme ci-dessous de fonctionner.

```
class Application {

  public void run() {

     Voiture v = new Voiture("Titine", 7);
       // création d'une voiture nommée "Titine" de 7 chevaux

     System.out.println(v.donneVitesse());
     // affiche : 0

     v.passeVitesseSuperieure();
       // passe la première vitesse

     v.passeVitesseSuperieure();
       // passe la seconde vitesse

     v.passeVitesseSuperieure();
       // passe la troisième vitesse

     v.retrograde();
       // retourne en deuxième

     System.out.println(v.donneVitesse());
       // affiche : 2

     System.out.println(v);
       // affiche : Titine, 7 chevaux
  }

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

Remarque : Ne concevez que le **strict minimum** (attributs, constructeur et méthodes) nécessaire au fonctionnement du programme `Application` .

6.3. Question III - Voiture : Paramétrage du nombre de vitesses de la boîte (5 points)

On souhaite ajouter un attribut `maxVitesse` , correspondant au nombre de vitesses disponibles pour une voiture donnée (par exemple 5).

On vous demande d'aménager la classe `Voiture` en 2 étapes :

- Déclarer l'attribut et modifier le constructeur en conséquence. (**2 points**)
- Faire en sorte que les méthodes `passeVitesseSuperieure()` et `retrograde()` laissent toujours la voiture dans un **état cohérent** (pas de vitesse impossible). (**3 points**)

6.4. Question IV - Voiture : Ajout d'une méthode, et test unitaire (4 points)

On vous demande d'ajouter à la classe `Voiture` :

- Une méthode nommée `estPointMort` qui rend la valeur booléenne *vrai* si la vitesse en cours est nulle (valeur zéro) et *faux* dans le cas contraire. (**1 point**)

Afin de vérifier le bon fonctionnement de la méthode `estPointMort()` , on vous demande de compléter la classe `TestVoiture` , et plus particulièrement de :

- Concevoir la méthode `testEstPointMort()` . Cette méthode devra créer une voiture, puis tester la méthode `estPointMort` en changeant plusieurs fois l'état de l'objet.

  La méthode `testEstPointMort` affichera `OK` si la valeur de la méthode est conforme à celle que vous attendez, `ERREUR` dans le cas contraire.

  Remarque 1 : les valeurs booléennes de Java sont représentées par les littéraux **`true`** et **`false`** .

  Remarque 2 : vous serez évalué sur la qualité de votre test, qui devra comporter au moins 3 appels de la fonction `estPointMort` . (**3 points**)

```
public class TestVoiture {
  /**
  * Test la méthode donneVitesseSuperieure
 */
  public void testDonneVitesseSuperieure() {
     Voiture v = new Voiture("Deuche", 2);
     v.passeVitesseSuperieure();
     if (v.donneVitesse() == 1) {
       System.out.println("OK")
     }
     else {
       System.out.println("ERREUR")
     }

     etc.

  }

  /**
  * Test la méthode estPointMort :: A FAIRE ::
  */
  public void testEstPointMort() {

     Voiture v;









  }

  static public void main(String[] args){
     TestVoiture test = new TestVoiture();
     test.testEstPointMort();
```

```
    }
}
```

=>une solution ici : Voiture.java [15]

6.5. Question V - MicroOrdi : (conception) Ajout de méthodes (8 points)

Une entreprise spécialisée dans la vente de micro-ordinateur, souhaite informatiser la gestion de ses produits. Pour l'occasion on ne retiendra d'un micro-ordinateur que les caractéristiques suivantes : processeur, taille de la mémoire vive (en Mo), capacité de stockage du disque dur (en Go) et prix. Voici une représentation UML de sa classe.

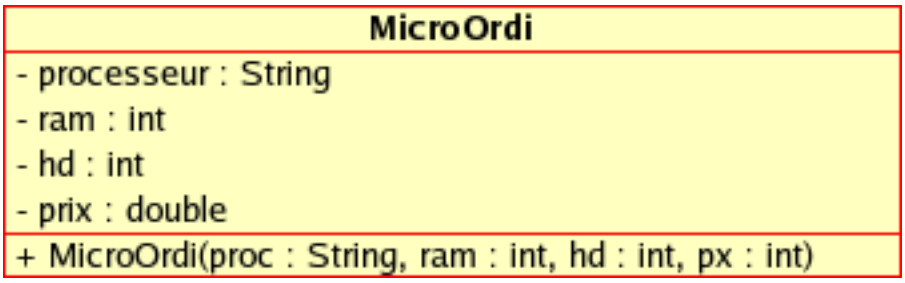


Figure 12. Classe MicroOrdi

Proposez une représentation Java de cette classe (avec accesseurs et modificateurs des attributs privés) (**4 points**).

Certains clients de la société souhaitent pouvoir installer sur leur machine à la fois un linux du monde libre et un Windows de Microsoft, en dual boot. On considère qu'une machine ne peut accepter confortablement ce type d'installation que si elle dispose d'au moins 128 Mo de Ram et de 6 Go de disque dur. De plus, une machine compatible dual boot WinLux et disposant d'un disque de capacité supérieure ou égale à 10 Go pourra opter pour une distribution développeur de Linux.

On désire ajouter à la classe MicroOrdi les méthodes `estCompatibleDualBootWinLux()` , et `estOkPourDevLinux()` .

- Représenter par un diagramme de Classe UML, la classe MicroOrdi modifiée, SANS LES ACCESSEURS/MODIFICATEURS. (**2 points**)
- Donner la représentation Java de ces méthodes (on ne vous demande pas de réécrire la classe dans son ensemble mais seulement les nouvelles méthodes). (**4 points**)

6.6. Question VI - MicroOrdi : Création d'objets et tests unitaires (6 points)

On souhaite construire un programme qui teste la classe MicroOrdi.

Pour cela, vous concevez un programme qui crée deux objets de la classe MicroOrdi.

Le premier ordinateur a comme processeur "Intel 4", une capacité RAM de 256, 40 Go de HD, coûtant 1500 euro.

[15] solution/Voiture.java

Le second ordinateur a comme processeur "AMD", une capacité RAM de 64, 6 Go de HD, coûtant 600 euro.

Après avoir affiché les caractéristiques d'un ordinateur, le programme testera le bon fonctionnement des méthodes `estCompatibleDualBootWinLux()` et `estOkPourDevLinux()` . Il affichera automatiquement le message **"OK"** si les méthodes répondent correctement (sachant que vous connaissez à l'avance le résultat) et **"ERREUR"** dans le cas contraire.

- Réalisez le programme - classe Application et méthode main. On ne cherchera pas à exploiter la liste des arguments (args) de la méthode main (**6 points**)

7. Abstraction

7.1. Intro

Nous présentons ici le dernier des quatre concepts fondamentaux de l'objet : l'abstraction

Rappel des concepts :

- **Encapsulation**

  L'encapsulation permet de cacher des détails d'implémentation aux clients de la classe (par exemple en déclarant des attributs privés).

- **Héritage**

  L'héritage permet de réutiliser une implémentation et d'étendre les responsabilités d'une classe (par exemple en héritant de la classe `Object`).

- **Polymorphisme**

  Le polymorphisme est une technique qui permet de déclencher des opérations différentes en réponse à un même message (par exemple en redéfinissant la méthode `toString`).

- **Abstraction**

  L'abstraction est un moyen de fournir une classe "incomplète", qui devra être complétée (on dit *implémentée* ou *réalisée*) par une à plusieurs autres classes.

  Une classe "incomplète" est dite **abstraite** . Si elle ne comporte que des opérations abstraites, on la qualifie alors d' **interface** parce qu'elle ne présente que les partie publiques.

  On ne peut pas directement créer des objets d'une classe abstraite ou d'une interface.

Nous introduisons également la notion de **collection** , déjà visitée en PHP.

7.2. Un zoo

Pour illustrer le concept d'abstraction, nous représenterons un zoo et les animaux qui l'habitent.

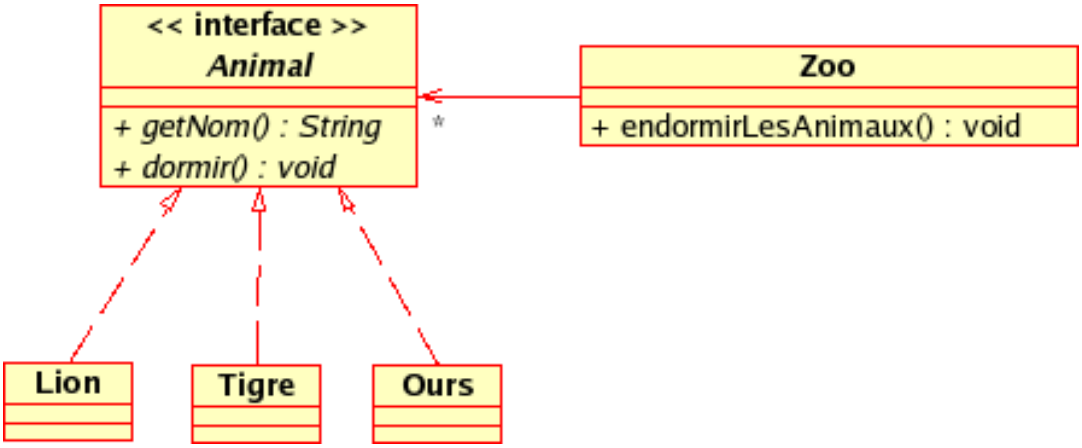


Figure 13. Diagramme de classes : un zoo est composé d'animaux

Comment représenter cette classification dans un langage objet comme Java ?

Déclarons une interface (une classe abstraite pure) :

```
public interface Animal {

    public String getNom();
    public void dormir();

 }
```

Puis les différentes classes d'animaux (Tigre, Lion et Ours) qui **implémentent** cette interface.

Classe `Lion` :

```
public class Lion implements Animal {
   //attribut
   private String nom;

   //constructeur
   public Lion (String nom) {
      this.nom = nom;
   }

   // méthodes
   public String getNom() {
     return this.nom;
   }
   public void dormir() {
     System.out.println(this.nom + " : Je dors sur le ventre.");
   }
 }
```

Classe `Tigre` :

```
public class Tigre implements Animal {
   //attribut
   private String nom;

   // constructeur
   public Tigre (String nom) {
```

```
      this.nom = nom;
   }
   //méthodes
   public String getNom() {
     return this.nom;
   }
   public void dormir() {
     System.out.println(this.nom + " : Je dors sur le dos.");
   }
 }
```

Classe `Ours` :

```
public class Ours implements Animal {
   //attribut
   private String nom;

   //constructeur
   public Ours (String nom) {
      this.nom = nom;
   }

   //méthodes
   public String getNom() {
     return this.nom;
   }
   public void dormir() {
     System.out.println(this.nom + " : Je dors dans un arbre.");
   }
 }
```

Déclarons le zoo : Il est constitué d'une collection d'objets (ce sera des objets de type `Animal` bien-sûr), son seul attribut, d'un constructeur qui crée des animaux (une façon de ne pas construire un zoo vide), et d'une méthode demandant à chacun des animaux du zoo de s'endormir.

```
import java.util.*;

public class Zoo {
 private List animaux; // collection d'animaux

 public Zoo(){
    // création de la collection
    this.animaux = new Vector();
      // List est une interface (classe abstraite)
      // on ne peut donc pas créer directement
      // un objet de ce type. Vector est
      // une classe qui réalise List. Ok.
      // A ce niveau, la liste est crée, mais vide.
 }

 /** endore tous les animaux
 *   en appelant leur méthode dormir().
 */
 public void endormirLesAnimaux() {
  // Les indices d'une liste commencent à zéro.
  // La méthode size() retourne le nombre d'éléments
  // de la collection.

  for (int i=0; i < animaux.size(); i++) {
    Animal animal = (Animal) animaux.get(i);
    animal.dormir();
  }

 }

 /** Permet d'ajouter un animal au zoo
```

```
 *  @param animal l'animal a ajouter
 */
 public void ajouteAnimal(Animal animal) {
   this.animaux.add(animal);
 }
}
```

Créons une application qui crée un zoo et endort ses animaux :

```
public class GestionZoo {
 //attribut
 private Zoo zoo;

 //constructeur
 public GestionZoo() {
   this.zoo = new Zoo();
 }

 //méthode
 public void run() {
    zoo.ajouteAnimal(new Lion("seigneur"));
    zoo.ajouteAnimal(new Tigre("gros chat"));
    zoo.ajouteAnimal(new Ours("balou"));
    zoo.ajouteAnimal(new Tigre("sherkan"));

    String choix = Console.lireClavier("0=Quitter, 1=Endormir : ");

    while (!choix.equals("0")) {
      if (choix.equals("1")) {
        zoo.endormirLesAnimaux();
      }
      choix = Console.lireClavier("0=Quitter, 1=Endormir : ");
    }//while
  }//run

  public static void main(String[] args) {
    GestionZoo app = new GestionZoo();
    app.run();
 }
}
```

Exemple d'éxecution

```
$ java GestionZoo
0=Quitter, 1=Endormir : 1
seigneur : Je dors sur le ventre.
gros chat : Je dors sur le dos.
balou : Je dors dans un arbre.
sherkan : Je dors sur le dos.
0=Quitter, 1=Endormir : 0
$
```

La classe `Zoo` **ne** dépend **pas** de classes d'implémentation d' `Animal` , elle **dépend d'une abstraction** (seulement de l'interface `Animal`).

Nous avons pu ainsi nous concentrer, dans la classe `Zoo` , essentiellement sur des traitements génériques, comme endormir les animaux. De nouvelles espèces d'animaux pourront être conçues *à posteriori* , tout en profitant de traitements génériques développés bien avant elles !

Les exercices suivants vont vous permettre de vérifier cela par vous même.

7.3. Exercices

Pour ajouter une nouvelle espèce, il suffit de concevoir une sous-classe de `Animal` et de définir sa méthode `dormir()` et `getNom()` , sans avoir besoin de modifier la méthode `endormirLesAnimaux()` .

Voici les fichiers : GestionZoo.java [16], Zoo.java [17], Animal.java [18], Lion.java [19], Tigre.java [20], Ours.java [21], Console.java [22].

1. Ajouter une nouvelle espèce, comme indiqué ci-dessus.

2. Instancier la nouvelle classe (créer un objet) dans la méthode `run` de `GestionZoo` , en donnant un nom à l'animal, et ajouter votre animal aux animaux du zoo.

3. Compiler et exécuter le programme.

4. Ajouter une méthode nommée `exprimeToi()` dans l'interface `Animal` que vous testez dans la classe `Zoo` .

   Cette méthode affichera le cri de l'animal, par exemple "miaou..", pour un gros minet. Vous modifierez les classes implémentant l'interface en conséquence.

5. Ajouter des traitements de gestion suivant : Affichage du nombre d'animaux du zoo, ajout d'un animal au zoo (l'utilisateur donne la classe et le nom de l'animal), suppression d'un animal du zoo (l'utilisateur donne son nom). Pour cela vous ajouterez les méthodes `nbAnimaux()` , `deleteAnimal(String nom)` dans la classe `Zoo` , et que vous testerez dans la méthode `run` de `GestionZoo` .

   Pour relever ce défi, vous vous appuierez sur l'objet `animaux` qui est de type `List` . **Vous devez aller voir ce qu'il est capable de faire dans la documentation de l'API java, à l'adresse suivante : `http://docig`** . Exemple :

[16] GestionZoo.java
[17] Zoo.java
[18] Animal.java
[19] Lion.java
[20] Tigre.java
[21] Ours.java
[22] Console.java

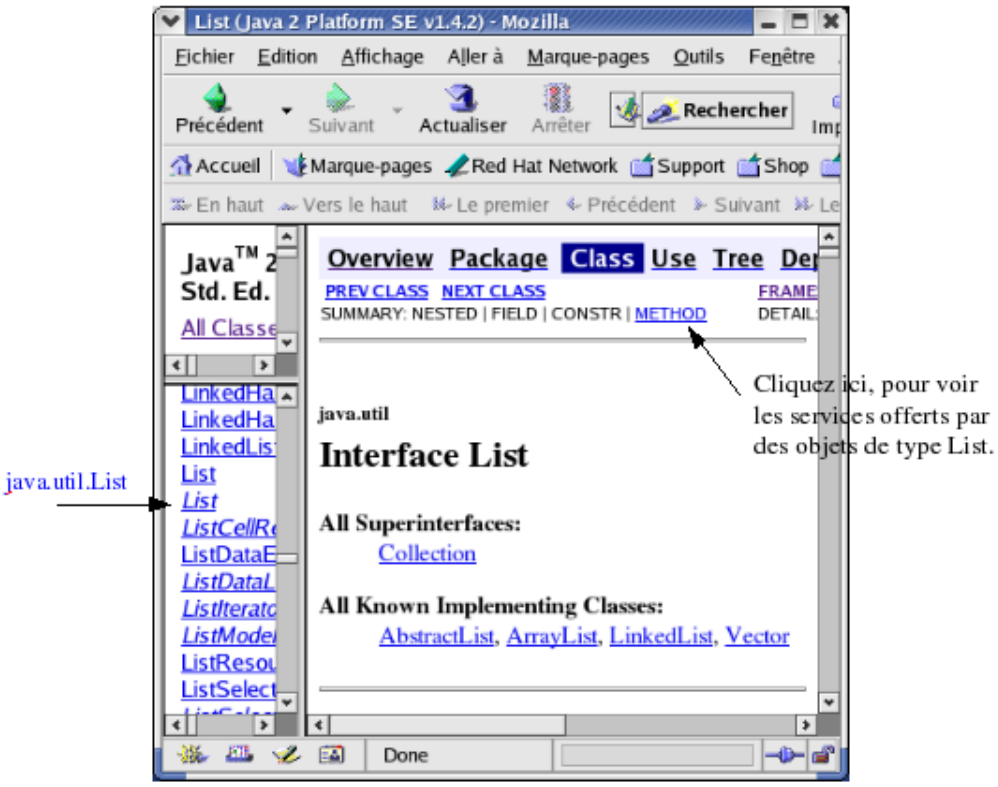


Figure 14. Extrait de la documentation de l'API

Bonne programmation !

8. Programmer des interfaces graphiques

8.1. Intro

Aujourd'hui, il est courant qu'une application de gestion fournisse une interface graphique à ses utilisateurs. Il existe deux types d'interface graphique utilisateur IGU, plus connues sous le sigle GUI pour *Graphic User Interface* : interface Web et interface de type window. L'une est orientée page et l'autre fenêtre. Ces deux types d'interface ont en commun les caractéristiques suivantes :

- Sensible aux événements externes et internes
- Construites avec une technologie objet
- Permettent plusieurs vecteurs de communication (image, son, texte, couleur, ...)

Avec Java le développeur dispose en standard de deux catégories d'outils (en dehors de HTML) pour construire des interfaces, ce sont **awt** et **swing** .

- **AWT** *Abstract Window Toolkit*

Avantages

- Gestion des événements efficace depuis Java 1.1.
- Plus rapide que Swing.
- Compatible avec les Navigateurs (Netscape et IE dans leur version >= 4 supportent les applets composées avec Java 1.1.)

Inconvénients

- API fortement lié aux ressources du SE hôte.
- Bibliothèque de composants moins riche que Swing.

Utilisation

- Applet (application Java embarquée dans un navigateur).
- Application autonome avec GUI simple.

- **SWING** Permet de construire des GUI de haut niveau. Basé sur AWT et sa gestion des événements.

Avantages

- Richesse des éléments d'interface.
- API d'accessibilité.
- Choix du "look and feel".
- Dessin 2D/3D.
- Gestion des événements sophistiquée.

Inconvénients

- Plus lent que l'AWT

Utilisation

- Application autonome, client/serveur.
- Application intranet et, dans une moindre mesure, internet (applet).

Il existe une alternative à AWT et Swing, c'est un projet initié par IBM, qui prend le meilleur de ces deux technologies : SWT (*Standard Widget Toolkit*) - voir http://www.eclipse.org/swt/ [23]). Nous nous concentrons ici sur **Swing** , le plus utilisé du moment (2003).

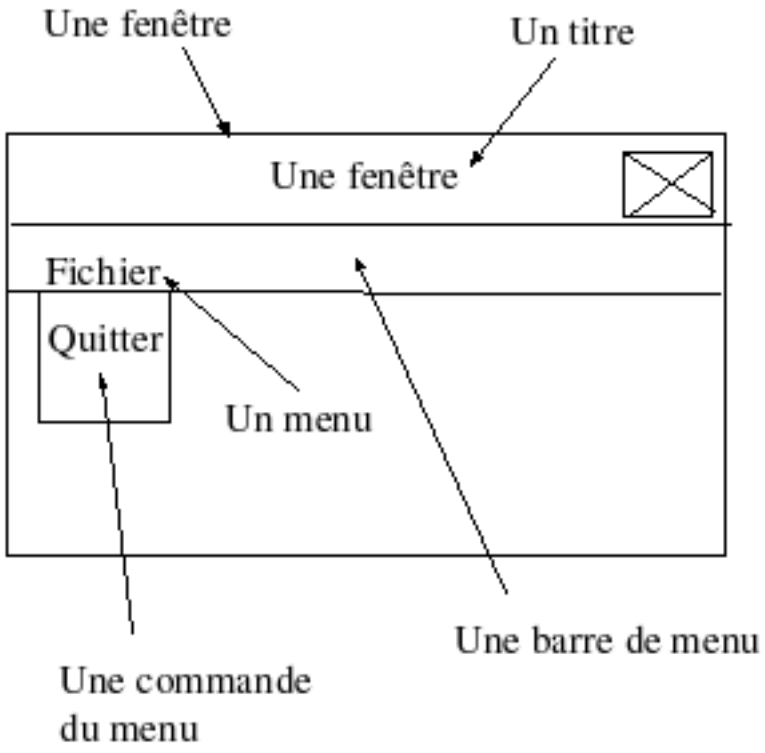
La construction d'une interface graphique utilisateur (GUI) nécessite environ 6 étapes

1. Concevoir une rapide maquette à la main, sur papier par exemple, afin de déterminer les principaux éléments de l'interface.
2. Concevoir une classe puis déclarer, créer et configurer les composants (couleur, texte...).
3. Placer les composants dans leur conteneur, puis dans le conteneur principal de la fenêtre.
4. Gérer le positionnement des composants dans leur conteneur (un panneau le plus souvent).
5. Associer aux composants générateurs d'événements un gestionnaire d'événements (le plus souvent la fenêtre elle-même).
6. Coder la logique événementielle de traitement.

Une application fenêtrée comporte généralement un menu. Nous vous proposons de construire une fenêtre d'application disposant d'un menu, que vous étendrez en exercice.

8.2. Étape 1 : La maquette papier

Nous choisissons de concevoir une fenêtre avec une barre de menus, munie d'une commande permettant de quitter l'application.



[23] http://www.eclipse.org/swt

Figure 15. Maquette de l'application exemple

8.3. Étape 2 : Conception de la fenêtre et de ses composants

`JFrame` est une classe de la bibliothèque `javax.swing` qui représente une fenêtre graphique.

L'idée est de **spécialiser** la classe de base `JFrame` en une classe qui contient les composants dont vous avez besoin. Par exemple, nous désirons concevoir une fenêtre ayant une barre de menus.

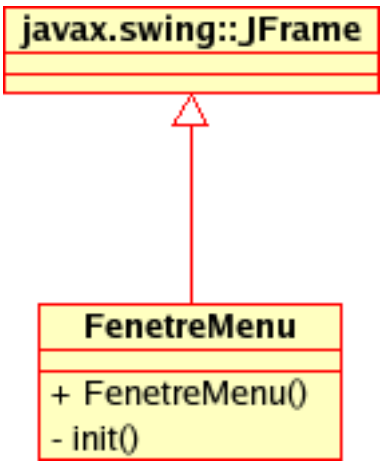


Figure 16. Réutilisation par héritage

Nous ajoutons un constructeur et une méthode d'initialisation. Voici une traduction partielle en Java :

```
import javax.swing.*;  public class FenetreMenu extends JFrame {
  // attributs
  ...
  // constructeur
  public FenetreMenu() {
    // appel un constructeur de son parent, en passant
    // une valeur de type chaîne de caractères
    super ("Fenetre avec une barre de menus");
    // effet : donne un titre à la fenêtre

    // permettre de quitter l'application lorsque l'utilisateur
    // clique sur la croix en haut à droite de la fenêtre.
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    this.init();
      // voir plus bas pour des explications concernant cette méthode.
  }
  ...
}
```

On remarquera le mot clé **`super`** , qui permet d'appeler un des constructeurs de la classe parent (rappel : un constructeur n'est pas une méthode).

Nous devons maintenant créer et configurer les composants.

Nous pouvons réaliser cela soit directement dans le constructeur, soit dans une méthode

d'initialisation spécialement conçue à cet effet (**`init`**) et appelée dans le corps du constructeur. Afin de ne pas alourdir le constructeur, nous créons la barre de menus et ses composants dans la méthode `init` . Extrait :

```
private void init(){
  //on a besoin de créer une barre de menus
  JMenuBar menuBar;
  // et un menu
  JMenu menuFichier;

  //création de la barre de menus
  menuBar = new JMenuBar();

  //construisons le premier menu
  menuFichier = new JMenu("Fichier");
  menuFichier.setMnemonic(KeyEvent.VK_F);

  // création de la commande "quitter" (un JMenuItem)
  JMenuItem mnItemQuitter = new JMenuItem("Quitter",
                    KeyEvent.VK_Q);

  mnItemQuitter.setActionCommand("Quitter");

  ...

  }
```

Nous avons choisi d'utiliser un constructeur `JMenuItem` permettant de spécifier une touche d'accès rapide (un *mnemonic*) spécifiée en second paramètre. Vous noterez que la valeur du paramètre est une **constante static** de la classe `KeyEvent` , `VK` voulant dire *Virtual Key* .

Nous spécifions le **nom de l' "action" auquel le composant sera lié** . C'est une simple information (`String`) que le composant transmet à ses "écouteurs" lorsqu'il est activé (ce point est détaillé plus bas dans ce document).

Remarque : L'aide en ligne est la principale ressource permettant de connaître les possibilités de configuration offertes par un composant.

8.4. Étape 3 : Placer les composants dans un conteneur

La barre de menus **contient** des menus, et les menus **contiennent** des commandes (*menu item*).

Ces relations sont sous la responsabilité du programmeur. Comme on peut s'y attendre, les conteneurs disposent d'une méthode nommée **`add`** permettant l'ajout de composants.

```
private void init() {
   ...
   // le menu Fichier contient la commande Quitter
   menuFichier. add (mnItemQuitter);

   // on peut placer une barre de séparation pour
   // séparer des groupes logiques de commandes.
   // menu.addSeparator();

   // la barre de menus contient le menu Fichier
   menuBar. add (menuFichier);
   // plus un autre menu bidon, pour l'exemple
   menuBar. add (new JMenu("Un autre menu"));

   ...
 }
```

8.5. Étape 4 : Gérer la position du composant dans la vue

Deux façons de faire :

1. Gérer soi-même la position en x, y du composant.
2. Sous-traiter le positionnement du composant par un *gestionnaire de positionnement* (**layout**).

   Java propose en standard quelques gestionnaires, qui seront étudiés prochainement.

En ce qui concerne la barre de menu, il existe exceptionnellement une méthode, nommée `setJMenuBar` , dédiée à son placement dans la fenêtre (une barre de menus est traditionnellement située sous la barre de titre) :

```
private void init() {
   ...
  // fournir à la fenêtre une barre de menus
  this. setJMenuBar (menuBar);
  ...

  // donnons une largeur et une hauteur à la fenêtre
  this.setSize(300,200);
}
```

Ce qui donne :

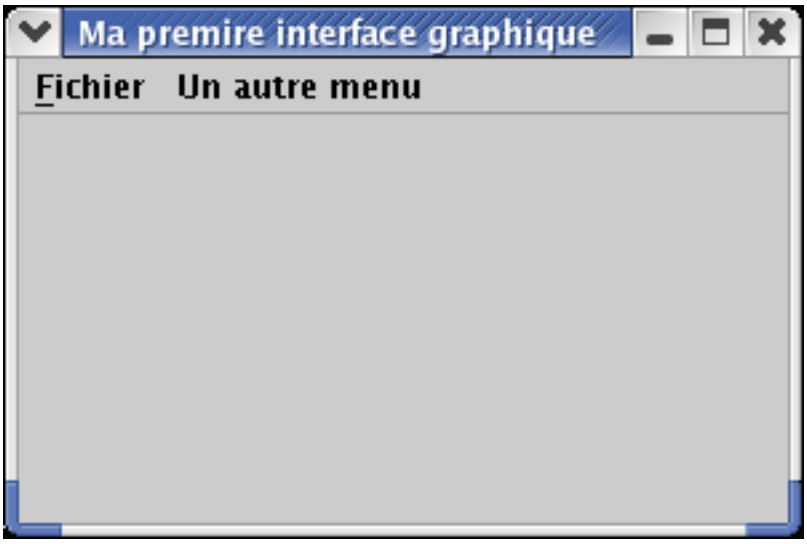


Figure 17. FenetreMenu à l'exécution

Une fois notre composant créé (avec **new**), placé et positionné dans la vue, il ne nous reste qu'à nous occuper de la gestion de ses événements.

8.6. Étape 5 : Associer un gestionnaire d'événement

Bon nombre de composants sont générateurs d'événements. Par exemple, lorsque l'utilisateur clique sur un bouton, celui-ci est capable de prévenir tout autre objet qui l'écoute. Pour qu'un objet puisse

être prévenu de l'événement, il doit **s'abonner au composant générateur de l'événement** .

La fenêtre s'abonne à la commande du menu de cette façon :

```
mnItemQuitter. addActionListener (this);
```

this fait référence à la fenêtre en cours (celle qui sera en exécution). Par cette instruction, on **abonne** la fenêtre en tant qu' **écouteur** (*listener*) des événements générés par `mnItemQuitter` .

Pour que le `JMenuItem` accepte la fenêtre en tant que gestionnaire de ses événements, la fenêtre doit être **une sorte de** *ActionListener* (un écouteur d'événement). Pour cela la fenêtre devra **implémenter** l' **interface** `ActionListener` .

```
public class FenetreMenu
  extends JFrame implements ActionListener {
    ...
}
```

... et donc **donner corps à la méthode** :

```
public void actionPerformed(ActionEvent evt)
```

... seule opération déclarée par l'interface `ActionListener` . C'est l'objet de l'ultime étape.

8.7. Étape 6 : Coder la logique de traitement

C'est dans cette dernière étape que le développeur justifie pleinement son rôle...

Il doit programmer le comportement de l'application lorsqu'un événement survient.

Par exemple, ici l'utilisateur souhaite mettre fin à l'application.

```
...
public void actionPerformed(ActionEvent evt) {
  String action = evt.getActionCommand();
  if (action.equals("Quitter")) {
    System.exit(0);
  }
}
...
```

La méthode `actionPerformed` est automatiquement appelée lorsque l'utilisateur sélectionne la commande Quitter.

Puisqu'une fenêtre est généralement abonnée à plusieurs composants, la méthode `actionPerformed` , détermine l'action (méthode) devant être déclenchée. Pour cela, elle interroge la "command action", détenue par le paramètre.

Si la *command action* vaut "Quitter", alors, nous savons que `mnItemQuitter` est à l'origine de l'événement et nous mettrons fin à l'application.

8.8. L'exemple complet

```
mport javax.swing.*;
import java.awt.event.*;

public class FenetreMenu extends JFrame implements ActionListener{
```

```
   // une constante (mot clé final)
   // c'est un moyen très pratique d'associer un écouteur d'événement
   // à un générateur d'événement.
   static final String ACTION_QUITTER = "Quitter";

   // constructeur
   public FenetreMenu() {
     // appel un constructeur de son parent
     super("Ma première interface graphique");
        // effet : donne un titre à la fenêtre

     // l'application s'arrête lorsque la fenêtre est fermée.
     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

     // initialisation de la fenêtre
     init();
   }

 private void init(){
   //on a besoin de créer une barre de menus
   JMenuBar menuBar;
   // et un menu
   JMenu menuFichier;

   //création dela barre de menus
    menuBar = new JMenuBar();
   //construisons le premier menu
   menuFichier = new JMenu("Fichier");
   menuFichier.setMnemonic(KeyEvent.VK_F);
   menuFichier.getAccessibleContext().setAccessibleDescription(
        "Menu permettant d'accéder à une commande pour quitter");

   //création de la commande "quitter"
   JMenuItem mnItemQuitter = new JMenuItem(ACTION_QUITTER,
                         KeyEvent.VK_Q);
   mnItemQuitter.getAccessibleContext().setAccessibleDescription(
        "Quitter le programme");

   // le menu Fichier contient la commande Quitter
   menuFichier.add(mnItemQuitter);
   //menu.addSeparator();
   // la barre de menus contient le menu Fichier
   menuBar.add(menuFichier);
   menuBar.add(new JMenu("Un autre menu"));
   // on l'ajoute à la fenêtre
   setJMenuBar(menuBar);

   // la fenêtre est à l'écoute d'une action sur ce menu
   mnItemQuitter.addActionListener(this);

   setSize(300,200);
  }

  public void actionPerformed(ActionEvent evt) {
     String action = evt.getActionCommand();

     if (action.equals(ACTION_QUITTER)) {
       System.exit(0);
     }
  }

  static public void main(String[] args) {
     JFrame f = new FenetreMenu();
     f.setVisible(true);
  }

}// FentreMenu
```

8.9. Résumé

Nous avons vu, au travers d'un exemple, les **6 étapes nécessaires à la conception d'une interface graphique en Java** .

1. Analyser et concevoir une maquette "papier".
2. Réutiliser par héritage, puis créer et configurer les composants.
3. Placer les composants dans un conteneur.
4. Gérer la position des composants dans le conteneur.
5. Lier des gestionnaires d'événements aux composants retenus.
6. Coder la logique en réponse aux événements.

Composants Swing et Awt

Les composants `swing` sont reconnaissables par le fait que leur nom commence par un J, comme par exemple `JButton` (pour des boutons), `JTextField` (zone de texte pour la saisie), `JLabel` (simple zone de texte), `JPanel` (des panneaux), etc.

En règle générale, vous ne devriez **jamais mélanger** dans une même interface, des composants AWT et SWING.

Action Command

Nous vous conseillons d'utiliser des CONSTANTES comme valeur des *action command* , comme par exemple ACTION_QUITTER.

```
static final String ACTION_QUITTER = "Quitter";
```

Par convention, les constantes sont spécifiées en MAJUSCULE, et sont généralement déclarées *static* . Il est en effet inutile de dupliquer une constante pour chaque objet. Rappelons qu'une propriété *static* est liée à la classe elle-même et non à chacun des objets de la classe en particulier. Du coup, tous les objets d'une même classe partagent la même propriété (si celle-ci est déclarée static). C'est pourquoi, cette dernière est très souvent une constante (*final*).

Vous trouverez chez Sun un tutoriel dédié à l'utilisation des menus. Vous apprendrez comment placer des cases à cocher, des sous-menus, des images dans un menu, c'est ici http://java.sun.com/docs/books/tutorial/uiswing/components/menu.html [24].

8.10. Exercices

1. Ajouter à la barre de menus de la fenêtre, un menu nommé `Configuration` .
2. Ajouter à ce nouveau menu, les commandes `Modifier la hauteur` , `Modifier la largeur` , `Modifier la position en x` , `Modifier la position en y` .
3. Coder la logique de traitement pour chacune des commandes. Une `JFrame` a un attribut **privé** nommé `width` (largeur) et `height` (auteur). Ces propriétés sont interrogeables par la méthode `getSize` et modifiable par `setSize` . Attention, `getSize` retourne un objet de type `Dimension` qui dispose de deux attributs publics : `width` et `height` . Exemple

[24] http://java.sun.com/docs/books/tutorial/uiswing/components/menu.html

d'utilisation :

```
Dimension dim = this.getSize();
String msg = "La largeur d la fenêtre est : " + dim.width ;
javax.swing.JOptionPane.showMessageDialog(null, msg);
```

Pour modifier la position en X,Y d'un composant, vous pouvez utiliser la méthode `setLocation` , et accessoirement la méthode `getLocation` . Exemple d'aide de l'API :

```
// méthode de la classe java.awt.Component
public Point getLocation()
  // Returns:
  //  an instance of Point representing the top-left corner
  //  of the component's bounds in the coordinate space
  //  of the component's parent.
```

Remarque 1 : `Point` est une classe qui a deux attributs publics X et Y, de type `int` .

Remarque 2 : le composant "parent de la fenêtre" (celui qui contient la fenêtre) est dans notre cas l'écran.

On vous demande de concevoir **une méthode privée par commande (`JMenuItem`)** , ainsi le gestionnaire d'événements ne fera que sélectionner la bonne méthode en fonction de la source de l'événement. Exemple :

```
public void actionPerformed(ActionEvent evt) {
   String action = evt.getActionCommand();
   if (action.equals(ACTION_QUITTER) {
      System.exit(0);
   }
   else if (action.equals(ACTION_MODIF_X) {
     // une méthode à concevoir dans cette classe
     this.setPositionX();
   }
   // etc.
}
```

4. Présenter à l'utilisateur, dans la boîte de dialogue, la valeur actuelle qu'il s'apprête à modifier.
5. Ajouter une commande permettant à l'utilisateur de changer le titre de la fenêtre.
6. Ajouter une commande permettant de placer automatiquement la fenêtre au centre de l'écran.

Astuce

Voici comment obtenir la taille de l'écran :

```
Dimension screenSize = getToolkit().getScreenSize();
```

La taille de la fenêtre s'obtient par :

```
Dimension paneSize = getSize();
```

Il ne vous reste plus qu'à chercher dans l'aide en ligne comment utiliser un objet de type `Dimension` .

Vous remarquerez que ses champs (attributs) `width` et `height` sont publics. C'est un des rares contre-exemple du principe *d'encapsulation* .

9. Gérer les événements de la souris et le positionnement des composants

9.1. Intro

Un événement souris (ou touch pad, stick...) peut intervenir à "n'importe quel moment" dans une application.

Les composants traditionnels des interfaces graphiques (boutons, zone de liste, case à cocher, etc.) sont "naturellement" sensibles aux événements. Par exemple, tous les composants Java qui **héritent** , plus ou moins directement, de la classe `java.awt.Component` sont sensibles et **transmetteurs d'événements souris** .

Le positionnement des composants dans leur conteneur peut être géré par un composant dédié à cette tâche : `LayoutManager` .

Nous présentons en premier lieu les interfaces pour la gestion des événements souris.

9.2. MouseListener, MouseMotionListener

Java distingue deux types d'événements souris, représentés par **deux interfaces** : `ActionListener` et `MouseMotionListener` :

MouseListener

- `public void mouseClicked(MouseEvent e)`

  Condition de déclenchement : **Clic de souris (pressed and released) sur un composant**

- `public void mouseEntered(MouseEvent e)`

  Condition de déclenchement : **Lorsque la souris entre dans un composant**

- `public void mouseExited(MouseEvent e)`

  Condition de déclenchement : **Lorsque la souris quitte un composant**

- `public void mousePressed(MouseEvent e)`

  Condition de déclenchement : **Lorsqu'un bouton a été pressé sur un composant**

- `public void mouseReleased(MouseEvent e)`

  Condition de déclenchement : **Lorsqu'un bouton a été relâché sur un composant**
  .

MouseMotionListener

- `public void mouseDragged(MouseEvent e)`

  Condition de déclenchement : **Lorsqu'un bouton a été pressé et glissé sur un composant**

- `public void mouseMoved(MouseEvent e)`

  Condition de déclenchement : **Lorsque le pointeur de la souris a été déplacé au-dessus d'un composant, mais sans qu'aucun bouton ne soit pressé.**

Lorsqu'un composant, par exemple une interface graphique `JFrame` , souhaite être averti d'un événement souris en provenance d'un de ses composants, par exemple un `JPanel` , la `JFrame` doit **s'abonner** à ce dernier. Pour cela notre `JFrame` devra **implémenter** une des deux (ou les deux) interfaces. Exemple :

```
public class GUI extends JFrame, implements MouseListener {
 ...

 public void mouseClicked(MouseEvent e){}
   // Invoked when the mouse button has
   // been clicked (pressed and released) on a component.
 public void mouseEntered(MouseEvent e){}
   // Invoked when the mouse enters a component.
 public void mouseExited(MouseEvent e){}
   // Invoked when the mouse exits a component.
 public void mousePressed(MouseEvent e){}
   // Invoked when a mouse button has been pressed on a component.
 public void mouseReleased(MouseEvent e){}
   // Invoked when a mouse button has been released on a component.

 ...
}
```

9.3. Gestionnaire de positionnement

Un "gestionnaire de positionnement" (*Layout management*) est un système qui détermine la taille et la position de composants. Par défaut, chaque conteneur a un layout manager - un objet qui gère la taille et la position des composants à l'intérieur du conteneur. Un composant peut tenter de gérer sa taille et sa position, mais au final, c'est le layout manager qui aura le dernier mot.

Les gestionnaires de positionnement les plus courants sont : `BorderLayout` , `FlowLayout` , `GridLayout` . Il en existe d'autres : `BoxLayout` , `GridBagLayout` ... à chaque nouvel version du jdk.

- **BorderLayout** : Permet de positionner des composants selon un positionnement repéré par des points cardinaux (nord, est, sud, ouest) plus le centre.

  Remarque : `BorderLayout` est le layout par défaut de chaque content pane (conteneur principal des objets de type `JFrame` , `JApplet` et `JDialog`).

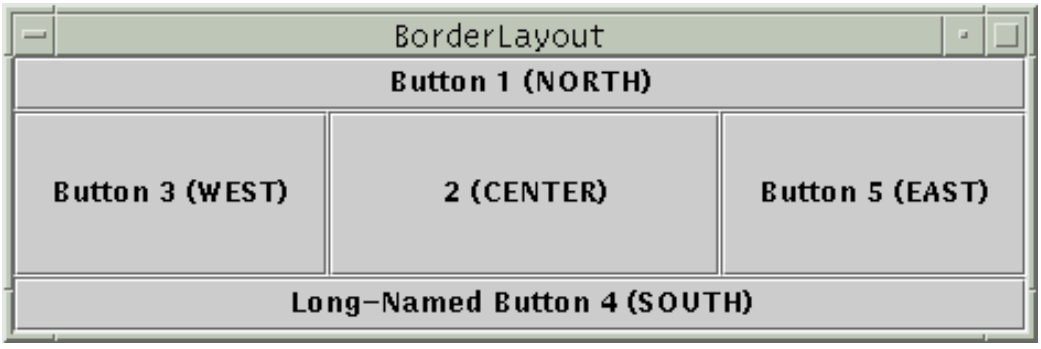


Figure 18. Exemple d'utilisation d'un BorderLayout

- **FlowLayout** : Place les composants de la **gauche vers la droite** , et au besoin, continue sur une nouvelle ligne en dessous.

  Remarque : C'est le gestionnaire par **défaut** des JPanel.

Figure 19. Exemple d'utilisation d'un BorderLayout

- **GridLayout** : **Taille** les composants à une *même* taille et les **range** selon une grille exprimée en nombre de lignes (*rows*) et de colonnes (**columns**).

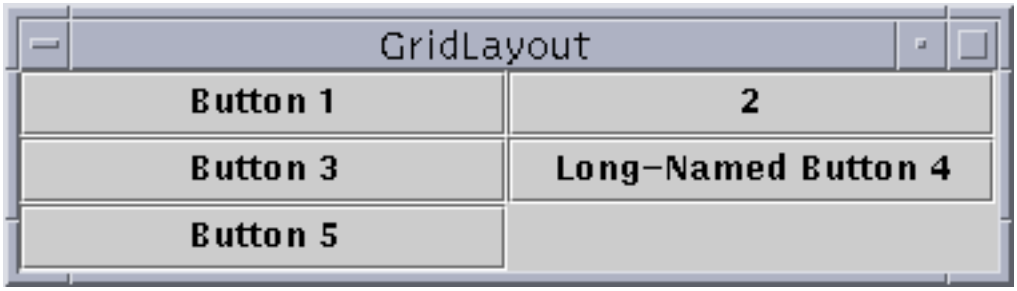
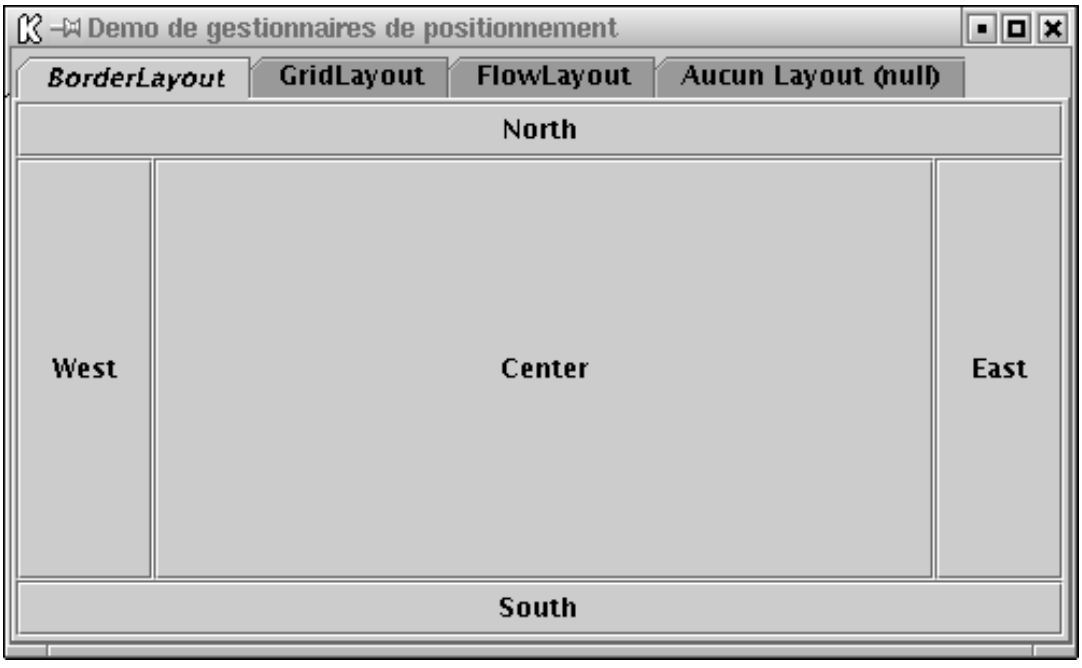


Figure 20. Exemple d'utilisation d'un GridLayout

9.4. Programme exemple

Référence : Java Tutorial [25]

Le programme ci-dessous a deux objectifs : Le premier est de vous montrer par l'exemple comment utiliser plusieurs *layout manager* , le second est de vous présenter, également par l'exemple, le composant swing `JTabbedPane` qui permet de gérer des groupes de composants dans des onglets.



[25] http://java.sun.com/docs/books/tutorial/uiswing/layout/index.html

Figure 21. Exemple d'utilisation d'un GridLayout

Chacun des onglets se compose d'un panel (un `JPanel`) servant de conteneur à d'autres composants comme des `JButton` ou `JLabel` . Chaque onglet dispose de son propre `LayoutManager` .

La construction de chaque onglet est sous-traitée à des méthodes privées (`makePanelWithXXXX`), dans le but d'en faciliter la lecture et la maintenance.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class TestLayout extends JFrame {
 public TestLayout(String titre) {
  // appel du constructeur du parent (un JFrame)
  super(titre);
  // pour terminer l'application lorsque la fenêtre se ferme
  this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  init();
}
private void init() {
  ImageIcon icon = null;
  JTabbedPane tabbedPane = new JTabbedPane();

  Component panel = makePanelWithBorderLayout();
    // appel une méthode qui retourne un panel après l'avoir
    // construit (avec new)

  tabbedPane.addTab("<html><i>BorderLayout</i></html>", icon,
                    panel, "info bulle");
    //  on remarquera le "code" html (obligatoirement bien formé)
    //  utilisable pour formater le texte.
  tabbedPane.setSelectedIndex(0);
    // définit l'onglet actif au démarrage de l'application

  panel = makePanelWithGridLayout();
  tabbedPane.addTab("GridLayout", icon, panel, "avec des JButton");

  panel = makePanelWithFlowLayout();
  tabbedPane.addTab("FlowLayout", icon, panel, "une info bulle");

  panel = makePanelWithNoLayout();
  tabbedPane.addTab("Aucun Layout (null)", icon,
                    panel, "avec JLabel et JButton");

  //Ajoute TabbedPane au panel conteneur.
  getContentPane().setLayout(new GridLayout(1, 1));
  getContentPane().add(tabbedPane);
} // init
...
}
```

La méthode `init` crée le composant `JTabbedPane` et appelle successivement des méthodes privées (définies plus bas dans le corps de la classe) afin d'ajouter des onglets.

Les dernières instructions du constructeur, allouent un LayoutManager au conteneur principal de la fenêtre, puis placent dans ce dernier le composant gestionnaire d'onglets.

Ci-dessous, le code des méthodes privées :

```
private JPanel makePanelWithBorderLayout() {
```

```
  JPanel panelBorderLayout = new JPanel(false);
  panelBorderLayout.setLayout(new BorderLayout());
  panelBorderLayout.add("North",new JButton("North"));
  panelBorderLayout.add("South",new JButton("South"));
  panelBorderLayout.add("East",new JButton("East"));
  panelBorderLayout.add("West",new JButton("West"));
  panelBorderLayout.add("Center",new JButton("Center"));
  return panelBorderLayout;
}
private JPanel makePanelWithFlowLayout() {
  JPanel panelFlowLayout = new JPanel(false);
  panelFlowLayout.setLayout(new FlowLayout());
  for (int i=0;i<10;++i) {
    panelFlowLayout.add(new JButton(String.valueOf(i)));
  }
  return panelFlowLayout;
}
private JPanel makePanelWithGridLayout() {
  JPanel panelGridLayout = new JPanel(false);
  panelGridLayout.setLayout(new GridLayout(4,3));
  for (int i=0;i<2;++i) {
     panelGridLayout.add(new JButton(String.valueOf(i)));
  }
  return panelGridLayout;
}
```

A chaque fois l'algorithme est le même :

1. Création d'un conteneur (généralement un JPanel)
2. Affectation d'un LayoutManager à ce conteneur
3. Ajout de composants à l'intérieur du conteneur, le LayoutManager se chargeant de leur positionnement et parfois de leur taille.

Notons le cas à part où **aucun** LayoutManager n'est souhaité. Dans ce cas on affecte comme valeur de LayoutManager la valeur *null* . On doit alors positionner chacun des composants via, par exemple, leur méthode `setLocation` ou `setBounds` .

```
private JPanel makePanelWithNoLayout() {
  JPanel panelNoLayout = new JPanel(false);
  panelNoLayout.setLayout(null);
  JLabel label1 = new JLabel("Label en positon (10,10)");
  JLabel label2 = new JLabel("Label en positon (70,70)");
  JButton button1 = new JButton("Bouton en positon (130,130)");
  JButton button2 = new JButton("Bouton en positon (190,190)");
  label1.setBounds(10,10,200,30);
  label2.setBounds(70,70,200,30);
  button1.setBounds(130,130,250,30);
  button2.setBounds(190,190,250,60);
  panelNoLayout.add(label1);
  panelNoLayout.add(label2);
  panelNoLayout.add(button1);
  panelNoLayout.add(button2);

  return panelNoLayout;
}
```

Ce qui donne :

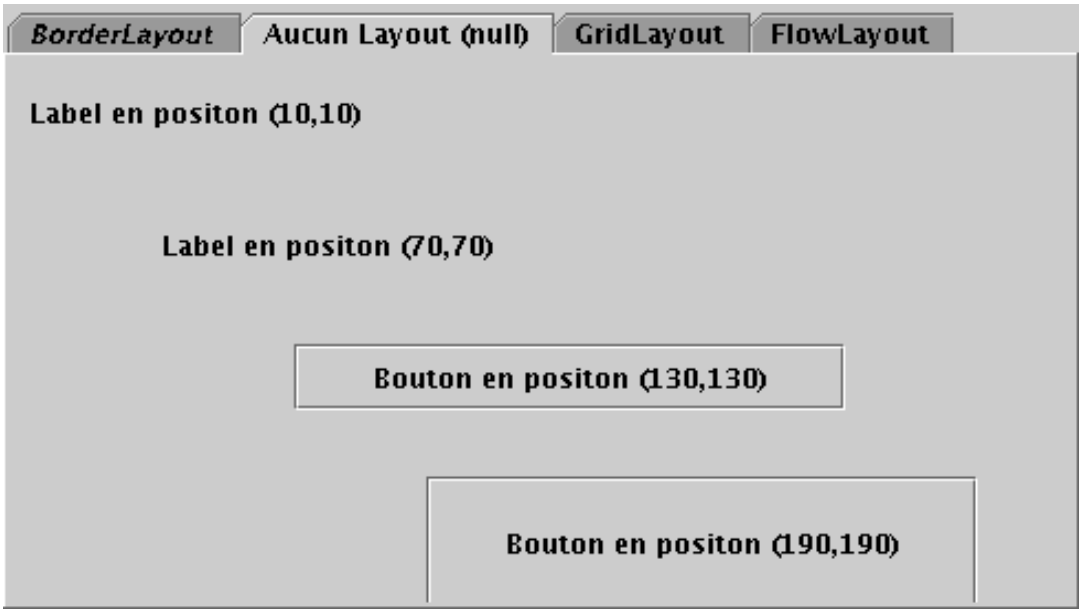


Figure 22. Exemple sans Layout

Enfin, un main classique, qui centre la fenêtre à l'écran après l'avoir dimensionnée.

```
public static void main(String[] args) {
   JFrame frame =
      new TestLayout("Demo de gestionnaires de positionnement");
   Dimension dimScreen = frame.getToolkit().getScreenSize();
   frame.setSize(500, 300);
   frame.setLocation((dimScreen.width-frame.getWidth())/2,
                     (dimScreen.height-frame.getHeight())/2);
   frame.setVisible(true);
 }
```

9.5. Exercice

1. Reconstruire l'application

2. Sans rien changer à l'ordre de création d'ajout des onglets (Border, Grid, Flow, No), mais peut-être à la façon de les ajouter, faire en sorte que l'onglet sans layout soient placés en seconde position. Vous chercherez dans l'API, une autre méthode d'insertion d'un onglet dans un `JTabbedPane` que celle utilisée dans l'exemple (qui ne permet pas de choisir la position du nouvel onglet parmi les autres onglets).

3. L'onglet qui présente le `GridLayout` , est composé d'un panel qui contient un certain nombre de boutons, tous "numérotés" de 0 à 11.

   Modifier la construction de ce panel de sorte qu'il contienne 24 boutons marqués d'une **lettre** , de 'A' à 'Z', sur 4 lignes :

Figure 23. Exemple avec boutons de 'A' à 'Z'

Astuce : les caractères peuvent être représentés par un type primitif : `char` . Dans ce cas, n'oubliez pas que ce n'est qu'un nombre entier particulier (qui peut faire office de variable de boucle).

4. On souhaite que lorsque l'utilisateur clique sur un des boutons marqué d'une lettre, le programme affiche une boîte de dialogue contenant la valeur de la lettre du bouton :

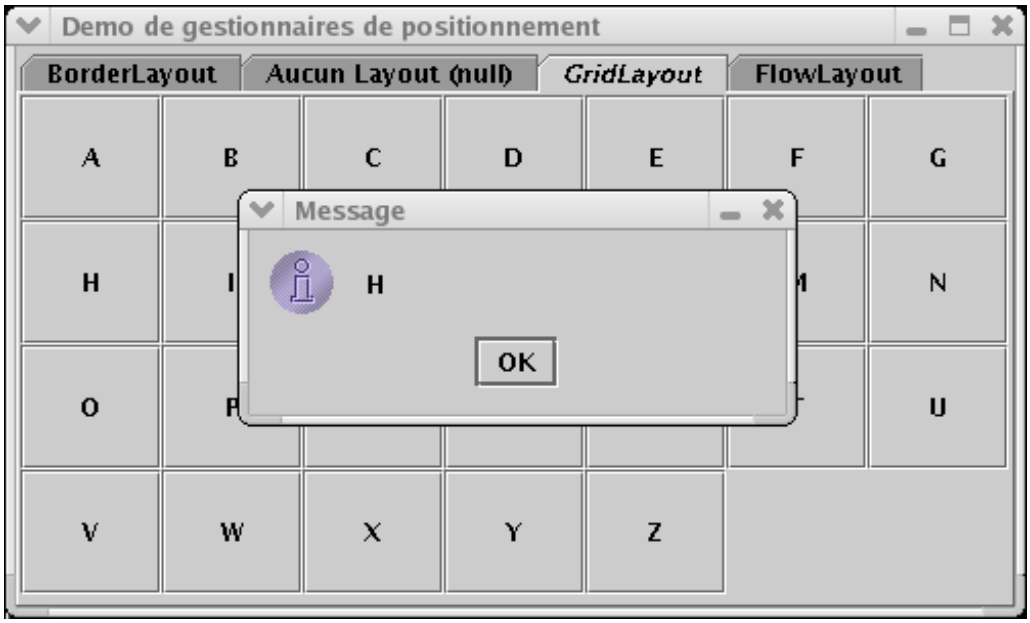


Figure 24. Exemple d'interception d'un clic de souris

Aide 1 : la fenêtre devra implémenter l'interface `MouseListener` , et s'abonner à **chacun** des boutons lettrés.

Aide 2 : la méthode événementielle `mouseClicked` (voir explications plus haut), devra interroger l'objet à l'origine de l'événement :

```
public void mouseClicked(MouseEvent e){
   //obtenir l'objet émetteur
   Object o = e.getSource();
   if (o instanceof JButton) {
     //si c'est un JButton alors on peut y aller
     JButton b = (JButton) e.getSource();
     // puis faire quelque chose, selon les
     // spécifications énnoncées ci-dessus
     ...
   }
 }
```

5. On veut placer une zone d'information au bas de la fenêtre. Cette zone devra fournir :

   - Le nom de l'onglet actif.
   - L'index de l'onglet actif.

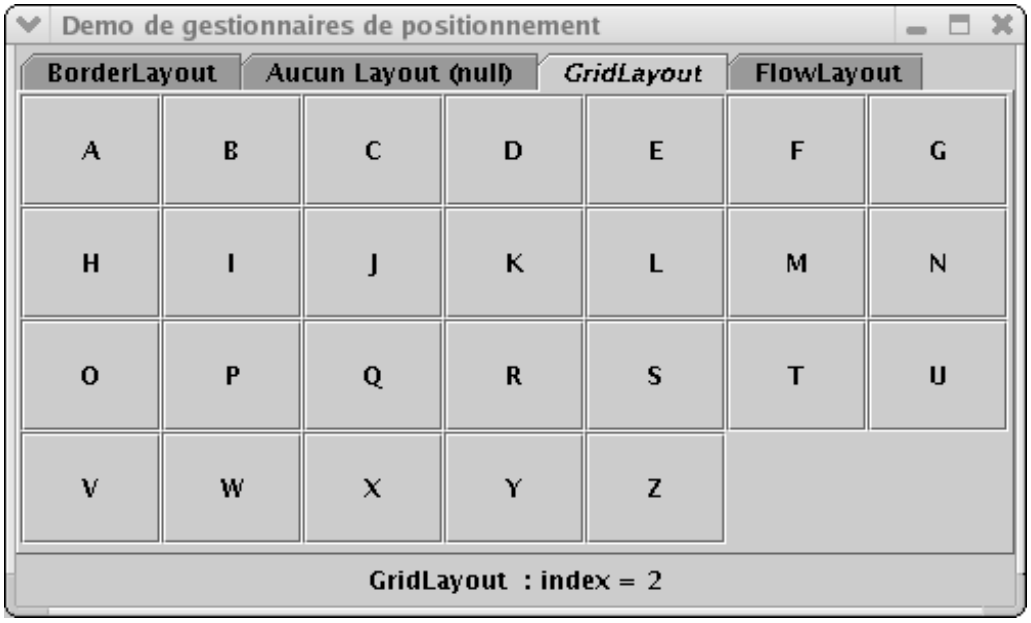


Figure 25. Exemple de barre de statut

Indication : fournir un BorderLayout au content pane de la fenêtre, puis placer au centre le composant à onglets, et au sud un panel contenant un label. C'est la valeur de ce label qui sera modifiée en cours d'exécution.

6. (Pus difficile) On souhaite que **seul le titre de l'onglet actif soit en italique** . Pour cela, le formatage du titre (avec des instructions HTML) doit se faire à l' **exécution** et non à la **conception** comme actuellement.

   Pour intercepter l'événement déclenché lors d'un changement d'onglet, vous devez implémenter un **ChangeListener** qui expose une seule méthode : `void stateChanged(ChangeEvent e)` .

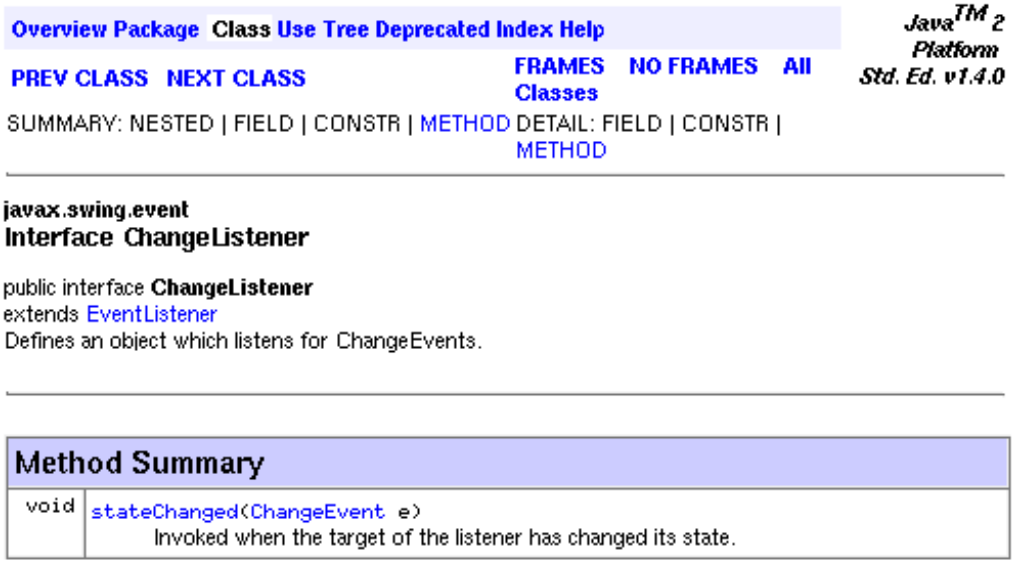


Figure 26. Exemple d'aide de l'API

Ce que nous vous conseillons de faire :

a. votre JFrame implémente l'interface ChangeListener (ajouter un import du package adéquate)

b. la méthode `ChangeListener` obtient l'index de l'onglet sélectionné (voir quelle méthode utiliser pour cela dans l'aide en ligne)

c. puis obtient le titre de cet onglet (via une méthode existante) et l'affiche simplement.

A ce stade, vous savez récupérer le titre de l'onglet sélectionné. Il ne vous reste plus qu'à modifier le titre de l'onglet précédemment sélectionné afin de supprimer les ordres de formatage HTML ayant pour effet de le mettre en italique, et de réaliser l'opération inverse sur le titre de l'onglet courant.

Vous pouvez introduire un nouvel attribut, par exemple `int index_precedent` , afin de ramener le titre du dernier index sélectionné à son format original. Pour cela, comme d'habitude, il y a plusieurs façons de faire. On vous propose ici deux idées, mais vous pouvez très bien en choisir une autre :

Idée A =>Un titre (title) est de type `String` , aidez-vous de l'aide en ligne pour savoir comment rechercher et extraire une sous-chaîne d'une chaîne.

Idée B =>Utiliser d'un tableau de titres, exemple avec 2 titres :

```
// création d'un tableau de chaînes de caractères
final String []  TITRES = { "BorderLayout", "GridLayout"};

...
```

Avec ce tableau, le code suivant :

```
tabbedPane.addTab("GridLayout", icon, panel, "avec des JButton");
```

est remplacé par :

```
tabbedPane.addTab(TITRES[1], icon, panel, "exemple info bulle");
```

On vous laisse deviner la suite...

Bonne programmation !

10. Packager une application

Package , terme, parfois traduit par paquetage, correspondant à un nom d'unité organisationnelle, une sorte de boîte dans laquelle sont localisées les classes.

Le nom du package doit être **unique** , au niveau planétaire, afin qu'il ne rentre pas en conflit avec un autre package défini par une autre organisation. C'est pourquoi le nom d'un site est souvent utilisé ici, noté à l'envers de son usage, suivi d'un nom de paquetage (convention).

10.1. Déclaration du package

Exemple de **définition d'un package** :

```
/**
 * Projet     : Cyber Promeneur
 * Initiateur : Olivier Capuozzo
 *              Lycee Leonard de Vinci
 *              77000 Melun
 * Modifié par
 * l'étudiante : Destierdt Audrey
 * Date       : 06 janvier 2004
 * Public     : BTS IG 1ere annee
 * @version 1.01, 13 dec. 2003
*/
package org.vincimelun.cyber;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Date;

abstract public class CyberPromeneur extends JPanel
implements ActionListener
{
...
}
```

L'instruction de déclaration de package se place **avant** les imports.

10.2. Compilation des sources du package

Avant compilation

```
$ll
total 36
-rw-rw-r--    1 kpu      kpu    4053 jan  8 12:34 Bonhomme.java
-rw-rw-r--    1 kpu      kpu    9436 jan  8 12:34 CyberPromeneur.java
-rw-rw-r--    1 kpu      kpu    2677 jan  8 12:34 GrandControleur.java
-rw-rw-r--    1 kpu      kpu    4529 jan  8 12:34 Scene.java
```

```
-rw-rw-r--    1 kpu      kpu    2219 jan  8 12:34 SceneLayout.java
-rw-rw-r--    1 kpu      kpu    2197 jan  8 12:34 Truc.java
$
```

On prendra soin de spécifier le chemin des classes, avec l'option `-d <chemin de destination>`

```
$ javac *.java -d .
$ ll
total 40
-rw-rw-r--    1 kpu      kpu    4053 jan  8 12:34 Bonhomme.java
-rw-rw-r--    1 kpu      kpu    9436 jan  8 12:34 CyberPromeneur.java
-rw-rw-r--    1 kpu      kpu    2677 jan  8 12:34 GrandControleur.java
drwxrwxr-x    3 kpu      kpu    4096 jan  8 18:00 org
-rw-rw-r--    1 kpu      kpu    4529 jan  8 12:34 Scene.java
-rw-rw-r--    1 kpu      kpu    2219 jan  8 12:34 SceneLayout.java
-rw-rw-r--    1 kpu      kpu    2197 jan  8 12:34 Truc.java
$
```

Soit, l'arborescence suivante :

```
[kpu@kpu tp1]$ -|- *.java
                |
                |-org--|
                       |
                       |-vincimelun--|
                                     |
                                     |-cyber--|
                                              |- *.class
```

10.3. Test

Il existe deux types d'erreur : erreur à la compilation et erreur à l'exécution. Dans les deux cas, le code source est retouché, puis recompilé. Ces deux étapes se réalisent donc en boucle, puis suivent les activités de tests fonctionnels qui vérifient le bon fonctionnement des parties et du tout.

Exemple d'exécution :

```
$ java org.vincimelun.cyber.Scene
```

10.4. Construction du fichier de déploiement

Une fois la mise au point réussie, lorsque l'application parait fonctionner correctement, on passe alors à la phase de déploiement : L'application est livrée au client, déployée sur un certain nombre de postes etc.

On réunit alors l'ensemble des fichiers nécessaires à l'application (les .class et autres ressources), dans **un seul fichier** , portant l'extension `.jar` (*java archive*).

Pour cela, il est nécessaire de fournir un **fichier manifest** , une sorte de fichier de configuration qui **désigne la classe principale** .

Structure du fichier manifest :

```
1: Manifest-Version: <numéro de version>
2: Main-Class: <nom d'une classe ayant une fonction main>
3: <une ligne vide>
```

Exemple (les numéros de lignes n'appartiennent pas au contenu du fichier) :

```
1: Manifest-Version: 1.0
2: Main-Class: org.vincimelun.cyber.Scene
3:
```

Plaçons ces trois lignes dans un fichier crée pour l'occasion (`manifest.mf`), et créons le fichier de déploiement `cyberpromeneur.jar` :

```
$ jar cvfm cyberpromeneur.jar manifest.mf org/
```

10.5. Exécution

Exemple d'exécution en ligne de commande

```
$ java -jar cyberpromeneur.jar
```

Vous remarquerez la présence de l'option **-jar** entre `java` et le nom de l'archive.

11. TP - réalisation d'un composant d'envoi de méls

11.1. Intro

Nous souhaitons mettre à la disposition de nos applications Java, un objet permettant d' **envoyer des emails** (*mél* en français).

Nous nous appuierons pour cela sur l' `API JavaMail` de `Sun` . Mais avant d'étudier une façon d'envoyer un *mel* avec cette API, nous concevons une IHM Graphique en Swing.

11.2. Analyse 1

L'IHM devra permettre la saisie des informations suivantes :

- **FROM** : adresse Internet de l¿expéditeur
- **TO** : adresse Internet des destinataires
- **CC** : adresse Internet des destinataires en copie
- **DATE** : date et heure de l'expédition (automatique)
- **SUBJECT** : l'objet du message
- **MESSAGE** : le message à transmettre

ainsi que l'envoi ou l'annulation (vider les champs)

11.3. Maquette

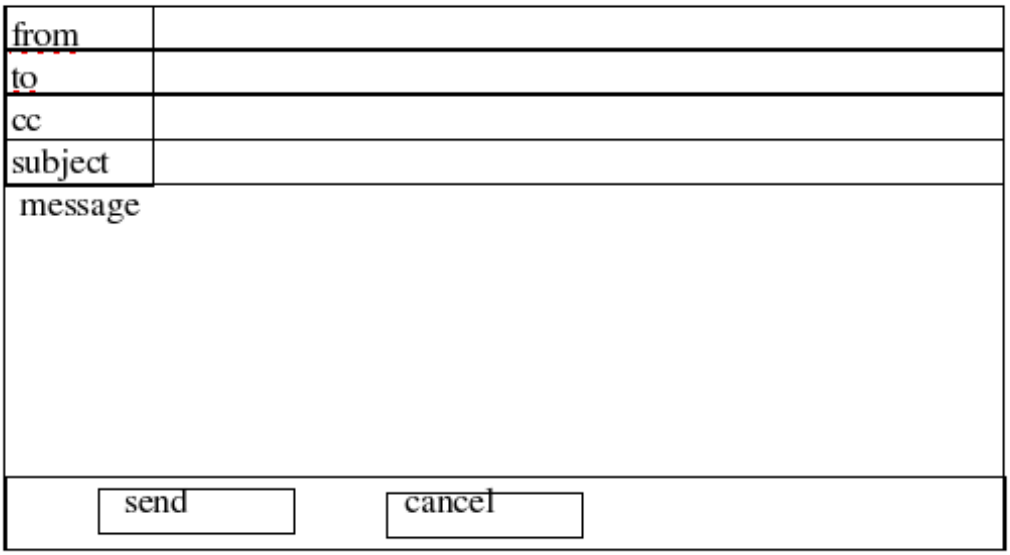


Figure 27. Maquette approximative du composant

Remarque : le positionnement de la date reste à déterminer.

11.4. Conception 1

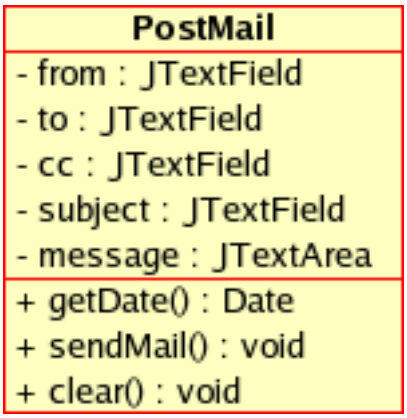


Figure 28. Diagramme de la classe visuelle du composant

11.5. Quelle classe de base ?

Si nous choisissons une `JFrame` (une classe de fenêtre), nous aurons du mal à l'intégrer dans une autre fenêtre. Le plus souple serait alors de choisir un panel, soit la classe `JPanel` .

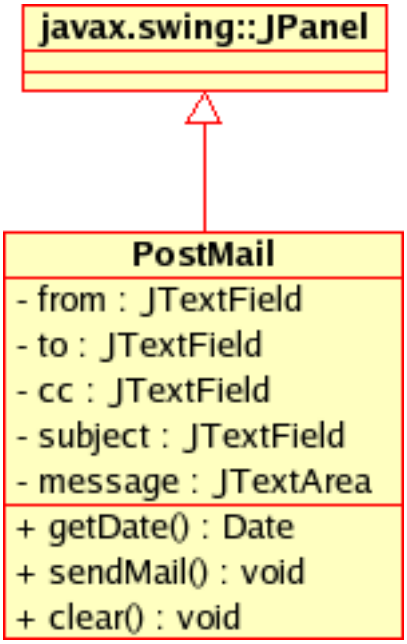


Figure 29. Le composant visuel sera un panel

Code source :

```
import javax.swing.*;
import java.util.*;
import java.awt.*;

/**
  * class PostMail
  *
  */

public class PostMail extends JPanel
{

/**Attributes: */

  private JTextField to;
  private JTextField from;
  private JTextField cc;
  private JTextField subject;
  private JTextArea message;
  private Frame frame; // son conteneur
/** constructeur */
public PostMail(Frame owner) {
  this.frame=owner;
  init();
}

/**
* Création et positionnement des éléments
* de l'IHM.
*/
private void init(){
  to = new JTextField();
  from = new JTextField();
  cc = new JTextField();
  subject = new JTextField();
  message = new JTextArea();
```

```
  this.setLayout(new BorderLayout());
  JPanel panel = new JPanel(new GridLayout(0,2));
    // un panel de 2 colonnes et de n lignes
  panel.add(new JLabel("From    : "));
  panel.add(from);
  panel.add(new JLabel("To      : "));
  panel.add(to);
  panel.add(new JLabel("Cc      : "));
  panel.add(cc);
  panel.add(new JLabel("Subject : "));
  panel.add(subject);
  panel.add(new JLabel("Message : "));

  this.add("North", panel);
  this.add("Center", message);
}

/** Public methods: */

public void sendMail(  )
  {

  }

public void clear(  )
  {

  }

}
```

11.6. Programme de test

Nous construisons une application test permettant d'embarquer notre objet (composant).

Nous concevons une classe `Test` disposant d'une méthode `main` .

```
i
public class Test {
  static public void main(String[] args){
    JFrame frame = new JFrame("Test PostMail");
    PostMail postMail = new PostMail(frame);
    frame.getContentPane().add(postMail);
    frame.setSize(400,300);
    frame.setVisible(true);
  }
}
```

On place notre composant (un `JPanel`) **dans** le `contentPane` d'une `JFrame` .

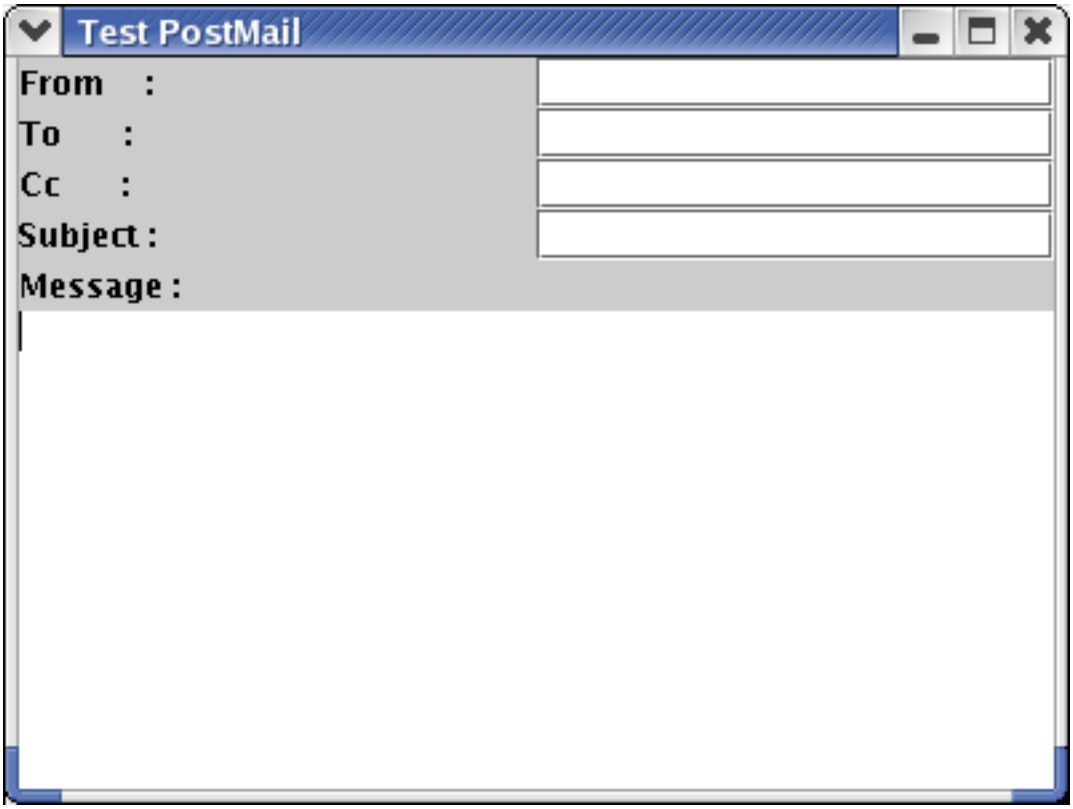


Figure 30. Première version du composant

11.7. ITÉRATION : mise au point par retour sur les points d'analyse et de conception

Critiques, les premiers points à améliorer :

- **Problème** : la fenêtre de test ne se ferme pas correctement

  Solution : dans le main, appeler la méthode `setDefaultCloseOperation` de la fenêtre avec le bon paramètre (voir Etape 2 de Programmer des interfaces graphiques).

- **Problème** : la fenêtre est coupée en 2 parties égales (libellés et champs de saisie). C'est le `GridLayout` qui est responsable de cela (c'est son contrat). Comment faire pour réserver une taille fixe à la zone des libellés, tout en conservant la partie élastique des champs de saisie ?

  Solution : ne pas utiliser de `GridLayout` mais un `BorderLayout` qui nous permette de placer les libellés à gauche (*West*) et les champs de saisie à droite (*Center*).

  Pour créer un panel à "structure rigide", que nous appellerons par exemple `panelGauche` , nous devons spécifier une dimension "préférée" (une donnée utilisée par les `Layout`). Exemple de création :

```
...
 this.setLayout(new BorderLayout());
 JPanel panel = new JPanel(new BorderLayout());
 JPanel panelGauche = new JPanel(new GridLayout(1,1));
 panelGauche.setSize(80,100);
 panelGauche.setPreferredSize(new Dimension(80,100));
 ...
```

rien n'empêche ensuite de placer dans ce panel un autre panel avec un gestionnaire de positionnement (`layout`) approprié.

L'exemple proposé ici [26](à exécuter avec la commande `java _TestLayout)` , n'utilise que des `GridLayout` et `BorderLayout` .

- **Problème** : il n'y a pas de boutons !

  Solution : à vous de jouer. Ils devront être placés au sud et au centre (sur l'axe des X) de la fenêtre.

- **Problème** : il faut une vérification "raisonnée" de validité des champs. On pourra, dans un premier temps, simplement vérifier s'ils ne sont pas "vide".

  On **ne** tiendra **pas** compte du fait qu'il puisse y avoir plusieurs destinataires.

  Solution : plongez dans l'API swing. Il se pourrait bien qu'un `JTextField` ait un attribut privé nommé `text` .

  Sous l'action du bouton "envoyer", une boîte de dialogue apparaîtra. Elle présentera soit une explication d'erreur, soit un **récapitulatif** des informations (from, to et l'objet).

- **Problème** : doit-on accepter un message sans objet ?

  Solution : en absence d'objet (*subject*), et si tous les autres champs sont corrects, le programme doit demander à l'utilisateur s'il souhaite vraiment envoyer son message sans objet. S'il confirme, le programme alloue à l'objet la valeur "(sans objet)" et poste le message, sinon rien ne se passe et retour à l'interface.

- **Problème** : l'adresse *from* est-elle bien formée ?

  Solution : difficile à vérifier. Toutefois, on créera une fonction privée (qui sera déclenchée au moment de la demande d'envoi), que nous nommerons `badEMail` .

  Cette fonction sera chargée de vérifier la présence, **dans son paramètre** , d'un seul caractère '@'. Ce dernier ne devant se trouver ni en première ni en dernière position, et obligatoirement à gauche d'un point, lui-même situé à plus d'un caractère de l'arobase, sans être le dernier... Si cette condition n'est pas respectée, alors la fonction rendra `true` , sinon `false` .

  Remarque : cela peut paraître étrange de rendre VRAI lorsque la condition n'est pas respectée. Mais à y regarder de près, il est plus correct d'affirmer qu'une adresse est fausse que valide !

11.8. la suite...

Patience, l'itération suivante, objet de la prochaine section, se focalisera sur la fonction d'envoi de message.

12. TP suite : la fonction d'envoi

Cette seconde itération va donc nous replonger dans des activités d'analyse, de conception et de test.

Pour l'envoi d'un mail, nous nous appuierons sur un projet nommé `JavaMail` .

`Java Mail` est l'API de référence permettant à des programmes java d'envoyer et de recevoir des méls en se connectant à des serveurs de messagerie. Vous pouvez donc, avec `JavaMail` concevoir des clients de messagerie genre `Evolution` , `Messenger` et autres `OutLook` ... Mais dans la majorité des cas, vous utiliserez `JavaMail` pour *envoyer* des méls, par exemple pour

[26] _TestLayout.class

valider une inscription à une liste, confirmer une commande, etc.

Prise en main rapide de `JavaMail` pour l'envoi de courrier

1. Télécharger la dernière version de `JavaMail` , une implémentation de `Sun` de l'API.

   Lien : http://java.sun.com/products/javamail/index.html [27]

2. Télécharger la dernière version de `JavaBeans Activation Framework` de `Sun` .

   Lien : http://java.sun.com/products/javabeans/glasgow/jaf.html [28]

3. Placer les fichiers `mail.jar` et `activation.jar` dans le répertoire du projet.

4. Compiler le programme exemple MailExample.java [29](dérivé d'un exemple livré avec `JavaMail`):

   ```
   $ javac -classpath mail.jar:activation.jar:. MailExample.java
   ```

5. Exécuter le programme en passant de bons arguments :

   ```
   $ java -classpath mail.jar:activation.jar:. MailExample \
       un-serveur-de-mail from-adresse to-adresse
   ```

   Exemple :

   ```
   $ java -classpath mail.jar:activation.jar:. MailExample \
      smtp.vinci-melun.fr monAdresse  monAdresse
   ```

6. Connectez-vous sur votre messagerie Web habituelle, et vérifiez la bonne réception du mél.

12.1. MailExample

Voici le code du programme exemple :

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;

public class MailExample {
  public static void main (String args[]) throws Exception {
    String host = args[0];
    String from = args[1];
    String to = args[2];

    // obtenir des propriétés du sytème
    Properties props = System.getProperties();

    // inscrire le nom du serveur SMTP que nous
    // utiliserons
    props.put("mail.smtp.host", host);

    // Création d'une session utilisateur, sur la
    // base des propriétés du système
    Session session = Session.getDefaultInstance(props, null);

    // Création d'un message Mime à partir de la session
```

[29] MailExample.java

```
    MimeMessage message = new MimeMessage(session);

    // Inscrition de l'adresse de l'expéditeur
    // (une chaîne de caractères)
    message.setFrom(new InternetAddress(from));

    // Ajout d'une adresse de destination
    message.addRecipient(Message.RecipientType.TO,
      new InternetAddress(to));

    // idem, en "copie carbone"
    // messsage.addRecipient(Message.RecipientType.CC,
    //   new InternetAddress(cc));

    // fournir un objet au mel
    message.setSubject("Hello JavaMail");

    // ainsi qu'un contenu (une chaîne de caractères)
    message.setText("Welcome to JavaMail");

    // la date d'envoi (optionnel)
    message.setSentDate(new java.util.Date());

    // envoi du message en utilisant le service 'send' de la
    // classe Transport ('send' est une méthode 'static')
    Transport.send(message);
  }
}
```

À la lecture de ce code source, vous découvrez enfin l'objet et le contenu du message que vous vous êtes transmis.

12.2. Analyse : comment intégrer la fonction d'envoi dans notre composant ?

Nous pourrions simplement copier le programme ci-dessus et le coller dans le corps de la méthode `sendMail` , puis aménager certaines parties, en référence à nos `JTextField` , mais il y a mieux à faire.

Nous allons créer une classe dont la fonction principale sera d' *encapsuler* le comportement du programme exemple. Nous créons ainsi un composant **sans interaction avec l'utilisateur** , et permettant l'envoi d'un mél à plusieurs destinataires.

Ainsi, notre composant IHM (un JPanel) n'aura qu'à instancier cet autre composant non visuel, pour pouvoir envoyer un mél, selon les directives de l'utilisateur.

Vous avez peut-être remarqué que la méthode `main` déclare dans son entête une exception.

La mise au point de notre composant va nécessiter une **gestion des exceptions** . Nous ferons donc un détour (chapitre suivant) afin d'apprendre ce qu'est une exception et comment les gérer, puis nous poursuivrons la construction de notre composant d'envoi de méls.

13. Comprendre la gestion des exceptions

Une exception est un événement déclenché par une erreur survenant à l'exécution.

Le déclenchement d'une exception provient d'une fonction qui se trouve dans l'incapacité de réaliser son **contrat** , c'est à dire le service dont elle a la charge.

Prenons un exemple de fonction (une méthode) chargée d'établir une connexion à une base de données :

```
...
DB.connectionBaseDeDonnées( nom_de _la_base );
...
```

Cet appel de méthode peut échouer pour 2 raisons :

1. Le paramètre fourni à la méthode ne correspond pas à un nom de base valide.
2. Le serveur de la base de données ne répond pas.

Dans le premier cas, c'est le **client** qui est responsable (client : utilisateur de la méthode, l'appelant).

Dans le second cas, le responsable est le **fournisseur** (la fonction appelée).

Une erreur survenant à l'exécution, un **bogue** , génère une exception.

Une exception est une instance (un objet) de type `Throwable` . Java distingue trois types d'erreur à l'exécution : **Error** , **RuntimeException** et les autres, les plus nombreuses, qualifiées d' **Exceptions Contrôlées** , représentées par le diagramme suivant :

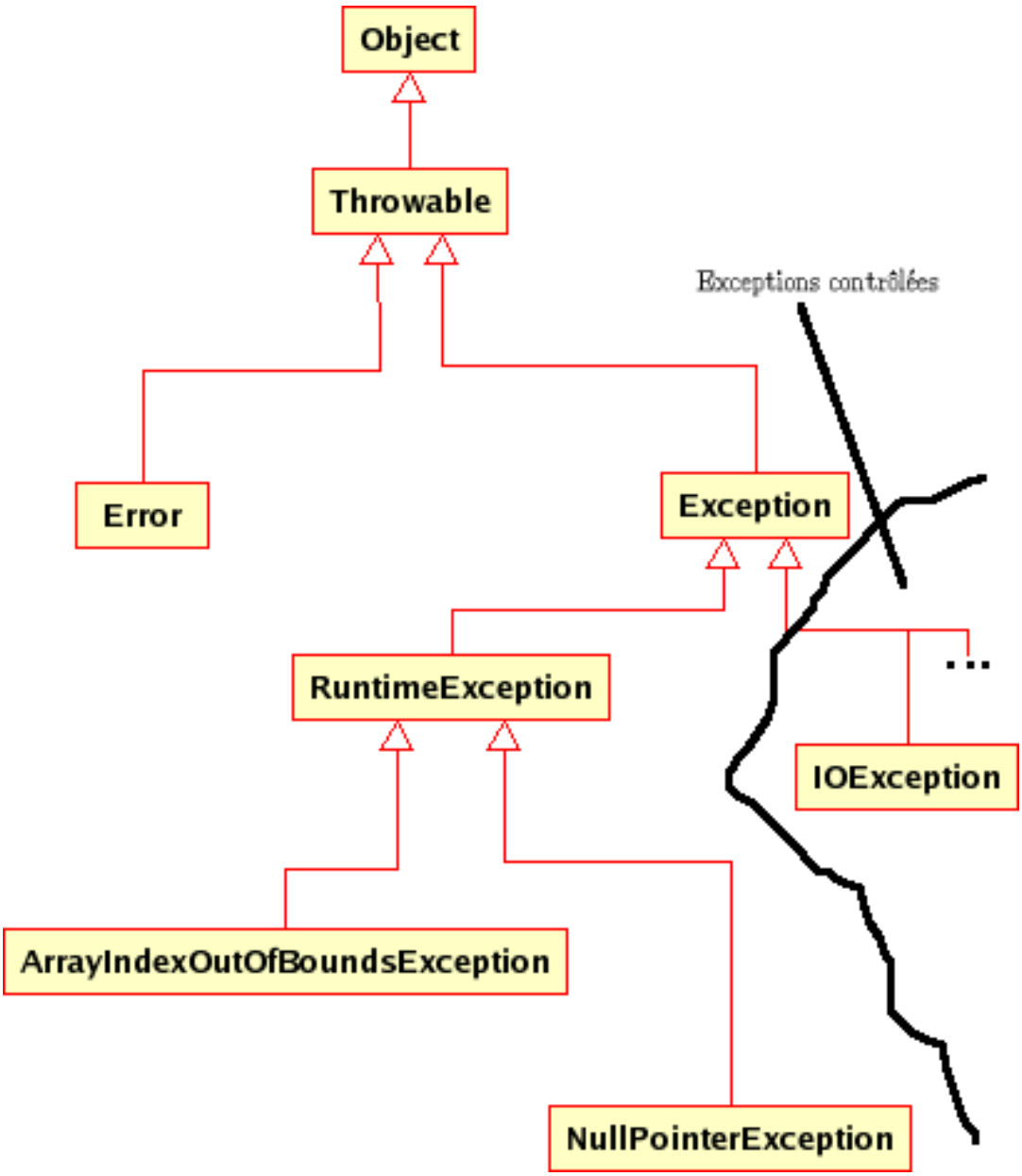


Figure 31. Extrait de la hiérarchie des classes d'exceptions

Les **RuntimeException** sont des exceptions internes à la JVM pouvant être déclenchées dans de multiples contextes. Ces exceptions comprennent les exceptions arithmétiques (par exemple la division par zéro), les exceptions de pointeur (lors d'une tentative d'accès à un objet par l'intermédiaire d'une référence null), et les exceptions d'indexation (valeur d'index hors bornes).

Les **Error** sont des exceptions liées aux problèmes de chargement des classes en mémoire, problème matériel...

Les **Exceptions Contrôlées** sont des exceptions d'entrée/sortie, et typiquement d'autres définies par vous-mêmes ou par les objets des librairies/package que vous utilisez.

Remarque : une exception est un **objet** , instance d'une classe d'exception.

13.1. Comment générer une exception ?

Une méthode ne pouvant assurer sa fonction ne **doit pas retourner de valeur** , elle doit lancer (générer) une exception.

Exemple, la méthode `connexionBaseDeDonnées` ne peut assurer sa fonction si le serveur de base de données ne répond pas.

Elle prévient alors l'appelant qu'elle est susceptible de générer une exception par la clause **throws** :

```
public void connexionBaseDeDonnées(...)
  throws ConnexionImpossibleException
```

... l'exception sera générée soit par un composant interne utilisé par la méthode, soit directement par la méthode, en créant un objet d'une classe d'Exception, et en le propageant par **throw** . Exemple :

```
public void connexionBaseDeDonnées(...)
throws ConnexionImpossibleException {
  if (serveur.noResponding()) {
    throw new ConnexionImpossibleException();
  }
  ...
}
```

Charge alors à l'appelant d' **intercepter** (*catch*) l'exception.

Attention, il y a deux mots clés ici : **`throw`** et **`throws`** (avec un ' **s** ').

13.2. Comment intercepter une exception ?

L'appelant intercepte une exception en plaçant le code à risque dans le corps d'un **`try`** , puis en filtrant les exceptions selon leur classe, par un **`catch`** . Exemple :

```
...
try {
  DB.connexionBaseDeDonnées();
  String sql ="Select * from clients";
  DB.excuteSql(sql);
  ...
 }
 cath (ConnexionImpossibleException ex) {
    // faire quelque chose ici
 }

...
```

Prise de responsabilités

Java *impose* que les **Exceptions Contrôlées** soient interceptées (*catch*) ou spécifiées (*throws*). Voir Extrait de la hiérarchie des classes d'exceptions.

En d'autres termes, le langage *oblige* le développeur à prendre explicitement ses responsabilités en ce qui concerne les Exceptions Contrôlées.

13.3. Comment provoquer l'évaluation d'une instruction coûte que coûte ?

La clause **finally** est là pour ça, à placer après le corps du `try` (comme les *catch*). Exemple :

```
..
 try {
   DB.connectionBaseDeDonnées();
   Strin sql ="Select * from clients";
   DB.excuteSql(sql);
   ...
 }
 cath (ConnectionImpossibleException ex) {
    // faire quelque chose ici
 }
 finally {
   afficher ("Merci d'être venu.");
 }
...
```

Le message *Merci d'être venu.* sera **toujours** affiché, à moins que la méthode `afficher` ne soit à l'origine d'une exception.

13.4. Exercice

Voici un programme Java qui demande l'année de naissance de l'utilisateur, et retourne son age en retour (à 12 mois près...)

```
import java.util.*;
import javax.swing.*;

class TestGE {

  public void run() {
    String input =
       JOptionPane.showInputDialog(
           "Entrez votre année de naissance");
    String age = calculerAge(input);
    JOptionPane.showMessageDialog(
           null, "Vous avez "+age+" ans.");
  }

  public String calculerAge(String annee) {
    String age;
    Calendar now = Calendar.getInstance();
    int ann_nais = Integer.parseInt(annee);
    age = String.valueOf(now.get(Calendar.YEAR)-ann_nais);
    return age;
  }

  static public void main(String[] args) {
     TestGE test = new TestGE();
     test.run();
     System.exit(0);
  }

}//TestGE
```

Ce programme manque sérieusement de robustesse. Encore une fois les données situées à l'extérieur en sont la cause. Est-ce bien raisonnable de faire confiance à l'utilisateur ? Que se passe-t-il s'il rentre `mille neuf cent quatre vingt quatre` au lieu de `1984` ?

Proposez une solution.

=>Une bonne solution : TestGE.java [30], une autre possible, moins satisfaisante et plus longue _TestGE.java [31].

[30] solution/TestGE.java

14. Améliorer la robustesse du composant `JPostMail`

Nous allons mettre en pratique la gestion des exceptions, afin de rendre exploitable notre composant par des utilisateurs lambda.

Nous souhaitons architecturer notre application en 2 classes aux responsabilités bien distinctes : l'une (`JPostMail`) aura la responsabilité d'interagir avec l'utilisateur (paramétrage, messages d'erreur, etc.) et l'autre (`SendMail`) aura la responsabilité d'envoyer un mél (et d'interagir avec les packages `mail.jar` et `activation.jar`).

14.1. Conception de `SendMail` , une classe d'envoi de méls, sans AUCUNE interaction avec un utilisateur.

A partir de la classe `MailExample` , nous implémentons une classe `SendMail` en Java.

```
/* Fichier : SendMail.java
 * Créé le : 31 janv. 2004
 * par : kpu
 */

import java.util.List;
import java.util.Properties;
import java.util.Vector;

import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.AddressException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

/**
 *
 * Pour envoyer un mel à un ou plusieurs destinataires
 *
 * @author kpu
 * @version 1.0, 31 janv. 2004
 */
public class SendMail {

  /**
   *  le nom du serveur de mel
   */
  private String host;

  private String from;
  private List to;
  private List cc;
  private String subject;
  private String message;

  /**
   *
   */
  public SendMail() {
      to = new Vector();
      cc = new Vector();
      from = "";
      host = "";
      subject = "";
      message = "";
```

[31] solution/_TestGE.java

```
}

/**
 * @return
 */
public String getHost() {
  return host;
}

/**
* @param string
*/
public void setHost(String string) {
  host = string;
}

/**
 * @return
 */
public String getFrom() {
    return from;
}

/**
 * @return
 */
public String getMessage() {
    return message;
}

/**
 * @return
 */
public String getSubject() {
    return subject;
}

public void addTo(String mel_dest) {
   this.to.add(mel_dest);
}

public void addCc(String mel_dest) {
  this.cc.add(mel_dest);
}

/**
 * @param string
 */
public void setFrom(String string) {
    from = string;
}

/**
 * @param string
 */
public void setMessage(String string) {
    message = string;
}

/**
 * @param string
 */
public void setSubject(String string) {
    subject = string;
}


public void send() throws AddressException, MessagingException {
  // obtenir des propriétés du sytème
```

```
    Properties props = System.getProperties();

    // inscrire le nom du serveur SMTP que nous
    // utiliserons
    props.put("mail.smtp.host", host);

    // Création d'une session utilisateur, sur la
    // base des propriétés du système
    Session session = Session.getDefaultInstance(props, null);

    // Création d'un message Mime à partir de la session
    MimeMessage message = new MimeMessage(session);

    // Inscrition de l'adresse de l'expéditeur
    // (une chaîne de caractères)
    message.setFrom(new InternetAddress(from));

    // Ajout d'une adresse de destination
    // (la première de la liste)
    message.addRecipient(Message.RecipientType.TO,
          new InternetAddress(getTo(0)));

    if (cc.size() != 0) {
      // idem, en "copie carbone"
      message.addRecipient(Message.RecipientType.CC,
         new InternetAddress(getCc(0)));
    }

    // fournir un objet au mel
    message.setSubject("Hello JavaMail");

    // ainsi qu'un contenu (une chaîne de caractères)
    message.setText("Welcome to JavaMail");

    // la date d'envoi (optionnel)
    message.setSentDate(new java.util.Date());

    // envoi du message en utilisant le service 'send' de la
    // classe Transport ('send' est une méthode 'static')
    Transport.send(message);
 }

 public static void main(String[] args)
 throws AddressException, MessagingException {
    SendMail sendMail = new SendMail();
  /* un test
    sendMail.setHost("un serveur smtp");
    sendMail.setFrom("mon adresse");
    sendMail.addTo("mon adresse");
    sendMail.addTo("une autre adresse");
    sendMail.setSubject("un objet");
    sendMail.setMessage("le texte du message");
    sendMail.send();
  */
   System.out.println("Message envoyé avec succès.");
  }
}
```

On remarquera la méthode `send` qui spécifie deux types d'exceptions susceptibles d'être levées par la version actuelle de cette méthode.

14.2. Exercices

1. Le code présenté (`SendMail.java`) manque cruellement de commentaires. Complétez-le. Vous trouverez sur developpeur.journaldunet.com/tutoriel/jav/ [32]une présentation rapide de

[32] http://developpeur.journaldunet.com/tutoriel/jav/030117jav_javadoc1.shtml

javadoc .

2. Ajoutez (à `SendMail`) un constructeur un peu plus fonctionnel que celui présenté.

   Vous n'effectuerez **aucun** contrôle de validité sur les données fournies (host, from,etc.). C'est à l'appelant de fournir des données correctes. A lui d'intercepter une des deux exceptions susceptibles d'être déclenchées.

3. Fournir des données de test correctes afin de tester le bon fonctionnement de l'objet (cela se passe dans le *main*).

```
$ javac SendMail.java
$ java SendMail
```

4. Retour à `JPostMail`

5. Renommer la classe `PostMail` en `JPostMail` (J = référence à swing). Vous la placerez dans le même package que `SendMail` .

   La calsse `JPostMail` étant déclarée `public` , vous devez également renommer le fichier source par le même nom, soit `JPostMail.java` .

6. Le composant visuel `JPostMail` utilise un objet `SendMail` pour envoyer un mél (à l'image de ce qui se passe dans le *main* de `SendMail`).

   Réaliser cette intégration. Attention, la méthode `send` de `JPostMail` devra intégrer une **gestion des exceptions (try ... catch)** .

   Pour cela, vous instanciez (avec **new**) cet objet au moment de l'envoi afin d'utiliser ses services.

   Attention : la méthode `send` de notre objet `SendMail` est susceptible de déclencher une exception qui devra impérativement être gérée par notre objet `JPostMail` :

```
ublic class JPostMail
  extends JPanel
  implements ActionListener
{
  /**Attributes: */
  final String ENVOYER = "envoi";
  final String ANNULER = "annule";

  private JTextField to;
  private JTextField from;
  private String host;
  etc.

public void sendMail( )
   throws AddressException, MessagingException

{
   SendMail sendMail = new SendMail();
   // il faudra prévoir interagir avec l'utilisateur
   // pour stocker le nom du serveur de mail
   // dans une variable (nommée host)
   send.setHost(host);

   send.setFrom(from.getText().trim());

   // ...
   // etc. pour l'objet, le message, les destinatires ...
   // ...

   // puis, finalement, envoyer le mel
```

```
   sendMail.send();
}

 // comme avant

public void actionPerformed(ActionEvent e) {
  if (e.getActionCommand().equals(ENVOYER)) {
    if (verifChamps()) {
    // JOptionPane.showMessageDialog(null, "Ok pour l'envoi");

    try {
        sendMail();
    }
    catch (javax.mail.internet.AddressException ex) {
        JOptionPane.showMessageDialog(null,
              "ECHEC à l'envoi \n"+ex.toString());
      }
     catch (javax.mail.MessagingException ex) {
        JOptionPane.showMessageDialog(null,
              "ECHEC à l'envoi \n"+ex.toString());
      }

    }

    else {
      JOptionPane.showMessageDialog(null,
              "Certains champs sont mal formés");
    }
  }
  else if (e.getActionCommand().equals(ANNULER)) {
    clear();
  }
}
```

7. On souhaite présenter à l'utilisateur une boîte de dialogue qui :

   - confirme le succès de l'opération (message de succès).
   - informe de l'échec de l'opération, et sa cause.

     (Plus difficile) On mettra en valeur la donnée incriminée : host, to, from etc.

     =>pour cela, placer le message de l'exception dans une `String` , puis rechercher une occurrence d'une des valeurs fournies par l'utilisateur (détenues par les attributs). La classe `String` dispose d'une méthode nommée `indexOf` bien pratique pour rechercher une sous-chaîne.

Bonne programmation.

14.3. Plus loin

Problème d'envoi de courrier à partir de votre domicile ?

Si vous utilisez un serveur `SMTP` autre que celui de votre prestataire de connexion, vous devrez vraisemblablement lui fournir un nom d'utilisateur et un mot de passe.

Dzenan Ridjanovic (http://drdb.fsa.ulaval.ca/sujets/javamail/ [33]), de l'Université Laval, a extrait des programmes exemples, un moyen pour cela (exemple 2). Il présente aussi quelques liens intéressants sur le sujet.

[33] http://drdb.fsa.ulaval.ca/sujets/javamail/

N'hésitez pas également à consulter le code source des programmes du répertoire `demo` de `JavaMail` .

14.4. Exercice à faire à la maison et à rendre

Ce que l'on attend de vous :

- Le code source **commenté** de votre composant. Vous soignerez particulièrement l'implémentation des méthodes `badEMail` et `verifChamps` ainsi que l'envoi à de multiples destinataires (en un seul appel) (**noté sur 10**)
- Un **rapport** présentant vos **apports personnels** , et éventuellement les problèmes non résolus, et **un diagramme de classe UML** , *sans les getter/setter* . (**noté sur 8**)
- Un exécutable (code compilé) et un **mode d'emploi pour lancer le test** . (**noté sur 2**)

Pour les curieux, voici un extrait d'une FAQ (FAQ sur jguru [34]) sur le sujet :

source : Comment packager mon applicaton dans un seul fichier .jar ? [35]

```
Question
  How do I package a JavaMail application into
   a single jar file along with the mail.jar and activation.jar?
Answer
 You need to unjar all the JAR files into a single directory tree
 and then JAR them back up.
 The trick is preserving the location of some files:

 425 Fri Dec 01 00:05:16 EST 2000 META-INF/javamail.default.providers
  12 Fri Dec 01 00:05:16 EST 2000 META-INF/javamail.default.address.map
1124 Fri Dec 01 00:05:16 EST 2000 META-INF/javamail.charset.map
 469 Fri Dec 01 00:05:16 EST 2000 META-INF/mailcap
```

Donc la démarche à suivre est :

- créer un repertoire <temp>
- y copier `mail.jar, activation.jar, SendMail.class, JPostMail.jar`
- se déplacer dans ce répertoire et décompresser les fichiers jar (par exemple : `unzip mail.jar ; unzip activation.jar`)

  attention, renommer le fichier `MANIFEST.MF` d' `activation.jar` afin qu'il n'écrase pas celui de `mail.jar` . Puis fusionner ces deux fichiers (pr la commande `cat` et une double redirection `>` . Supprimer l'entête en double, et ajouter une ligne `Main-Class: JPostMail` .

  le fichier résultat est ici : MANIFEST.MF [36]. Copier-le dans le répertoire `META-INF` .
- Enfin, créer le fichier de déploiement par la commande : `jar cvfm postmail.jar META-INF/MANIFEST.MF .`
- Il ne vous reste plus qu'à le tester.

  copier `postmail.jar` dans un nouveau répertoire et taper la commande : `java -jar postmail.jar`

[34] http://www.jguru.com/faq/
[35] http://www.jguru.com/faq/view.jsp?EID=1111058
[36] MANIFEST.MF

15. Gestion de fichiers

15.1. Intro

Il existe deux types de format de fichier : **texte** et **binaire** . Est qualifié de "fichier texte", tout fichier composé exclusivement de caractères dédiés ou compatible avec une impression standard, un affichage en mode console.

Par exemple un code source est un fichier texte, par contre un fichier MS World 2000 *n'est pas* un fichier texte, car il comporte des caractères non affichables (information sur la police, objets incorporés, et autres informations plus ou moins documentées par l'éditeur)

ATTENTION : On appelle format de fichier, la structure de son **contenu** , et non la valeur de son **extension** . Un fichier portant le nom de "document.txt" est certainement un fichier de type texte, pouvant être lu par un éditeur quelconque, mais **rien ne le garantit !** L'inverse est également vrai : un fichier nommé "document.jpg" est certainement un fichier de type binaire (ici une image), mais ce n'est qu'une convention.

Le format texte

Contrairement au fichier binaire, le contenu d'un fichier texte est constitué de valeurs correspondant, pour la majorité, à des codes de caractères affichables (plus certains interprétables comme \n (fin de ligne), \t (tabulation)...).

Les fichiers au format texte sont le plus souvent structurés soit par :

1. **des lignes non formatées, séparées par un symbole de fin de ligne.**
2. **des lignes formatées CSV (Comma Separated Value)** une ligne est contituée d'une suite de champs séparés par un caractère spécial (une virgule par exemple) et d'un symbole de fin de ligne.
3. **des balises** , c'est le cas des fichiers XML (*eXtensible Markup Language*) - non étudié ici.
4. des mots clés d'un langage et une grammaire (comme le code source java) - non étudié ici.

Exemple document.txt

```
[kpu@kpu seance-15]$ cat document.txt
Bonjour,
futur informaticien,
nous vous souhaitons
bonne aventure.
```

Version hexadécimale

```
[kpu@kpu seance-15]$ hexdump document.txt
0000000 6f42 6a6e 756f 2c72 660a 7475 7275 6920
0000010 666e 726f 616d 6974 6963 6e65 0a2c 6f6e
0000020 7375 7620 756f 2073 6f73 6875 6961 6f74
0000030 736e 0a20 6f62 6e6e 2065 7661 6e65 7574
0000040 6572 0a2e
0000044
```

Version caractère

```
[kpu@kpu seance-15]$ hexdump -c document.txt
0000000   B   o   n   j   o   u   r   ,  \n   f   u   t   u   r      i
```

```
0000010   n   f   o   r   m   a   t   i   c   i   e   n   ,  \n   n   o
0000020   u   s       v   o   u   s       s   o   u   h   a   i   t   o
0000030   n   s      \n   b   o   n   n   e       a   v   e   n   t   u
0000040   r   e   .  \n
0000044
```

A titre d'information, il existe un outil bien pratique, nommé `file` , qui analyse le contenu d'un fichier pour en extraire des informations sur son type. Exemple :

```
$ file document.txt
document.txt: ASCII text

$ file zoo.png
zoo.png: PNG image data, 446 x 182, 8-bit/color RGB, non-interlaced

$ file TestGE.java
TestGE.java: UTF-8 Unicode Java program text
```

15.2. Java et la gestion des fichiers

Java considère un fichier comme un cas particulier d'un *flux* (*stream*).

Flux

Un flux est une séquence de caractères ou d'octets voyageant d'une origine vers une destinaton.

Un programme qui écrit dans un fichier est à l'origine d'un flux (ou *producteur*). Un programme qui lit le contenu d'un répertoire, d'un fichier, ou de toute autre ressource est considéré comme une destination (ou *consommateur*), qui reçoit progressivement le contenu du flux.

Java, pour des raisons pratiques, propose deux types de flux : les **InputStream/OutPutStream** et les **Reader/Writer** . Les premiers (`Stream`) sont utilisés pour la gestion de fichiers dit binaires, les `Reader/writer` sont spécialisés pour la gestion de flux de caractères UNICODE (16-bit) et Local (8-bit).

Notez qu'il existe de nombreuses classes de flux spécilisées (plus de 70 !), et certaines sont dans des packages utilitaires comme `java.util.zip` . Sachez également qu'il existe des passerelles pour passer d'une logique de flux d'octets à celle de flux de caractères (16-bit), ce sont les classes : `InputStreamReader` pour convertir un `InputStream` en un `Reader` et `OutputStreamWriter` pour convertir un `OutputStream` en un `Writer` .

Vous trouverez ici, une traduction d'un extrait d'un tutorial chez Sun [37], présentant la hiérarchie des classes I/O.

Les caractéristiques d'un fichier peuvent être interrogées en utilisant la classe `java.io.File` , qui est présentée comme une *représentation abstraite d'un fichier et d'un chemin de répertoire* . C'est cette classe qui connait par exemple le symbole de séparation des chemins (*pathSeparatorChar*) (par exemple `:` sous Unix et `;` sous MSWindows)

Les flux de caractères sont associés à des classes spécialisées dans la lecture des données (*reader*) et dans leur écriture (*writer*).

Les principales étapes de gestion d'un fichier sont :

1. **Ouverture** du flux

[37] traduc-overview-io.html

2. **Exploitation** (par lecture ou écriture)
3. **Fermeture** du flux

15.3. Exploitation en lecture d'un fichier texte

1 - Ouverture d'un fichier

```
// ouvre un fichier en lecture
FileReader fr = new FileReader(nomFic);

// passe par un buffer pour simplifier la lecture du fichier
BufferedReader buf = new BufferedReader(fr);
```

2 - Exploitatin en lecture séquentielle du fichier

```
// déclaration d'une chaîne de caractères
// afin de stocker la chaîne courante
// à chaque tour dans la boucle.
String ligne;

// lecture de la première ligne du fichier
ligne = buf.readLine();

// tant que la fin de fichier n'est pas atteinte
while (ligne != null) {
   // faire quelque chose avec ligne
   // ...
   // à la fin du corps de la boucle, lire la ligne suivante
   ligne = buf.readLine();
}
```

3 - Fermeture fichier

```
fr.close();
```

15.4. Exploitation en écriture d'un fichier texte

1 - Ouverture d'un fichier en écriture

```
// ouvre un fichier en écriture (1ère version)
FileWriter out  = new FileWriter(nomFic);

/* autre version avec choix du jeu de caractères
  (ici ISO Latin Alphabet No. 1 - ISO-LATIN-1)
OutputStreamWriter out =
  new OutputStreamWriter(new FileOutputStream(nomFic), "ISO-8859-1");
*/

// passe par un buffer pour simplifier l'écriture dans le fichier
PrintWriter pw = new PrintWriter(out);
```

Exemple d'écriture dans le fichier (relativement à la position courante dans le fichier)

```
// déclaration d'une chaîne de caractères
// afin de stocker la chaîne à écrire
```

```
String ligne = "coucou";
// écriture de la ligne dans le fichier
pw.write(ligne);
```

Exemple de fermeture fichier

```
pw.close();
```

15.5. Exercice

1. Concevoir un programme (`FileInfo.java`) de type console, qui reçoit le nom d'un fichier en argument (ligne de commande), et affiche certaines de ses caractéristiques. Exemple, le programme s'appelle `FileInfo` :

   ```
   $ java FileInfo document.txt
   document.txt est un fichier
   Taille : 68
   Chemin : document.txt
   Chemin absolu : /home/kpu/34/ig12/java/coursJava2/document.txt
   Date de dernière modification : 1077916583000
   Le fichier existe
   Le fichier peut être lu
   L'écriture est autorisée
   Le fichier n'est pas un répertoire
   $
   ```

   Vous n'avez pas besoin de l'ouvrir en lecture, utiliserez simplement la classe `java.io.File` pour les besoins énoncés.

15.6. Gestion de fiches Client

On souhaite gérer un fichier client (client.txt)

Une fiche client est caractérisée par plusieurs lignes, chacune représente successivement le nom, prénom, adresse rue, code postal, ville, pays, tel et email.

Donc une fiche est composée de 8 lignes. S'il y a 2 fiches, alors le fichier contiendra 8x2 = 16 lignes, et ainsi de suite. Exemple :

```
Durand
Michel
35 rue du lavoir
38000
Grenoble
France
0123456789
mdurand@nullepart.com
Valerian
Denise
250 avenue Louise
1050
Bruxelles
Belgique
00 32 (0)2 345 67 89
vale@belgeunefois.com
```

On souhaite produire une application permettant dans un premier temps de présenter, une à une, les

fiches client du fichier. Voici une maquette :

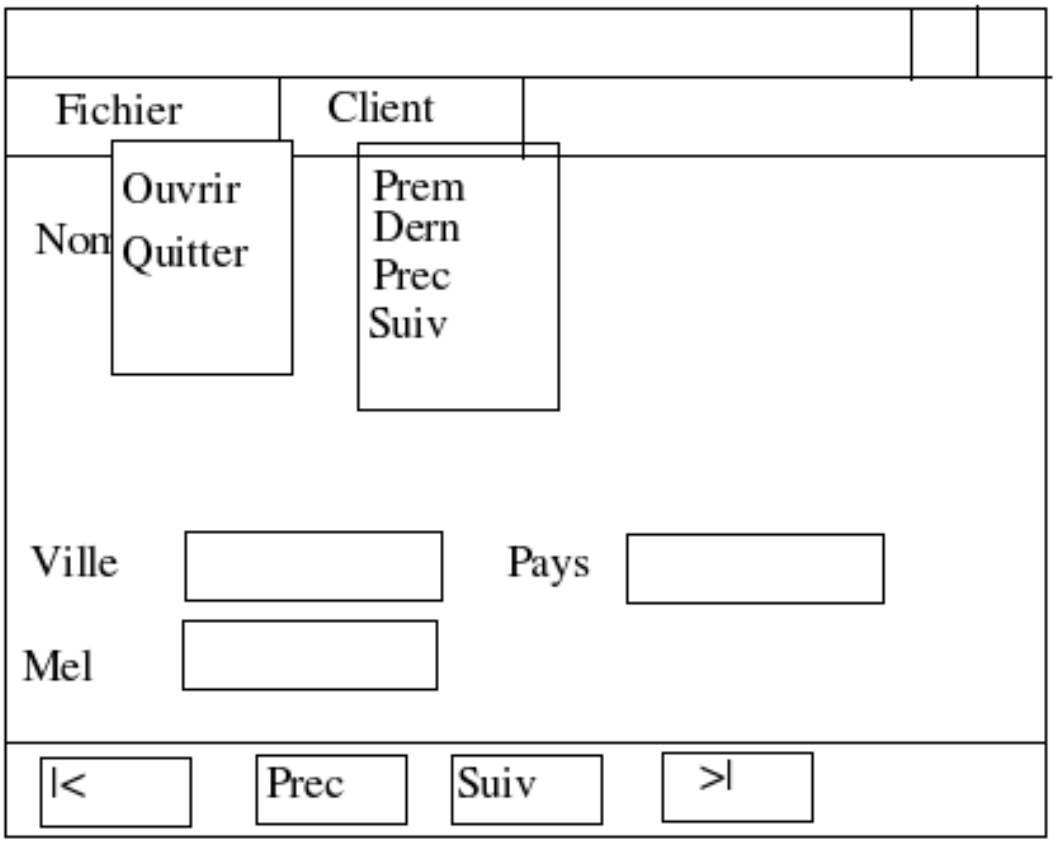


Figure 32. Maquette gestion de fiches Client

Première analyse. Nous concevrons trois classes : l'IHM (Interface Homme Machine) - une `JFrame` - , un contrôleur (*NavigatorClientFic*) qui détient une collection d'objets `Client` , et une classe `Client` représentant la structure type d'une fiche client.

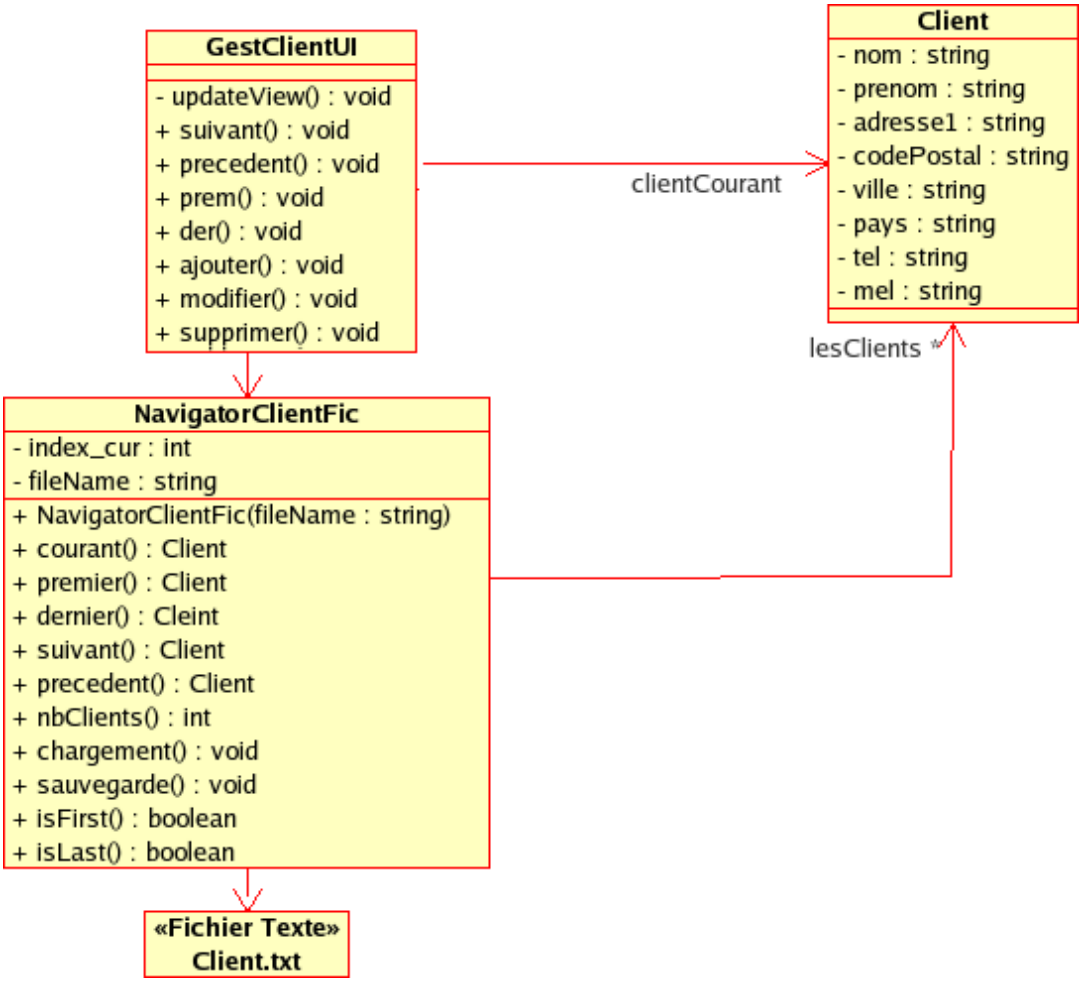


Figure 33. Diagramme de classes de l'application

La relation `NavigatorClientFic --> Client` dénote une liste nommée `lesClients` (un objet de type `Vector`). C'est un attribut privé de la classe `NavigatorClientFic` . Cet attribut est créé et valorisé dans le constructeur de `NavigatorClientFic` .

Donc, à la création, le fichier est lu et son contenu placé en mémoire sous forme d'une collection d'objets de type `Client` .

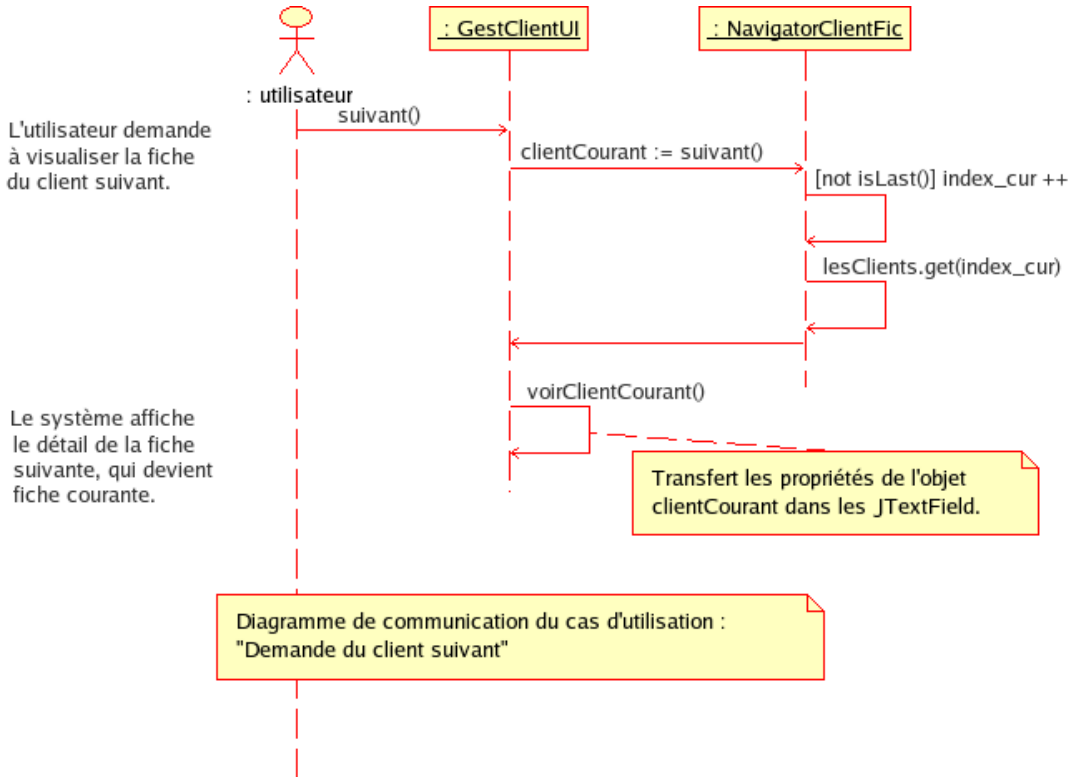


Figure 34. Diagramme de communication : Demande de visualisation de la fiche suivante

15.7. Exercices

1. Pour simplifier, nous considérons provisoirement qu'une fiche est caractérisée par un nom et un prénom (si on sait le faire pour 2, on saura le faire pour *n=8*).

   A FAIRE : Concevoir les classes `NavigatorClientFic` , et `Client` , de sorte que le programme de test suivant puisse fonctionner :

```
import java.io.*;
public class Test {

 public static void main(String[] a){
  try {
    NavigatorClientFic nav = new NavigatorClientFic(a[0]);

    System.out.println("-- Première fiche");
    System.out.println(nav.premier());
    System.out.println("-- Dernière fiche");
    System.out.println(nav.dernier());

    System.out.println("-- Voici les 3 premieres");
    System.out.println(nav.premier());
    System.out.println(nav.suivant());
    System.out.println(nav.suivant());
    System.out.println("-------------");

    System.out.println("-- Voici les \"3 précédentes\"");
    System.out.println(nav.precedent());
```

```
    System.out.println(nav.precedent());
    System.out.println(nav.precedent());
    System.out.println("-------------");

  }
  catch (FileNotFoundException e) {
    System.out.println("Fichier introuvable");
  }
  catch (IOException e) {
    System.out.println( e );
  }
 }
}
```

Sur le fichier de données suivant (`doc.txt`) :

```
Meyer
Bertrand
Goodwill
James
Kay
Michael
```

La commande `java Test doc.txt` , produit le résultat suivant :

```
$ java Test doc.txt
-- Première fiche
Nom : Meyer    prenom : Bertrand
-- Dernière fiche
Nom : Kay   prenom : Michael
-- Voici les 3 premieres
Nom : Meyer    prenom : Bertrand
Nom : Goodwill   prenom : James
Nom : Kay   prenom : Michael
-------------
-- Voici les "3 précédentes"
Nom : Goodwill   prenom : James
Nom : Meyer    prenom : Bertrand
Nom : Meyer    prenom : Bertrand
-------------
$
```

Remarquer que si la fiche courante est la première, la précédente est elle-même (même logique pour la fiche suivante de la dernière fiche).

Une solution : Client.java [38], NavigatorClientFic.java [39]

2. Modifier la solution proposée, afin qu'elle prenne en compte les huit propriétés énoncées précédemment. Voici un fichier contenant plusieurs fiches : clients.txt [40].

3. Implémenter les méthodes `isLast` , `isFirst` , `nbClients` et `sauvegarde` :

   - `isFirst` : rend vrai si la fiche courante est la première de la liste, faux sinon.
   - `isLast` : rend vrai si la fiche courante est la dernière de la liste, faux sinon.
   - `nbClients` : rend le nombre de clients dans la collection (initialement le nombre de fiches dans le fichier).

[38] solution/gestClient-iter1/Client.java
[39] solution/gestClient-iter1/NavigatorClientFic.java
[40] clients.txt

- `sauvegarde` : place le contenu de la liste dans le fichier initial.

Voici une solution : clientjava-26-mars-2004.tgz [41]

16. Interaction avec un SGBDR

16.1. Introduction

Les programmes Java disposent d'un pont logiciel pour s'interfacer avec un SGBDR, représenté par une API nommé JDBC (*Java Database Connectivity*). Par l'intermédiaire de JDBC, vous pouvez programmer des créations de tables, des insertions et modifications de valeurs et aussi des requêtes, le tout dans un contexte de transactions avec gestion d'exceptions.

16.2. Pilotes JDBC

L'API JDBC se trouve dans *java.sql* et les pilotes doivent implémenter l'interface java.sql.Driver.

Il existe quatre types de pilotes jdbc :

- *Type 1* : Pont jdbc-odbc (livré en standard, idéal comme premier driver sous Windows)
- *Type 2* : API native + un peu de java.
- *Type 3* : Comme type 2 mais avec un protocole réseau tout en java.
- *Type 4* : Protocole natif 100% java.

Les éditeurs de SGBDR proposent leurs propres pilotes JDBC.

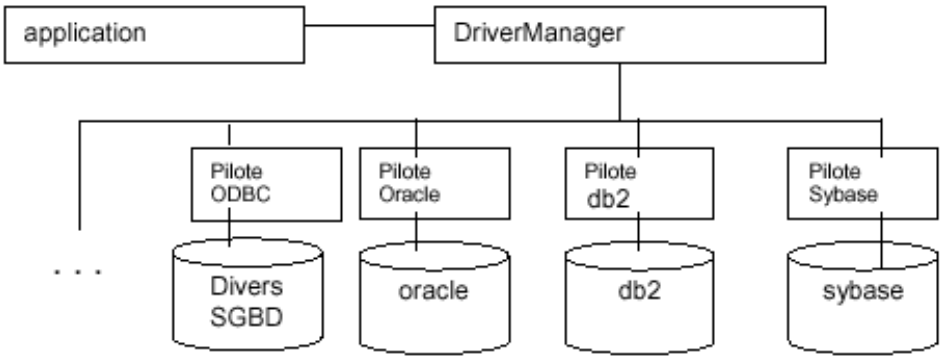


Figure 35. Exemple de pilotes

L'interaction à un système de gestion de base de données réquiert en général au moins quatre étapes :

1. **Chargement du pilote**

[41] solution/gestClient-iter2/clientjava-26-mars-2004.tgz

2. **Etablissement de la connexion**
3. **Exécution d'une requête**
4. **Exploitation des résultats**

Les étapes 1 et 2 le sont pour un ensemble d'opérations (étapes 3 et 4). Nous présentons ci-dessous chacune de ces étapes.

16.3. I - Chargement du pilote dans la JVM (*Java Virtual Machine*)

On charge généralement le pilote par son nom. Ci-dessous, un exemple de programme chargeant un des deux pilotes définis sous forme de chaînes de caractères.

```
final String driverPostgreSql = "jdbc.postgresql.Driver";
       // driver PostgreSql
final String driverOdbc       = "sun.jdbc.odbc.JdbcOdbcDriver";
       // driver odbc inclus dans le jdk
       // (pratique sous Windows pour SQL Server, Access...)
final String driverHsql       = "org.hsqldb.jdbcDriver";
       // driver Hypersonic SQL

String driver = driverHsql;

Class.forName(driver).newInstance();
  // Autochargement du driver
```

16.4. II - Etablissement de la connexion

Une fois le driver chargé en mémoire, nous pouvons obtenir une connexion via la méthode de classe `getConnection()` de la classe `DriverManager`

```
Connection con = DriverManager.getConnection(URL, "user", "passwd");
//  URL : url de connexion de la forme jdbc:sous-protocole:sous-nom
//     sous-protocole:identification du pilote
//     sous-nom  :  informations nécessaires au pilote
//                  pour la connexion (chemin, port, nom)
// "passwd" : Mot de passe
// "user"   : Nom de l'utilisateur référencé par la base
```

Exemple

```
final String driver = "org.hsqldb.jdbcDriver";
final String url = "jdbc:hsqldb:/home/kpu/hsql/refuge/refuge";
final String user="sa";
final String password=";
Connection con = null;
try {
  Class.forName(driver).newInstance();
  con = DriverManager.getConnection(url, user, password);
  ...
```

16.5. III - Exécution d'une requête SQL

L'exécution d'une requête SQL s'effectue via un objet de la classe *java.sql.Statement* . C'est l'objet `Connection` qui nous fournira une référence d'objet Statement (*à ne pas instancier directement*). Exemple :

```
Statement stat = con.createStatement();
```

On distingue deux types de requêtes : requête d'interrogation et de mise à jour.

- **Requête d'interrogation** avec l'ordre *SELECT*

  Nous utiliserons la méthode de Statement **executeQuery( )** qui retourne un objet `java.sql.ResultSet` .

  ```
  ResultSet rs = stat.executeQuery("SELECT * FROM ANIMAL");
  ```

- **Requête de mise à jour** avec les ordres *UPDATE, INSERT, DELETE*

  On utilisera la méthode **executeUpdate( )** de `Statement` .

  Cet exemple supprime de la table ENTREPRISES toutes les entreprises de Seine et Marne.

  ```
  stat.executeUpdate( "DELETE FROM ENTREPRISES WHERE CODEPOST LIKE '77%'");
  ```

16.6. IV - Exploitation des résultats

16.6.1. Requête d'interrogation avec l'ordre SELECT

Le retour d'un ordre `executeQuery(...)` est un objet de type `ResultSet` , une collection de lignes constituées de 1 à n attributs (colonnes).

Pour accéder à la première ligne du résultat, il est nécessaire d'appeler la méthode *next()* , pour passer à la ligne suivante, il suffit d'appeler de nouveau cette méthode, etc.

```
ResultSet rs = stat.executeQuery("SELECT * FROM ANIMAL");

// Pour accéder à chacun des tuples du résultat de la requête :
while (rs.next()) {
   String nom              = rs.getString("nom");
   java.sql.Date date_nais = rs.getDate("date_nais");
   int id                  = rs.getInt(1);
   ...
}
```

Remarque 1 : L'appel à la méthode *next()* de l'objet `Statement` est obligatoire *avant* tout appel aux méthodes permettant d'accéder à une valeur d'un attribut de la ligne courante.

Remarque 2 : Il y a deux façons d'accéder à une valeur d'un attribut (colonne) : 1/ soit par le *nom de la colonne* , comme par exemple les deux premiers appels de l'exemple. 2/ soit par position, qui commence *à la position 1* (et non 0 comme avec les collections), comme le montre le troisième appel.

16.6.2. Requête de mise à jour (UPDATE, INSERT, DELETE)

La méthode `executeUpdate( )` de `Statement` , ne retourne pas un objet `java.sql.ResultSet` mais retourne le *nombre de lignes impactées* par l'instruction.

Cet exemple supprime de la table ENTREPRISES toutes les entreprises de Seine et Marne.

```
int count =
  stat.executeUpdate(
    "DELETE FROM ENTREPRISES WHERE CODEPOST LIKE '77%'");
System.out.println("Il y a eu " + count + " lignes supprimées.");
```

16.7. Exemple de programme

Le programme ci-dessous utilise une base nommée *Refugedb* .

Cette base de données contient une table nommée ANIMAL; voici un script de création :

```
 CREATE TABLE ANIMAL (
   id INTEGER PRIMARY KEY,
   categorie VARCHAR NOT NULL,
   nom VARCHAR, race VARCHAR,
   sexe CHAR,
   date_nais DATE,
   id_proprio INTEGER,
   present BIT
 )
 INSERT INTO ANIMAL
   VALUES (1,'CRM', 'kiki','berger','M','2000-2-21',21,false)
 INSERT INTO ANIMAL
   VALUES (2,'CRM','rex','caniche','M','1996-12-2',11,true)
 ...
```

Lorsque l'on accède à une base de données, une gestion des exceptions s'avère nécessaire car de multiples problèmes peuvent survenir : le pilote ne peut être chargé (introuvable ?), connexion refusée, requête SQL mal formée... Voici l'exemple complet.

```
 1  import java.sql.*;
 2
 3  public class TestAnimal {
 4    public void test() {
 5      final String driver = "org.hsqldb.jdbcDriver";
 6      final String url = "jdbc:hsqldb:/home/kpu/Refuge/Refugedb";
 7      final String user = "sa";
 8      final String password ="";
 9
10      Statement  st  = null;
11      Connection con = null;
12      ResultSet  rs  = null;
13      String    sql  = "";
14      try {
15         Class.forName(driver).newInstance();
16         con = DriverManager.getConnection(url, user, password);
17         st  = con.createStatement();
18         sql = "SELECT * FROM ANIMAL";
19         rs  = st.executeQuery(sql);
20         System.out.println("ID\tTYPE\tNOM\t\tRACE\t");
21         while (rs.next()) {
22                 System.out.print(rs.getInt(1)+"\t");
23                 // ATTENTION, les indices commencent à 1.
24             System.out.print(rs.getString(2)+"\t");
25             System.out.print(rs.getString("nom")+"\t\t");
26                 System.out.println(rs.getString("race")+"\t");
27             }//while
28       }
```

```
29       catch (ClassNotFoundException e) {
30          System.err.println("Classe non trouvée : " + driver );
31       }
32       catch (SQLException e) {
33          System.err.println("SQL erreur : "+
                 sql + "  " + e.getMessage());
34       }
35       catch (Exception e) {
36          System.err.println("Erreur : "+ e);
37       }
38
39       finally {
40          try { if (con != null) { con.close(); } }
41          catch (Exception e) { System.err.println(e); }
42       }
43    }
44    static public void main(String[] arg) {
45       TestAnimal app = new TestAnimal();
46       app.test();
47    }
48 }
```

Quelques commentaires :

- *Ligne 15* : Chargement du pilote Hypersonic SQL.
- *Ligne 16* : Etablissement d'une connexion à la base.
- *Ligne 17* : Création d'un objet Statement en préparation à l'exécution d'une requête SQL.
- *Lignes 18 - 27* : Selection de toutes les lignes de la table ANIMAL.

  Affichage de quelques attributs (ID, TYPE, NOM et RACE). La condition de poursuite dans le `while` permet d'avancer à la prochaine ligne et de tester si la fin n'est pas atteinte (rend *false* alors).
- *Lignes 29 - 37* : Une gestion des exceptions.
- *Lignes 39 - 42* : A ne pas oublier, **fermeture de la connexion.**

17. Prise en main d'HypersoniSQL

Nous souhaitons faire évoluer notre application de gestion de fiches (de type Client). Nous allons lui permettre de s'interfacer avec un SGBDR.

Comme SGBDR, nous utiliserons HypersonicSQL, un projet open source.

17.1. Introduction à Hypersonic SQL

Hypersonic SQL est un petit système de gestion de bases de données écrit en Java, initialement développé par Thomas Mueller et Hypersonic SQL Group [42].

HSQL est principalement utilisé pour des démonstrations, du maquettage et de petites applications ayant besoin de s'interfacer avec un système de gestion de bases de données relationnel (SGBDR). Sa taille (<300 ko), son prix (gratuit), sa portabilité (Java) et sa rapidité de mise en oeuvre sont ses principaux atouts.

Ajoutons à ces qualités la relation privilégiée de ce système avec le langage Java, et nous avons une solution bien adaptée à un apprentissage de la programmation, portable sous Windows comme sur Linux.

[42] http://hsqldb.sourceforge.net

17.2. Installation

Téléchargez Hypersonic SQL à l'adresse suivante http://hsqldb.sourceforge.net [43].

Décompressez le fichier dans un répertoire prévu à cet effet (ex: hsqldb).

17.3. Démarrage d'Hypersonic SQL

Pour utiliser Hypersonic SQL, vous pouvez :

- Administrer une base via l'outil Database Manager livré avec Hypersonic SQL.
- Utiliser la base via une application cliente écrite en Java.

17.4. Outil Database Manager

La base de données que nous allons créer est celle qui correspond à notre fichier `Clients` .

Nous allons donc créer notre base de données. Pour cela nous utilisons Database Manager, un des outils livrés avec Hypersonic SQL (package `org.hsqldb.util`).

- *Lancer l'application*

  Le répertoire `demo` contient des fichiers permettant de lancer certains outils. Par exemple le fichier de commandes `runManager.bat` permet de lancer le Database Manager afin d'interagir avec les bases de données. Ces fichiers de commandes existent aussi pour les environnements Unix, par exemple `runManager.sh` .

  Nous vous invitons à comparer ces deux fichiers, `runManager.bat` et `runManager.sh` , leurs différences sont mineures.

  Lancer le gestionnaire.

  ```
  ./runManager.sh
  ```

  Une première fenêtre apparaît :

[43] http://hsqldb.sourceforge.net

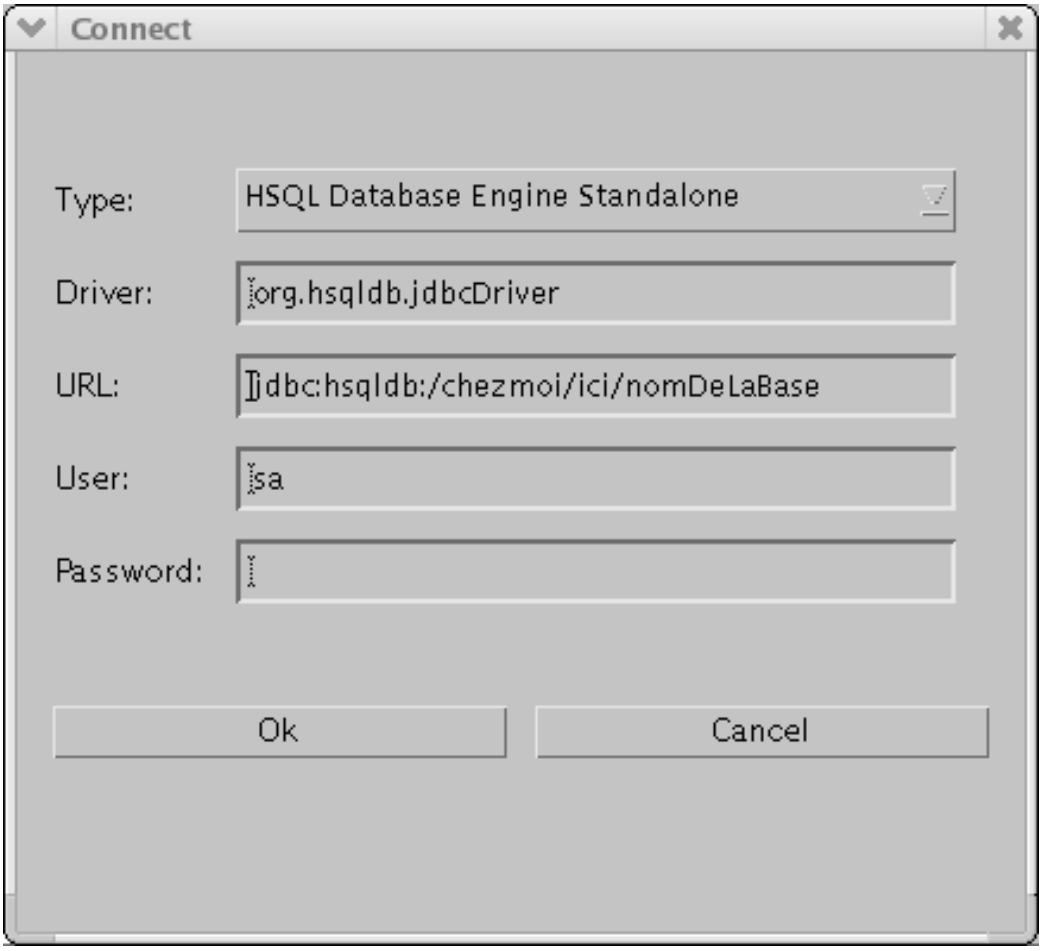


Figure 36. Manager SQL

Elle vous demande :

- Le mode dans lequel vous voulez créer ou ouvrir la base de données (Type).

  Choisir *Standalone* , en effet par défaut les données ne sont gérées qu'en mémoire (In-Memory), ce qui signifie que les informations sont perdues à la fermeture de l'application.

  En choisissant Standalone, vous utilisez HSQL en mode non partagé. En mode Client/ Serveur, nous choisirons Server ou WebServer selon le cas.

HypersonicSQL en mode serveur

Dans le cas d'une utilisation client/serveur, il faut lancer préalablement le serveur HSQL. Pour cela le plus simple est de configurer le fichier `bin/hsqldbserver` :

Exemple

```
# Step 1: Specify your dbhome here
dbhome="/home/kpu/java/hsqldb/"
# Step 2: your JDK directory.
jdkhome="/usr/java/j2sdk1.4.1/"
# Step 3: appropriate port, user and password
```

```
dbport="9001"
dbuser="sa"
dbpassword=""
# (Optional): Set user params
userparams="-database /repertoire/specifique/nomDeLaBase"
[...]
```

Puis de lancer le serveur :

```
[kpu@kpu hsqldb]$ bin/hsqldbserver start
```

Autres options : stop, status, restart et kill

Une autre façon de faire est d'entrer ces informations directement en ligne de commande :

```
$ java -classpath hsqldb.jar:. org.hsqldb.Server \
   -port 9001 \
   -database /chemin/repertoire/specifique/nomDeLaBase
```

Pour la création de la base nous nous contenterons d'une utilisation Standalone (In-Process).

- La classe du driver d'Hypersonic SQL (Driver)
- L'URL de la base de données que vous allez créer ou utiliser

  Dans l'url, le nom de la base fait suite à `jdbc:hsqldb:` . Dans notre exemple, la base se nomme `cltsdb` , dans le répertoire `/chezmoi/ici/` .

  (Dans le cas d'une utilisation client/serveur nous indiquerions l'adresse IP du serveur, comme dans cet exemple: `jdbc:hsqldb:hsql://127.0.0.1` .)
- Le nom de l'utilisateur (User)
- Le mot de passe de l'utilisateur (Password)

Vous accédez alors à la fenêtre suivante, permettant de réaliser des opérations SQL sur la base.

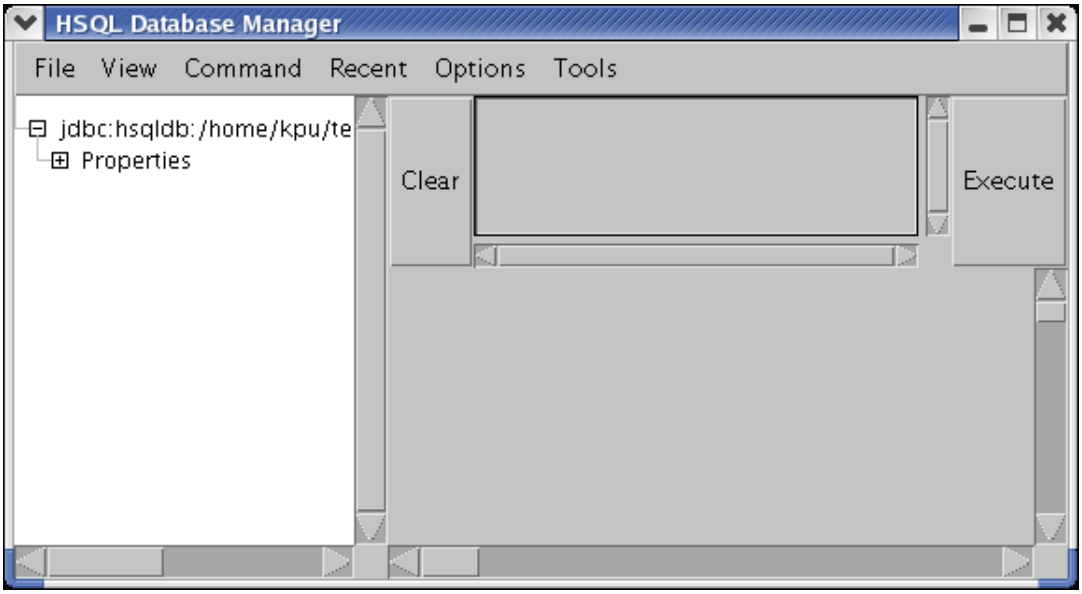


Figure 37. Database Manager (./runManager.sh)

Vous avez à votre disposition un fichier (creercltsdb.sql [44]) permettant de créer la table `client` et de la valoriser avec quelques données.

Une fois le fichier rapatrié, ouvrez-le via la commande `File->Open Script...` de l'application. Le contenu est alors automatiquement placé dans la zone d'édition de Database Manager.

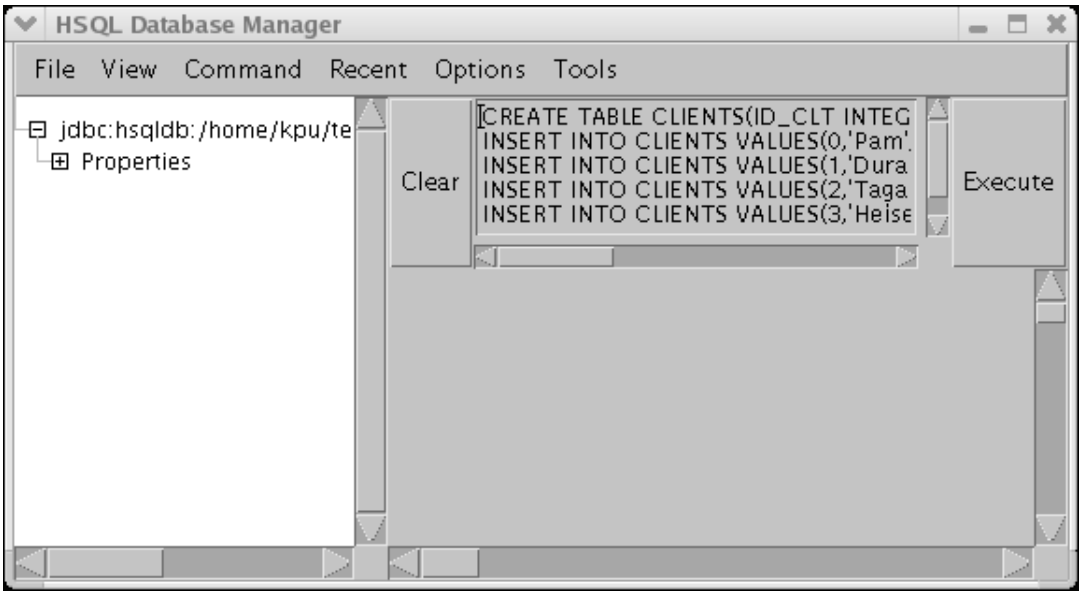


Figure 38. Database Manager (importation d'un script)

Le fichier `creercltsdb.sql` contient des ordres de création de tables, ainsi que des insertions. Il ne vous reste plus qu'à déclencher l'interprétation des ordres SQL en cliquant sur le bouton Execute.

[44] creercltsdb.sql

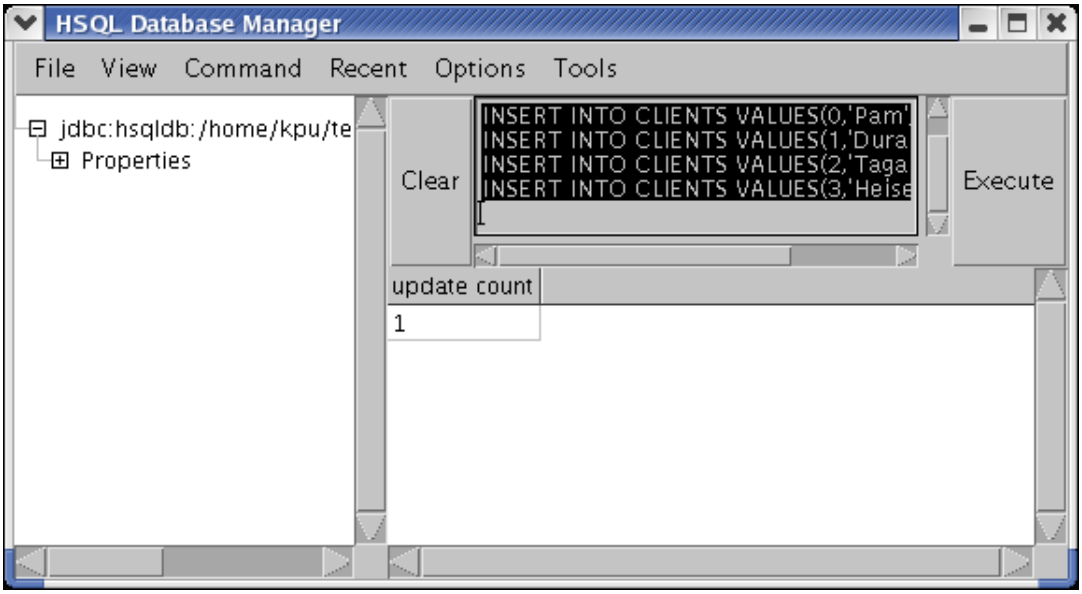


Figure 39. Database Manager (après exécution)

Vérifiez le bon déroulement de l'opération en sélectionnant `View -> Refresh Tree` . La table CLIENTS doit apparaître dans l'arbre de gauche.

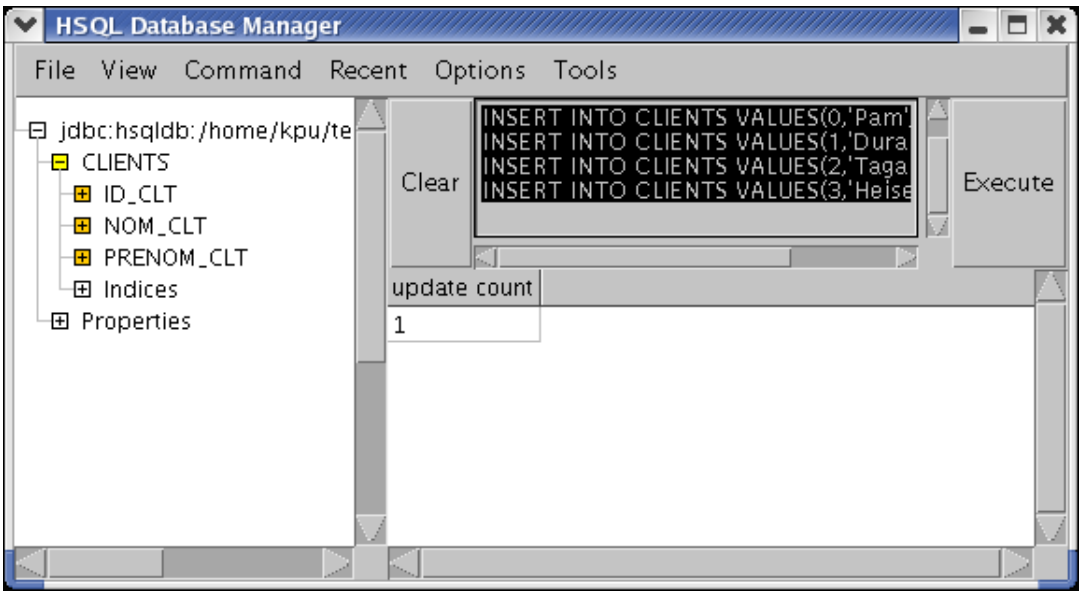


Figure 40. Database Manager (la table CLIENTS est visible)

... et en exécutant l'ordre de sélection de tous les tuples de la table `clients` .

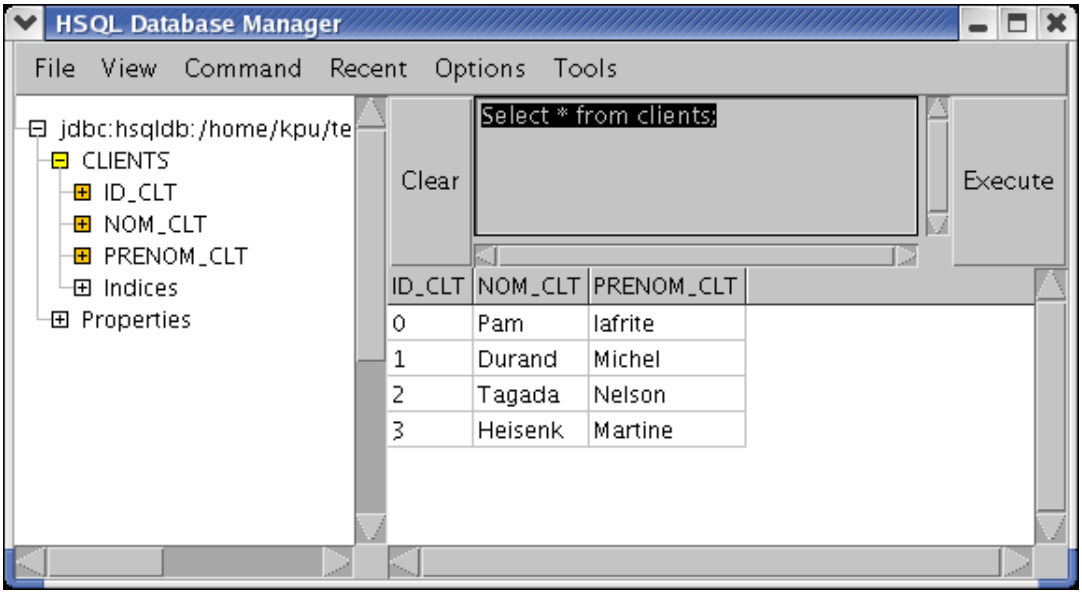


Figure 41. Database Manager (select * from clients;)

17.5. Application Test en Java

Scénario de test

1. Prérequis : la base est créée selon les indications ci-dessus.
2. Créez un répertoire, puis copiez le package `hsqldb.jar` à l'intérieur (pour faire simple).
3. Copiez le fichier TestClients.java [45]dans ce répertoire.
4. Modifier la valeur de la constante `url` de ce fichier :

```
import java.sql.*;

public class TestClients {
  public void test() {
    final String driver   = "org.hsqldb.jdbcDriver";
    final String url      = "jdbc:hsqldb:/chemin/rep/nom_de_la_base";
    final String user     = "sa";
    final String password ="";

...
```

   Afin qu'elle corresponde au nom de la base que vous avez créée précédemment (par exemple `c:\repDB\maBase`)

5. Compiler le fichier : `javac TestClients.java`
6. Exécuter le programme :

```
[dans le rep de test]$ java -classpath hsqldb.jar:. TestClients
max id = 3
ID      NOM     PRENOM
```

[45] TestClients.java

```
0       pam     lafrite
1       Durand  Michel
2       Tagada  Nelson
3       Heisenk Martine
[dans le rep de test]$
```

OK !

17.6. Conclusion

Nous avons vu comment créer une base de données pour Hypersonic SQL.

Avec Database Manager nous disposons d'un bon outil visuel pour gérer les données d'une base.

Bon à savoir

Vous trouverez dans le répertoire `doc` le fichier *hSqlSyntax.html* . Il contient la syntaxe SQL compatible avec Hypersonic SQL.

Le répertoire `demo` contient le fichier *TestSelf.txt* qui fourmille d'exemples d'ordres SQL compatibles avec Hypersonic SQL.

Hypersonic SQL tient sur une disquette, n'hésitez pas à le transporter chez vous ! (que vous soyez sous Windows ou Linux).

18. TP avec JDBC et HypersonicSQL

Nous souhaitons faire évoluer notre application de gestion des clients, en la faisant dépendre non pas d'un fichier mais d'un SGBDR.

Le diagramme de classe ci-dessous montre cette dépendance (à comparer avec le précédent)

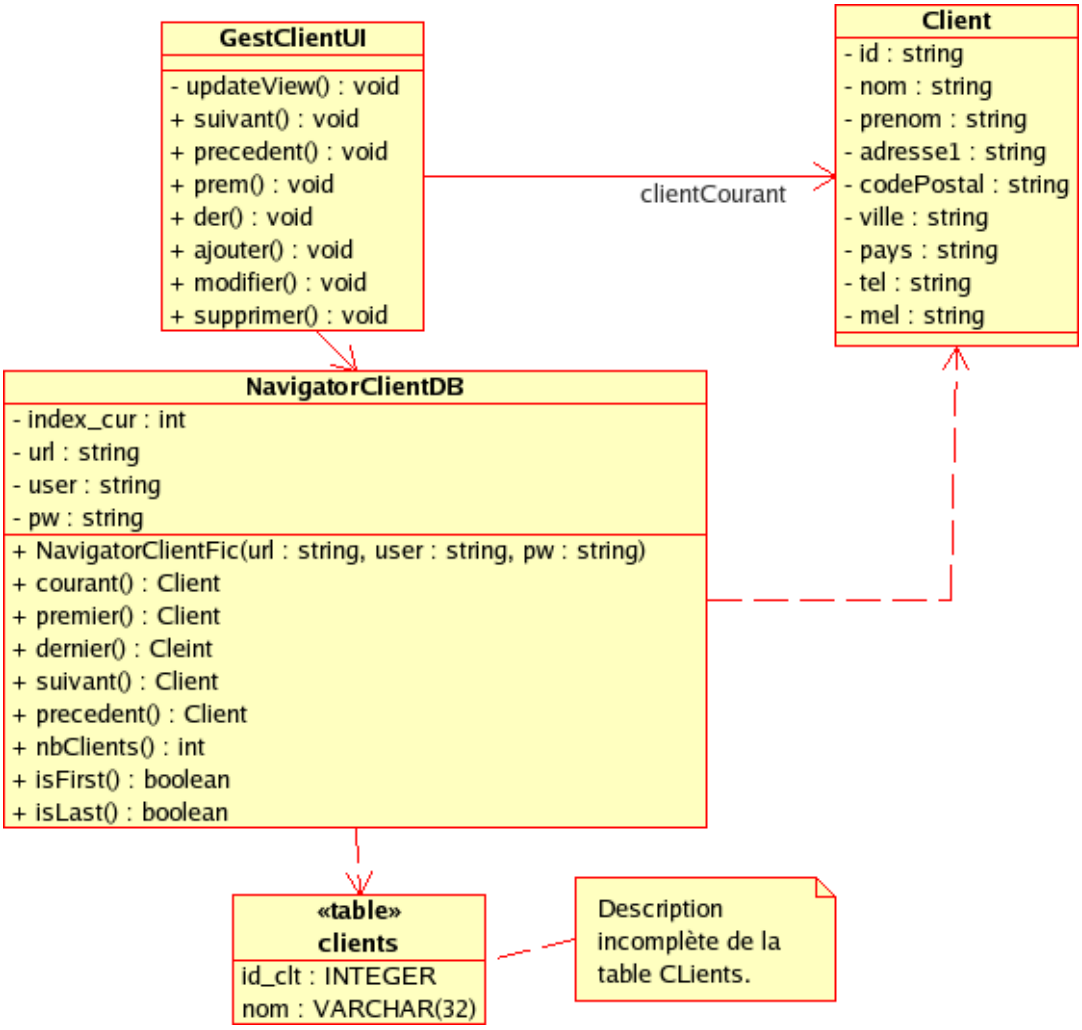


Figure 42. Diagramme de classes de l'application à développer

Si vous analysez bien la différence entre les deux versions (fichier texte et base de données), vous vous appercevrez qu'elle se situe à deux endroits : au niveau du navigateur de fiches : `NavigatorClientFic` et `NavigatorClientDB` et au niveau du Client (un attribut identifiant `id` a été ajouté).

Vous remarquerez également - **ceci est très important** - que ces deux classes ont **les mêmes déclarations d'opérations (entête de méthode)** .

On effectuera un premier changement sur la version fichier de notre application en ajoutant un attribut id aux caractéristiques d'un client. La valeur de cet attribut doit être unique pour un client (pas de doublons).

Donc, du point de vue de l'objet (la classe) qui utilise un navigateur de fiches, **seule la construction du navigateur (par new) change.** Exemple :

- Version avec Fichier texte

```
...
public GestionClient(String titre) throws Exception{
  super(titre);
  nav = new NavigatorClientFic("document.txt");
```

```
  init();
  client=nav.courant();
  updateView();
 }
...
```

- Version avec base de données

```
...
public GestionClient(String titre) throws Exception{
  super(titre);
  nav =
    new NavigatorClientDB("jdbc:hsqldb:/chezmoi/ici/labase", "sa","");
  init();
  client=nav.courant();
  updateView();
 }
...
```

Finalement, pour que ce soit aussi transparent, il faut concevoir une **interface** commune (un type) entre ces deux classes (`NavigatorclientFic` et `NavigatorClientDB`). C'est très facile car elles ont les mêmes opérations ! (à quelques aménagements près, comme expliqué plus loin).

Rappelez vous, une interface est une classe déclarant des "méthodes sans leur corps" (des opérations).

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

interface NavigatorClient{

public Client courant();
public boolean isLast();
public boolean isFirst();
public int nbClients();
public Client suivant();
public Client precedent();
public Client premier();
public Client dernier();
public void ajouter(Client c);
public void supprimer(Client c);
public void sauvegarde() throws FileNotFoundException, IOException;
}
```

Du coup, la déclaration de notre `NavigatorClientFic` devient :

```
public class NavigatorClientFic implements NavigatorClient {
  ...
  // comme avant
}
```

idem pour notre nouveau navigateur :

```
public class NavigatorClientDB implements NavigatorClient {...}
```

La méthode `sauvegarde` aura un corps vide dans `NavigatorClientDB` , car toutes les

opérations mis à jour seront immédiatement répercutées sur la base (via des requêtes SQL)

Dans la classe `GestionClient` , la déclaration du navigateur est :

```
// avant : private NavigatorClientFic nav;
private NavigatorClient nav;
```

Bon, il ne vous reste plus qu'à implémenter cela, et "coller les morceaux". Ce n'est pas forcemment un travail facile, réalisable en une séance de TP. Ce sera donc l'objet d'un mini-projet de fin d'année.

Bonne programmation.

18.1. Projet à rendre

Le projet sera composé de deux parties : rapport et code

Rapport écrit - format HTML - sur 10 points

- #### Critères d'évaluation

  - Description de l'organisation de votre travail (les étapes, qui a fait quoi)
  - Diagramme UML de l'ensemble. **Vous commenterez les relations.**
  - Compréhensible (pour une grande partie) par un lecteur non informaticien.
  - Bilan.

Code source sur 10 points

- #### Critères d'évaluation

  - Code suffisamment bien commenté
  - Originalité de vos apports personnels
  - Architecture de l'ensemble
  - Indentation, choix de nommage, mise en page